The Open University of Israel
Department of Mathematics and Computer Science

A Conflict Based SAW Method for Constraint Satisfaction Problems

Thesis submitted in partial fulfillment of the requirements
towards an M.Sc. degree in Computer Science
The Open University of Israel
Computer Science Division

By

Rafi Shalom

Prepared under the supervision of

Dr. Mireille Avigal, The Open University of Israel
Prof. Ron Unger, Bar-Ilan University

January 2009

In memory of my late father, Badri Shalom (1935-2008)

"The cheapest, fastest and most reliable components of a computer system are those that aren't there."

- Gordon Bell.

Contents

List of Figures

List of Tables

List of Acronyms

CSP	Constraint Satisfaction Problem
SAW	Stepwise Adaptation of Weights
EA	Evolutionary Algorithm
GA	Genetic Algorithm
CSAW	Conflict based SAW
HC	Hill Climber algorithm
(1+1)CSAW	The Conflict based SAW version of the (1+1) algorithm
(1+1)SAW	The Constraint based SAW version of the (1+1) algorithm
HCCSAW	The Conflict based SAW version of the HC algorithm
HCSAW	The Constraint based SAW version of the HC algorithm
rSAWEA	Stepwise-Adaptation-of-Weights EA with randomly initialized domain sets
SR	Success Rate
ACCS	Average Conflicts Checks to Solution
AES	Average Evaluations to Solution

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 evolutionary 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. The results in this thesis were submitted to the IEEE-CEC conference of 2009.

Chapter 1

Introduction

Weight adaptation originated in the context of 3-SAT solvers. In [19] Selman and Kautz introduced weight adaptation as a way to enhance GSAT [20], 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 [11] 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 [7, 10] including 3-SAT [9], and general CSPs [4]. 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 [10]. 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 [4, 2] it was concluded that using a weight per variable vs. a weight per constraint does not make a significant difference 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 [10]. 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.

Two hypotheses will be examined. The first is that an improvement is achieved by using conflict based SAW instead of constraint based SAW. This will be tested for easy CSPs as well as for hard CSPs (and for a wide range of intermediate problems). The test is done by introducing two simple EAs (both having population size 1, and straightforward representation of search points) and applying conflict based SAW and constraint based SAW to both. The second hypothesis is that given a weight for each conflict, a simple EA outperforms rSAWEA, which is the best known SAWing EA for constraint satisfaction problems. To test this hypothesis the published results of rSAWEA over a test-set is compared to the results of the best of the two conflict based SAW variants over the same test-set.

The robustness of the key parameter of SAW and its influence over the performance of constraint based SAW and conflict based SAW are checked to determine if conflict based SAW is robust, and to find out if a bad choice of the parameter might compromise the comparison between the two approaches. This also provides an estimation for reasonable values of this parameter, which we use for the above experiments.

The thesis is organized as follows : In chapter 2 we describe the basic concepts, such as constrained satisfaction problems and their related concepts, genetic and evolutionary algorithms, weight adaptation schemes, and performance measures. In chapter 3 we describe the proposed algorithms, and the rSAWEA algorithm. In chapter 4 we describe three test sets from three different models of random constraint satisfaction problems that are used as input to the algorithms. The test plan, and all experiments are detailed in chapter 5. In it we deal with parameter issues, explore the effectiveness of conflict based SAW for CSPs of varying hardness, and compare the results of a conflict based SAW algorithm with results of rSAWEA. The conclusions are presented in chapter 6.

Chapter 2

Basic Concepts

2.1 Binary Constraint Satisfaction Problems

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

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

A *binary constraint satisfaction problem* is a CSP for which all constraints are of arity two. In this thesis we limit our attention to binary CSPs, known to be computationally equivalent to general CSPs [18]. 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.

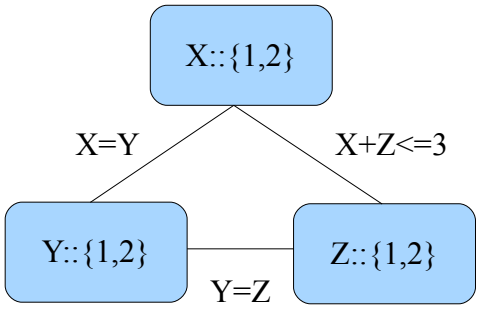

Fig. 2.1: An example of a constraint graph.

The standard graphic way to describe a binary CSP is by using its *constraint graph*, also known as its *dual graph*. In a constraint graph the nodes represent the variables and their domains, while edges represent constraints. Each edge is a relation over two variables. It is sometimes easy to represent this relation with a simple condition which is written next to the edge, or by detailing all the allowed pairs of values (which is feasible

when the domains of the variables are very small, or in case very few pairs of values are allowed). Figure 2.1 shows the constraint graph of a simple binary CSP.

This representation makes it easy to understand why all instances of the graph 3-coloring problem are binary CSPs. An instance of the graph 3-coloring problem is an undirected graph, and a solution of the instance is obtained by assigning each node one of three different colors so that no two nodes connected with an edge have the same color. The graph 3-coloring problem is known to be NP complete.

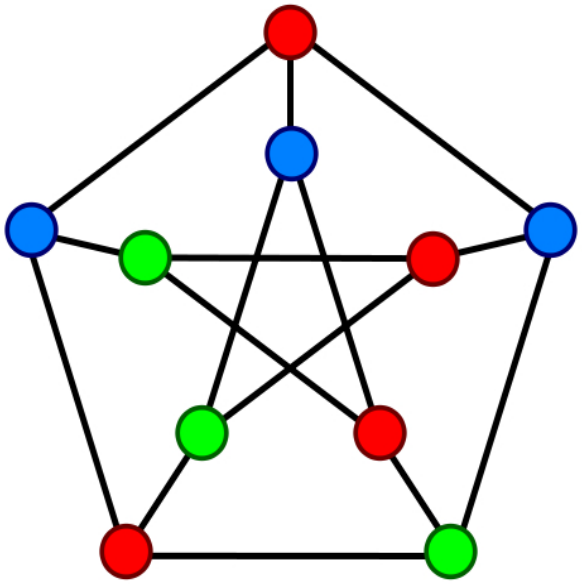

Fig. 2.2: A graph 3-coloring instance and its solution.

The constraints graph of an instance of the graph 3-coloring problem is the undirected graph of the instance. Each edge is an inequality relation between the values of its nodes. Figure 2.2 shows a graph 3-coloring instance, and its solution.

A well known example of a binary CSP is the *Queens Problem*, which is sometimes referred to as the *Queens Puzzle*. The problem asks for a way to place eight queens on a chessboard so that no queen attacks another using standard chess rules. Explicitly this means that no two queens share a row, a column or a diagonal (where a diagonal is any sequence of adjacent squares that share the same color, not only the two main diagonals). This problem is easily formulated as a binary CSP. The usual way to do this is to observe that no two queens share the same row, thus in a solution each row contains exactly one queen. Each row becomes a variable, and the value of each row is the position of a queen within the row. The position of a queen is described with the numerical value 1 for the leftmost column, which grows by one each time we move the queen one column to the right. This is a representation with eight variables $Z = \{X_1, ..., X_8\}$, each having the domain $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Note that in this problem any two variables are constrained. A simple way to describe the constraint between variables X_i and X_j is $R_{i,j} = \{(d_i, d_j) | d_i \neq d_j \wedge |d_i - d_j| \neq |i - j|\}$. It turns out that the queens problem has solutions. Figure 2.3 shows all twelve solutions of the problem unique under reflection and rotation symmetries. Otherwise there are 80 more solutions. The *N-Queens Problem* is a generalization of the queens problem so that there are N queens and the board is an $N \times N$ board.

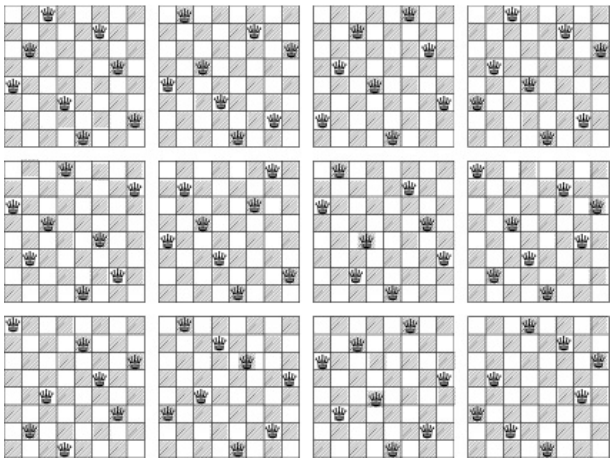

Fig. 2.3: All solutions to the queens problem.

2.2 CSP Related Concepts

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. This is done, for example, by doing a simple computation (if possible, like in the case of the N-Queens problem), keeping a matrix for each constraint, or simply by keeping a four dimensional matrix for pairs of variables and their values.

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. This value is expressed in formula (2.1). Note that since R_{i_1,i_2} is the relation that describes allowed assignments, conflicting pairs of values belong to the complement of R_{i_1,i_2}.

$$1 - \frac{|R_{i_1,i_2}|}{|D_{i_1} \times D_{i_2}|} \tag{2.1}$$

Since in our case the domain size of all variables is m, this can also be written as in formula (2.2). The average tightness (or simply tightness) of a CSP, denoted by p_2, is the average tightness of all (restrictive) constraints.

$$1 - \frac{|R_{i_1,i_2}|}{m^2} \tag{2.2}$$

For example, the density of the above representation of the queens problem is 1, because each pair of variables is constrained. This follows from the fact that for any two rows, it is possible to place queens that attack each other in those rows (by putting both on the same column for example). Each placement of a queen on a row conflicts with at least one and up to three positions in other rows. This means that the tightness of each constraint is between $\frac{1}{8}$ and $\frac{3}{8}$, and thus the tightness of the problem is also within that range. It turns out that the average tightness of the N-Queens problem is about $\frac{2.3}{N}$.

This means that even though the density of all these problems is the highest possible, the tightness diminishes as N increases, and the problems become less constrained.

2.3 Hard and Easy Constraint Satisfaction Problems

An algorithm that guarantees a solution if one exists, and able to identify a problem as insoluble is called a *complete algorithm*. The time needed for complete algorithms such as backtracking algorithms [17] 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 [22, 15].

Binary CSPs may be classified by using four parameters : n (the number of variables), m (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 of CSPs 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, EAs and stochastic search algorithms in general are not able to recognize insoluble CSPs, and they are therefore usually tested only with soluble CSPs.

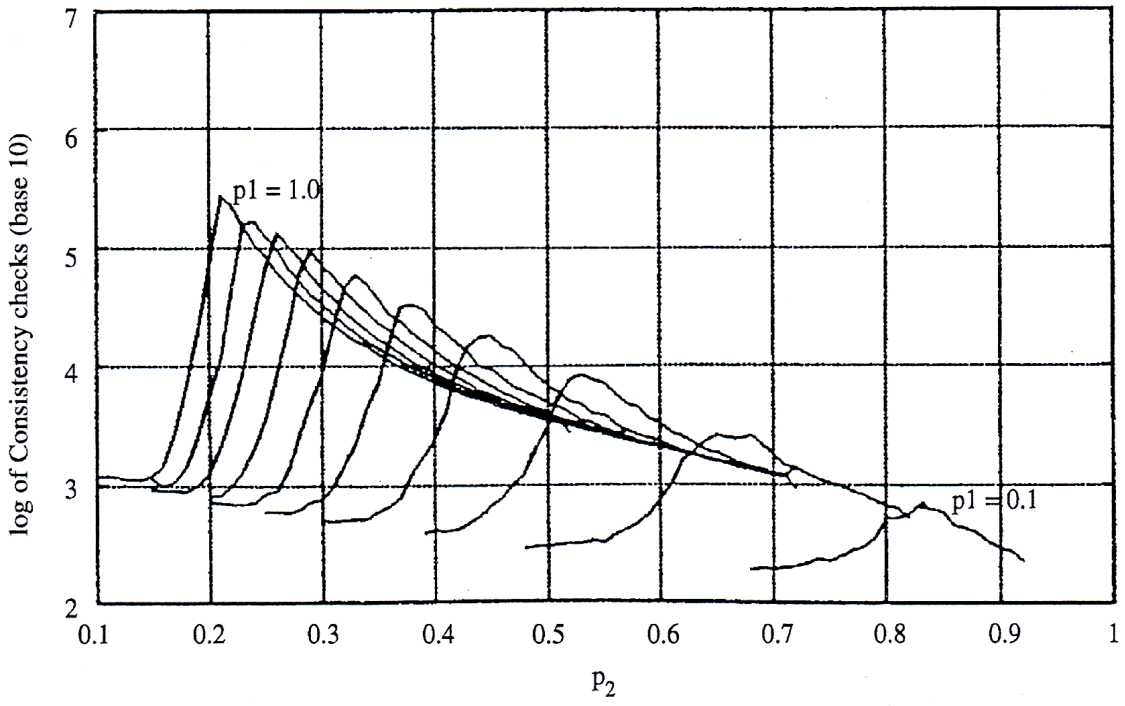

Fig. 2.4: Phase transition for various density values.

As might be expected, the higher the density is, the lower tightness has to be in order to reach the phase transition. Figure 2.4 shows the logarithm of the number of conflict checks a complete algorithm needs in order to solve CSPs with density varying from 0.1 to

1. The peak is reached at a tightness of about 0.8, and tightness of about 0.2 respectively. The illustration in figure 2.4 was taken from the learning guide of a CSP course (number 22924) of the open university of Israel.

2.4 Genetic Algorithms and Evolutionary Algorithms

Genetic algorithms (GAs) [16, 13] are stochastic optimizers that aim to find optimal or near optimal solutions quickly by simulating an evolutionary process. When designing a genetic algorithm for a specific optimization problem, the search points have to be encoded as "chromosomes" which are strings of "alleles". This may be, for example, a fixed length bit string, an array of numbers from a finite domain, or possibly an array of real numbers. These chromosomes are sometimes also referred to as *individuals*. Usually all possible search points are mapped to such a sequence so that each search point has exactly one representation.

Since Genetic algorithms are optimizers they are meant to be used for optimization problems. This means that the problem to be solved already defines an estimation of search space points. Genetic algorithms use this property of optimization problems in order to decide which chromosomes are better than others. Each chromosome is assigned a value called a *fitness value*, which is derived from the value the problem assigns to the search point it represents. It is allowed that some chromosomes would not represent a search point, as long as they are assigned a low fitness value.

2.4.1 The Basic Framework of Genetic Algorithms

A set of individuals, usually one that has a fixed size, is called a *population*. A *generation* is a population made of all the individuals the algorithm holds at the beginning of each iteration. The basic framework shared by most genetic algorithms is to randomly choose an initial generation, and use operators that simulate evolutionary behavior in order to produce new generations from former ones. The genetic algorithm does a series of iterations, each produces a new generation from a former one, halting when an optimal solution is found, or when another halting criteria is met (when a specified number of generations has been reached, when no improvement is seen for a specified time, etc.). A general flowchart of a genetic algorithm is displayed in figure 2.5.

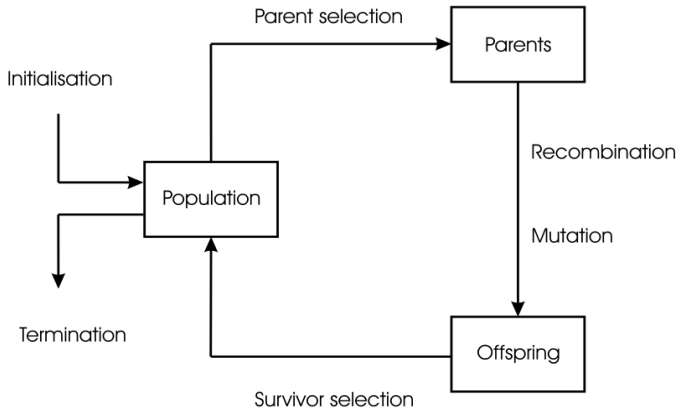

Fig. 2.5: A schematic depiction of a general genetic algorithm.

The operators used in order to create new generations from older ones are usually *selection*, *recombination* (or *crossover*) and *mutation*. There is a wide variety of these operators, just as there is a wide variety of search point representations. Several strategies for these operators are listed bellow.

The general tactics for creating a new population from a former one is usually as follows. First the selection operator is used in order to choose individuals that have a high fitness value. This move corresponds to the evolutionary idea of "survival of the fittest", and is the main engine for pushing the search forward. Recombination is then applied iteratively, each time on several selected individuals to produce new individuals. The idea of recombination is to "mix" the alleles of several selected chromosomes thus simulating the evolutionary idea of sexual reproduction. Finally, a mutation operator is applied to the resulting chromosomes by randomly changing the value of a small number of alleles. This corresponds to the evolutionary idea of duplication errors within chromosomes. The last two operators may degrade some of the chromosomes. They are used mainly to increase the diversity of the populations, thus allowing new search paths, and an escape from local optima. When the number of created individuals exceed the population size, a "survivor selection" strategy is used to pick individuals for the new generation.

2.4.2 Examples of Genetic Operators

Selection can be done in several ways. It usually prefers individuals with a higher fitness, while still allowing a selection of individuals with a low fitness. One way to do this is to determine that the probability to choose an individual is the ratio of its fitness value and the sum of the fitness values of all the individuals in the population. Another way may be to sort the individuals according to their fitness value, and to apply a choice that favors the ones at the top of the sorted list.

Some of the more frequent crossover operators are one point crossover and two point crossover. A one point crossover, shown in figure 2.6 is performed over two selected individuals called *parents*. Then it uniformly chooses a crossover point and replaces the parents with two individuals called *children* (or *offsprings*), for which the alleles before the crossover point are the ones of one parent, and the alleles from that point forward are the alleles of the other parent.

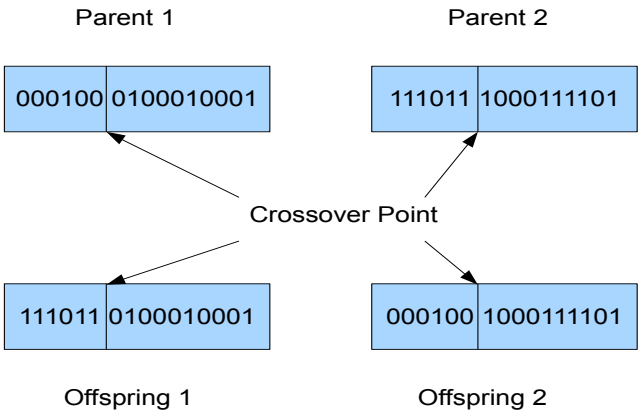

Fig. 2.6: One point crossover.

A similar two point crossover is shown in figure 2.7. In this case two crossover points are chosen and the internal segment between them is replaced in both parents by the same segment of the other parent. Many other variants were suggested that include, for example, crossovers that require more than two parents. Such a crossover is called a *multi-parent crossover*. Since the parents are "lost" using this operation, it is usually performed only at some predetermined probability called the *crossover rate*.

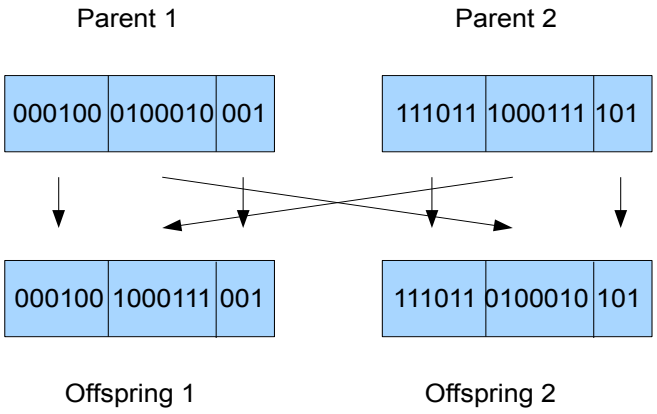

Fig. 2.7: Two point crossover.

Mutation operators also vary. Mutation is usually performed on a single individual, and it may even consider alleles as the operand. A simple mutation for the bit string representation may be a mutation that flips all alleles with a very small probability called the *mutation rate*. This kind of mutation leaves some individuals unchanged while allowing others to be changed in more than one allele. Another way to mutate individuals is by swapping two different alleles of the same individual, as seen in figure 2.8. This works well for non-binary representations as well as for binary representations, and also for permutations.

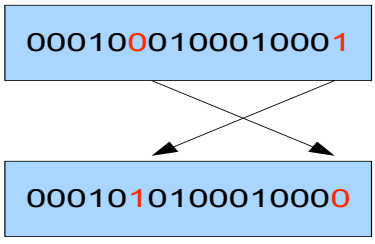

Fig. 2.8: Mutating an individual by swapping alleles.

A $(\mu + \lambda)$ GA is a GA 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) GA a new generation of size μ is selected only from the λ individuals generated.

2.4.3 Evolutionary Algorithms

Evolutionary algorithms (EAs) is a more general term than genetic algorithms. Evolutionary algorithms include, apart from Genetic algorithms, also *Evolutionary Strategies, Evolutionary Programming,* and *Genetic Programming.* A discussion of the differences between them can be found in [8].

$(1 + 1)$ and $(1, \lambda)$ EAs, that use no crossover, were traditionally used with SAW [9, 10]. A $(1 + 1)$ algorithm with a permutation of the variables as the representation, and a mutation which swaps two uniformly chosen variables, was initially found to work best for SAWing algorithms applied to the graph 3-coloring problem. However, using a population of a single individual has an inherent lack of diversity which proved harmful when SAW was used for CSPs. For CSP solvers, EAs with only one individual in the population perform worse than EAs with a larger population, and tend to get stuck at local optima [2].

We will use $(1+\lambda)$ EAs as a basis for conflict based SAW. Comparing the results with the ones of their constraint based SAW versions, will determine if the local optima problem occurs for conflict based SAW. A comparison with rSAWEA (which avoids the local optima problem by keeping a larger population), will show how much of an improvement a simple conflict based SAW algorithm can provide.

All the algorithms we present and survey here are related to genetic algorithms, even though some of them use a population of a single individual, and all of them do not use crossover. However, since evolutionary algorithms is a broader notion, all of them fit the notion of an EA.

2.5 Weight Adaptation Methods

Weight adaptation was found useful for improving stochastic search algorithms including 3-SAT solvers such as GSAT [19, 12, 11], and EAs designed to solve a variety of problems [10, 7, 9]. Being a general method, weight adaptation can be customized for a specific problem solver. 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 $\chi : S \times V \to \{0, 1\}$ 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.

Experiments [10, 9] show that weight adaptation is a robust technique. Hardly any difference is noticed when using different Δw values, and the usual choice of it is $\Delta w = 1$. 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.

When the above approach is used for 3-SAT [9, 11, 19], V is the set of clauses, S is the set of 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 represents all possible assignments to variables, and V is the set of constraints. The other approach is to let S be the set of 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) [3, 5, 6], which is the best known SAW algorithm for CSPs, uses it as well. This approach was also found to be superior under extensive experimentation [2].

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[12, 11, 14], which will not be surveyed here. In brief, refinement means adding a function to the objective function that allows differentiating search points that would otherwise evaluate the same. Decay means lowering the value of weights over time so new weight increments have a stronger impact than older ones. Refinement and decay were found not helpful for CSPs when implemented both with variable based SAW and with constraint based SAW [4]. Weight exponentiation led Frank to the idea that the search should focus on variables of unsatisfied clauses, which he implemented as a weight adaptation version of GSAT called UGSAT [11]. This is exceptional because different T_p values are determined and used at runtime. The combined effect of refinement, decay and optimizing variables of unsatisfied clauses with conflict based SAW is left for further research.

One of the main concerns in this thesis is the impact of using weights per conflict instead of the older applications of weight adaptation for CSPs which used a weight per variable or per constraint. 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. The schemes are tested using two simple EAs. 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 also discussed, and the choice of T_p is tested in our experiments.

2.6 Performance Measures

The most important measure is SR (success rate) which measures the percentage of runs at which a solution was found. Since evolutionary algorithms, and stochastic algorithms in general, usually cannot provide a way to detect insoluble instances, they are usually tested only with soluble ones. Accordingly, the test-sets we used here contain only soluble CSPs. This ensures the possibility of reaching 100% success rates.

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. The ACCS measure is secondary to SR because the primary objective of a CSP solver is to solve as many CSPs as possible, and/or solve them with a higher probability. Consequentially, the ACCS measure is of little importance when success rates differ considerably.

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 [17]. For the purpose of comparing constraint based SAW and conflict based SAW 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, because it is finer, and because for the algorithms suggested here it is runtime proportional.

Chapter 3

Algorithms

3.1 The Proposed Algorithms

In order to compare conflict based SAW with constraint based SAW 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, they resemble hill-climbing algorithms for which weight adaptation was originally used. In [9] Eiben et. al. noted that GSAT can be interpreted as a $(1, n)$ algorithm.

Though keeping a population of size 1 may seem problematic in the context of genetic algorithms, it was argued upon the introduction of SAW [10] (as a weight adaptation scheme for EAs), 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 conflict based SAW with constraint based SAW.

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.

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 keep 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 *violations*, 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 *violations*. 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). The *violations* variable can now be updated to hold the number of constraint violations of s_2 by letting $violations \leftarrow violations - CV_{s_1,X} + CV_{s_2,X}$. 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, it appears that for both HC and $(1+1)$ the individual selected at each iteration is needlessly re-evaluated at the beginning of the following iteration. This can be avoided by tracking weight sums for each variable, and keeping lists of currently violated constraints per variable. The result is about 50% less conflict checks for the $(1+1)$ algorithms and a slight improvement of the HC algorithms. There are several ways to maintain the needed data, even ones that keep CPU time strictly proportional to the conflict check count. Yet, since constants per conflict check are originally very low the entire operation becomes insignificant or even harmful in terms of CPU time. 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.

```
Violations(S)
  v ← 0
  for each (v1, v2) ∈ C do
      if cc(v1, v2, S[v1], S[v2])
          v++
  return v
```

Fig. 3.1: **Violations** routine : Full estimation of violated constraints.

The pseudocode of several routines our algorithms use appear on the left. But first, a few clarifications. We use $Variable++$ to denote the increment operation ($Variable \leftarrow Variable + 1$), and $Var1+ = Var2$ instead of $Var1 \leftarrow Var1 + Var2$. Search points are represented by the letter S. S is an array that contains the values of all the variables. Accordingly $S[v]$ denotes the value of variable v. As mentioned, the algorithms use a list of constrained pairs of variables. The list is denoted by the letter C. The algorithms also use a list of variables that share a constraint for each variable. C_v denotes this list for variable v. The function *cc* performs conflict checks. It receives two variables and their respective values, and returns true if and only if the assignment of these values

```
LocalConflicts(S, v)
  r.v ← 0
  r.w ← 0
  for each v' ∈ C_v do
      if cc(v, v', S[v], S[v']) then
          r.v++
          r.w+ = weight(v, v', S[v], S[v'])
  return r
```

Fig. 3.2: **LocalConflicts** routine : Constraints and weights relevant to a single variable.

to the variables constitutes a conflict. The function cc is also responsible for updating the conflict checks variable $CCcount$. Every time cc is called it performs the operation $CCcount++$.

Weights are accessed using the term $weight(v_1, v_2, S[v_1], S[v_2])$ where v_1 and v_2 are two variables. In the SAW variants of the algorithms, the values of the variables are ignored, and the weight is determined only using the variables. The weights are held in a matrix supposed to be initialized to one for all pairs of constrained variables, and zero otherwise. In the CSAW variants, all four values are used, and different weights are accessed for different values of the same variables. In this case, a four dimensional matrix is used, for which all entries denoting a conflict are initialized to one, and all other entries are initialized to zero. Since in both cases entries supposed to be initialized to zero are never accessed, they are allowed to hold garbage values.

If no weight adaptation is used, the only change is that the routine **ChangeWeights** is never called, or alternatively does nothing. In this case, it is also possible to replace the weight access in routine **LocalConflicts** with the constant value 1, and avoid using any weight matrices.

The algorithms use the constant values $\Delta w, T_p$ and $maxCC$. The first two are weight adaptation parameters, and the third determines the number of conflict checks allowed.

Let us now turn to the helper routines seen on the left. The first is called **Violations**, which appears in figure 3.1. It returns the number of violated constraints in the search point S. This is done by going over the list of all constraints.

```
HillClimb(S, v)
  saved ← S[who]
  bestvalue ← S[who]
  best ← LocalConflicts(S, v)
  initial ← best.v
  for each k ∈ D_v while best.v ≠ 0 do
      if k ≠ saved then
          S[v] ← k
          current ← LocalConflicts(S, v)
          if current.w ≤ best.w then
              best ← current
              bestvalue ← k
  S[v] ← bestvalue
  return initial − best.v
```

Fig. 3.3: **HillClimb** routine : Optimizing a single variable. Used only by **HC**.

Another auxiliary routine, called **LocalConflicts** is described in figure 3.2. It receives a search point and a variable, and returns a record r that contains two values v and w (denoted by $r.v$ and $r.w$ respectively). The first value is the number of constraints violated by the value of v, and the other is the sum of the weights that involve variable v. Since only costs relevant to v are counted, it is sufficient to check only the variables that share a constraint with v.

The **HillClimb** routine which appears figure 3.3 is used only by HC and its variants. It receives a search point and a variable and optimizes the value of the variable. This is done by computing the costs for the current value of the variable, and then going over all the other values and updating the *bestvalue* variable whenever the sum of the weights is at least as good as

```
ChangeWeights(S)
  for each (v_1, v_2) ∈ C do
      if cc(v_1, v_2, S[v_1], S[v_2]) then
          weight(v_1, v_2, S[v_1], S[v_2]) += Δw
```

Fig. 3.4: **ChangeWeights** routine : Performing weight adaptation.

what we have. The routine is slightly improved by halting the optimization when the variable is fully optimized, i.e. when a value that violates no constrained is reached. Even so, for the purposes of counting the number of evaluated individuals, the **HillClimb** routine is assumed to have evaluated exactly $m - 1$ individuals. The **HillClimb** routine returns the difference between the number of constraints violated by the replaced value and the number of constraints violated by the new value. This is the number by which the total number of violations in S decreased as a consequence of the changed value of the optimized variable.

```
(1 + 1)
  CCcount ← 0
  Choose a random start point and store it in S
  violations ← Violations (S)
  generation ← 1
  while violations ≠ 0 ∧ CCcount < maxCC do
      Randomly choose a variable and store it in v
      Randomly choose a value val ≠ S[v]
      before ← LocalConflicts(S, v)
      saved ← S[v]
      S[v] ← val
      after ← LocalConflicts(S, v)
      if after.w > before.w then
          S[v] ← saved
      else
          violations+ = after.v − before.v
      if (generation mod T_p) = 0 then
          ChangeWeights (S)
      generation + +
```

Fig. 3.5: $(1 + 1)$ algorithm.

```
HC
  CCcount ← 0
  Choose a random start point and store it in S
  violations ← Violations (S)
  generation ← 1
  while violations ≠ 0 ∧ CCcount < maxCC do
      Randomly choose a variable and store it in v
      violations− = HillClimb(S, v)
      if ((generation ∗ (m − 1)) mod T_p) = 0 then
          ChangeWeights (S)
      generation + +
```

Fig. 3.6: **HC** algorithm.

Weight adaptation is performed using the **ChangeWeights** routine shown in figure 3.4. The routine receives a search point, goes over the list of constraints, and increases the values of the relevant weight by Δw whenever a constraint is violated.

Finally, there are the two algorithms (1+1) and **HC** in figures 3.5 and 3.6 respectively. The code is fairly self explanatory. The variable m that appears in **HC** is the domain size of the variables.

3.2 The rSAWEA Algorithm

In order to test the improvement a conflict based SAW method can provide, a comparison of a CSAW variant with the best known SAWing EA for CSPs appears to be the most relevant. In this section we provide a detailed description of the rSAWEA (Stepwise-Adaptation-of-Weights EA with randomly initialized domain sets) algorithm which is currently the best SAWing EA for CSPs.

rSAWEA was presented by the name SAWEA r2 as part of a comprehensive study of EAs for CSPs [3]. It was later used for comparisons with better EAs for CSPs, namely STLEA (Simple Tabu List Evolutionary Algorithm) and CTLEA (Conflict Tabu List Evolutionary Algorithm) in [5] and [6] respectively, under the name rSAWEA. Even though it is possible to show that one of the proposed CSAW variants is better than the best EA for CSPs in general, and that it is competitive with complete algorithms, such a comparison is beyond the issue of improving the SAW technique, and thus beyond the scope of this thesis.

We begin by giving a detailed description of an algorithm named SAWEA. rSAWEA is a variant of SAWEA, and differs only in the way variable domains are handled. The top level pseudo-code of SAWEA is shown in figure 3.7.

SAWEA is an evolutionary algorithm for CSPs that keeps a population of ten individuals. The individuals are permutations of the variables of the CSP to be solved. For now, we will concentrate on the evolutionary aspects of the algorithm, and return to the way permutations are used later.

```
SAWEA(max_evaluations)
  evaluations ← 0
  P ← initialize
  generation ← 1
  while ¬contains_solution(P) ∨ evaluations < max_evaluations do
      S ← select_parents(P)
      S ← mutate(S)
      evaluations ← evaluations + evaluate(S)
      P ← select_survivors(P, S)
      if generation mod 25 = 0 then
          adapt_weights(P)
      generation + +
```

Fig. 3.7: The **SAWEA** algorithm - the basis of rSAWEA.

SAWEA receives a parameter called *max_evaluations* that determines the time the algorithm is allowed to use in order to find a solution. The variable *evaluations* tracks the number of evaluations of individuals.

A general description of the flow of SAWEA is as follows : Before the main loop starts, SAWEA sets *evaluations* to zero, and an initial population is randomly selected. The algorithm is run until a solution is found, or until *max_evaluations* is reached, whichever comes first. The creation of a new generation from an older one is performed by first selecting a parent population for mutation using the procedure *select_parents*, mutating the selected individuals using the procedure *mutate*, and using replace worst survivor selection via the procedure *select_survivors*. The function *evaluate* evaluates all the individuals in the population it receives, and returns the number of evaluations performed. There is no crossover operator.

Initial selection is done using the procedure *select_parents*. It selects with repetitions ten individuals using *biased ranking* selection with a 1.5 bias. Biased ranking was introduced in [21]. Biased ranking selection is performed by sorting the population according to fitness, and selecting at random individuals in a way that favors the top of the list. Biased ranking uses a function that transforms pseudo-random numbers distributed uniformly within $[0, 1)$ to a number in the same range with a higher probability for lower numbers. The function is shown in equation 3.1. Using this function, and given a uniformly distributed pseudo-random number r in the range $[0, 1)$, the index of the selected individual is defined by $\lfloor population\_size \cdot f(r) \rfloor$.

$$f(x) = \frac{bias - \sqrt{bias^2 - 4 \cdot (bias - 1) \cdot x}}{2 \cdot (bias - 1)} \tag{3.1}$$

The variable *bias* in equation 3.1 is the selection bias parameter. It may have values in the range $(1, 2]$. It is easy to verify using basic calculus that when *bias* approaches 1, $f(x)$ approaches x. Thus when selection bias is close to 1, biased ranking is close to having no selection pressure, i.e., letting all individuals have the same selection probability. It is also fairly easy to verify that $bias \geq 0$ implies $f(0) = 0$, and that $f(1) = 1$ whenever $bias \leq 2$. This means that when *bias* receives legitimate values, f is a transformation from $[0, 1)$ to itself. The higher *bias* is, the more $f(x)$ is biased towards lower values, which results in higher selection pressure. In [21] it is reported that a bias of 1.5 turned out to be preferable. This value was used for SAWEA. The meaning of setting $bias \leftarrow 1.5$ is that the top ranking individual is 1.5 times more likely to be selected than the median individual.

The selected population is mutated using the *mutate* procedure. Every individual in the population is mutated exactly once. A mutation is performed by swapping two uniformly selected variables of the permutation.

After the new ten mutated individuals are evaluated, the function *select_survivors* is used on twenty individuals. Ten of them are the individuals of the current generation, and the other ten are the ten selected and mutated individuals. Selection is done with replace worst survivor selection, which means that individuals with lower fitness of the current generation are replaced with better ones from the ten individuals created by selection and mutation. The function returns a new population of ten individuals, which is the next generation.

In order to use weight adaptation, SAWEA uses the parameter $\Delta w = 1$, and it performs weight adaptation once every 25 generations, which is derived from the traditional $T_p = 250$ (since population size is 10). As usual when SAW is applied to EAs, weight adaptation is done by the best individual of the relevant population.

SAWEA uses a count of the number of evaluations for the stop condition of the algorithm, but a conflict checks measure for the runtime is measured and reported. In

order to keep a fair accounting of the actual runtime, weight adaptation is counted as a single evaluation, and the conflict checks needed in order to perform weight adaptation are included in the ACCS measure. Most of the computational effort is done by the decoder which interprets and evaluates the permutations. The details of the decoder and the handling of variable domains in SAWEA and rSAWEA follow.

SAWEA uses the permutation representation. Each individual is a permutation of the variables, and a decoder is used in order to assign values to the variables. Variables are assigned according to their order in the permutation. Each variable receives the first value consistent with previous assignments, and otherwise considered violated. The assignment 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 [10, 4, 2] showed no advantage for a constraint based SAW over variable based SAW.

The original version of the decoder that was used for graph 3-coloring [10] always assigns values in the order of a static domain. This worked well for a $(1 + 1)$ algorithm applied to the graph 3-coloring problem for which the values of the variables are symmetric. The symmetry ensures that the decoder, though always assigning the first value to the first variable in the permutation, does not exclude a solution. Another reason which makes the decoder suitable for the graph 3-coloring problem is that there are only 3^n possible assignments and $n!$ permutations, which is an overrepresentation, yet does not seem to hurt performance. In CSPs, however, the values are usually not symmetric, and when the domain size is equal to the number of variables there are n^n possible assignments. SAWEA also uses a static domain for the variables.

In rSAWEA the decoder was enhanced by checking and testing several options for a dynamic assignment of the values [3]. 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. Determining at runtime which violation entities are relevant also means that rSAWEA does not conform with the weight adaptation model that was presented in section 2.5.

Chapter 4

Test-sets

Three test-sets of randomly generated CSPs from three different models are used to measure performance. Their contents and objectives are detailed in this chapter.

4.1 Model F Problems

For a comparison of the proposed CSAW algorithms with rSAWEA we use a model F [15] test-set of soluble phase transition CSPs with 10 variables of domain size 10. A model F binary CSP is created by selecting uniformly, independently and with repetitions, $p_1 p_2 m^2 \binom{n}{2}$ conflicts. Later, exactly $p_1 \binom{n}{2}$ pairs of variables are chosen to be the only pairs of constrained variables, and all other pairs become unconstrained again (by ignoring all chosen conflicts concerning them).

The test-set consists of nine categories, denoted by *F1, F2, ..., F9*, each holding 25 problems. The categories have the following density-tightness combinations F1:(0.1,0.9), F2:(0.2,0.9), F3:(0.3,0.8), F4:(0.4,0.7), F5:(0.5,0.7), F6:(0.6,0.6), F7:(0.7,0.5), F8:(0.8,0.5), F9:(0.9,0.4). The test-set was created by Craenen [3], and is downloadable at : *http://www.emergentcomputing.org/csp/testset_mushy.zip.*

4.2 Model RB Problems

Another phase transition test-set is an *RB* model [22] test-set, that has five instances of soluble phase transition binary 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 of 2007 and is downloadable at : *http://www.nlsde.buaa.edu.cn/ kexu/benchmarks/frb30-15-cnf.tar.gz.*

RB is a model that allows the creation of CSPs with constraints of arity $k \geq 2$, and uses two parameters, r and q. Since in our case we deal only with binary CSPs, we describe the RB model only for the special case where $k = 2$. The parameter r determines the number of constraints in the CSP instance, while p determines how restrictive the constraints are. The generation of an RB binary CSP instance is done by first selecting with repetitions $rn\ln n$ random constraints. The selection of a constraint is done by selecting without repetitions two of n variables. For each chosen constraint we select uniformly and without repetitions pm^2 conflicts.

In order to create phase transition CSPs the values of r and p can each be determined given the other, and given α which describes the relation between the number of the variables and their domain size according to $m = n^\alpha$. The way to set them, including

the theoretical justification, is provided in [22].

The choice of the parameters in the specific test-set was done in the following way : all test-sets created for the annual SAT competition used $\alpha = 0.8$ (Note that 15 roughly equals $30^{0.8}$). The choice of r was $r = \frac{\alpha}{\ln 4 - \ln 3}$. According to theorem 2 in [22], in order to get a phase transition CSP given r and α, one should set $p = 1 - e^{-\frac{\alpha}{r}}$ which in our case means letting $p = \frac{1}{4}$.

4.3 Model E Problems

The third test-set contains a model E [1] test-set created by Craenen, and used in [4, 2]. Model E employs only one parameter instead of a density-tightness combination to control the number of conflicts in the generated random binary 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. Model E is in fact a special case of model F with $p_1 = 1$.

The test-set contains ten categories, denoted by *E1, E2, ..., E10*. 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 [15], so we expect a density of 1 in all the problems of this test-set. The test-set is downloadable at : *http://www.xs4all.nl/ bcraenen/resources/csps_modelE_v15_d15.tar.gz*.

Chapter 5

Experiments

5.1 The Test Plan

The test plan is composed of three types of experiments. Tests are performed by running the four proposed variants (a conflict based SAW and a constraint based SAW variant for both $(1+1)$ and HC) on test-set categories. Given an algorithm and a test-set category, every instance within the category is run 50 times. The number of successful runs is used for the computation of the SR measure, and the conflict check counts of the successful runs are used for the ACCS measure.

We begin with tests of the algorithms with varying T_p values. These tests are shown in section 5.2. We aim to find a good estimation for the T_p parameter, and to check how the choice of specific T_p values affect the comparison of conflict based SAW with constraints based SAW. The test is performed over a limited number of categories of all test-sets. It is performed for the RB model test set, which has only one category, for categories F2, F5, and F8 of the model F test-set, and categories E1, E4, E7, and E10 of the model E test-set.

The second type of experiments check the impact of conflict based SAW vs. constraint based SAW when applied to CSPs of varying hardness. Here we compare the results of the four variants over the model E test-set. We also use different runtimes to investigate the effect of both weight adaptation schemes as the time allowed increases. These experiments are shown in section 5.3.

Finally, section 5.4 shows a comparison of the best of the two conflict based SAW variants compared with its constraint based SAW version over the model F test set. This provides a way to determine weather the local optima problem of the $(1+\lambda)$ EAs still occur when a conflict based SAW is used. The same conflict based SAW algorithm is also compared with rSAWEA over the same test set in order to find how much of an improvement is gained when former versions of the SAW technique are compared with a conflict based SAW algorithm.

5.2 Parameter Related Experiments

As mentioned in section 2.5, weight adaptation requires the parameters Δw and T_p. Experiments for different values of Δw consistently shows very strong robustness, and the usual choice of $\Delta w = 1$ seems safe enough. 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, i.e, that the same algorithm does not have different performance peaks for significantly different T_p values when constraint based SAW and conflict based

SAW are applied to it. The other reason is that T_p is known not to work well for very small values, and 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 $0 \leq i \leq 14$. Note that HC is assumed to create $m-1$ search points at every iteration in order to perform the stochastic hill-climbing. Thus the $(1+1)$ algorithm performs a weight adaptation once every $\beta n(m-1)$ iterations while HC does this only once every βn iterations.

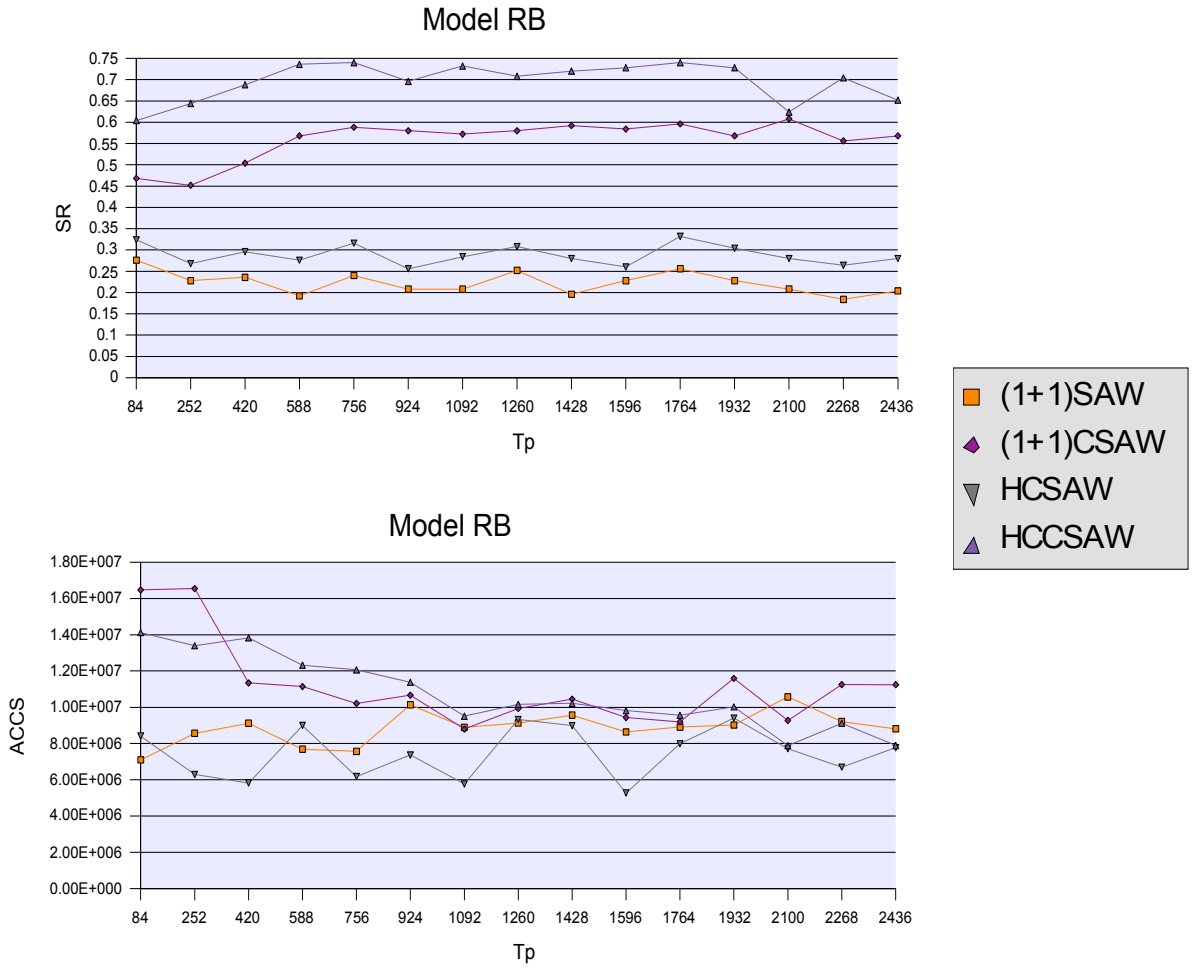

Fig. 5.1: The influence of T_p over the RB model test set, for which $n = 30$ and $m = 15$. The values of T_p are problem size dependent and range from 0.2n(m-1) to 5.8n(m-1). maxCC is set to 30,000,000.

Results for the RB model and $maxCC = 30,000,000$ are shown in figure 5.1. It appears that the traditional value $T_p = 250$ may be too low when conflict based SAW is applied to problems of this size. The graphs indicate this because it appears that the CSAW algorithms are sensitive to low values of T_p, and only when T_p reaches a value of 588 the SR and ACCS measures begin to stabilize. This justifies a choice of T_p that

takes problem size into account. Even though the CSAW algorithms perform worse than the SAW algorithms in terms of ACCS, they perform significantly better in terms of SR, which is the primary measure. When the success rate gap is as clear as in our case, the ACCS measure is of no significance.

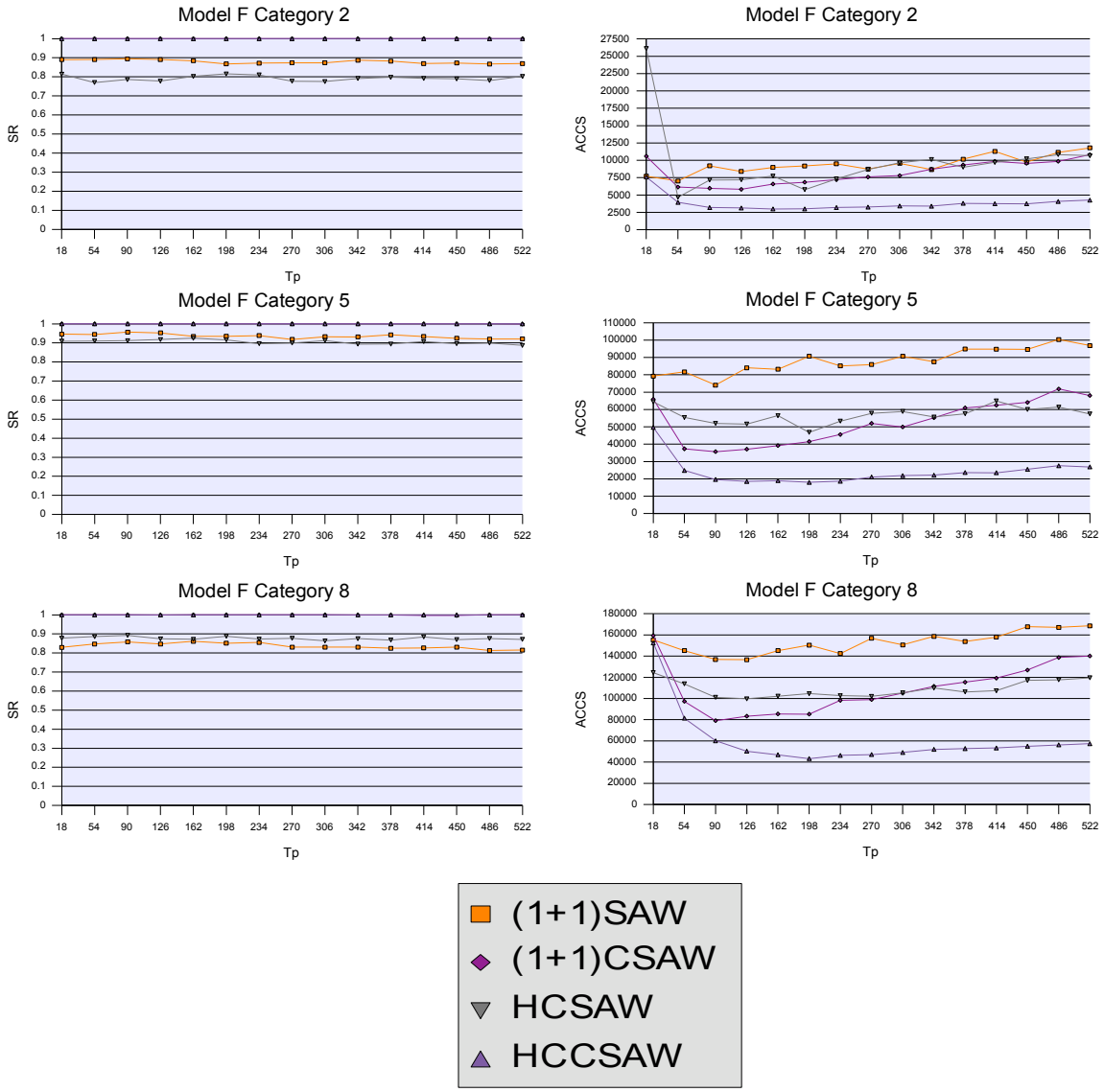

Fig. 5.2: The influence of T_p over categories F2, F5 and F8, for which $n = 10$ and $m = 10$. The values of T_p are problem size dependent and range from 0.2n(m-1) to 5.8n(m-1). maxCC is set to 1,000,000.

Figure 5.2 shows the results for all four suggested variants on three of the nine model F categories, namely, categories F2, F5 and F8, with $maxCC = 1,000,000$. Note that all model F categories are supposed to be close to the phase transition, yet category 2 is experimentally known to be easy to solve, so the results for it do not necessarily indicate results for hard CSPs. The graphs on the left show the success rates of the algorithms on the three categories. The success rate results show that the two conflict based SAW variants always have a 100% success rate, while the constraint based SAW variants get

close but constantly fail to achieve total success. The ACCS results show that only in category 2, $(1+1)$SAW becomes sensitive to a low T_p value. Otherwise the CSAW variants are more sensitive to low T_p values. Once $T_p = C$ is reached, all algorithms begin to stabilize. However, the $(1+1)$ variants suffer more from a substantial increase of T_p, whereas the HC variants are not affected much even when T_p reaches $5.8C$.

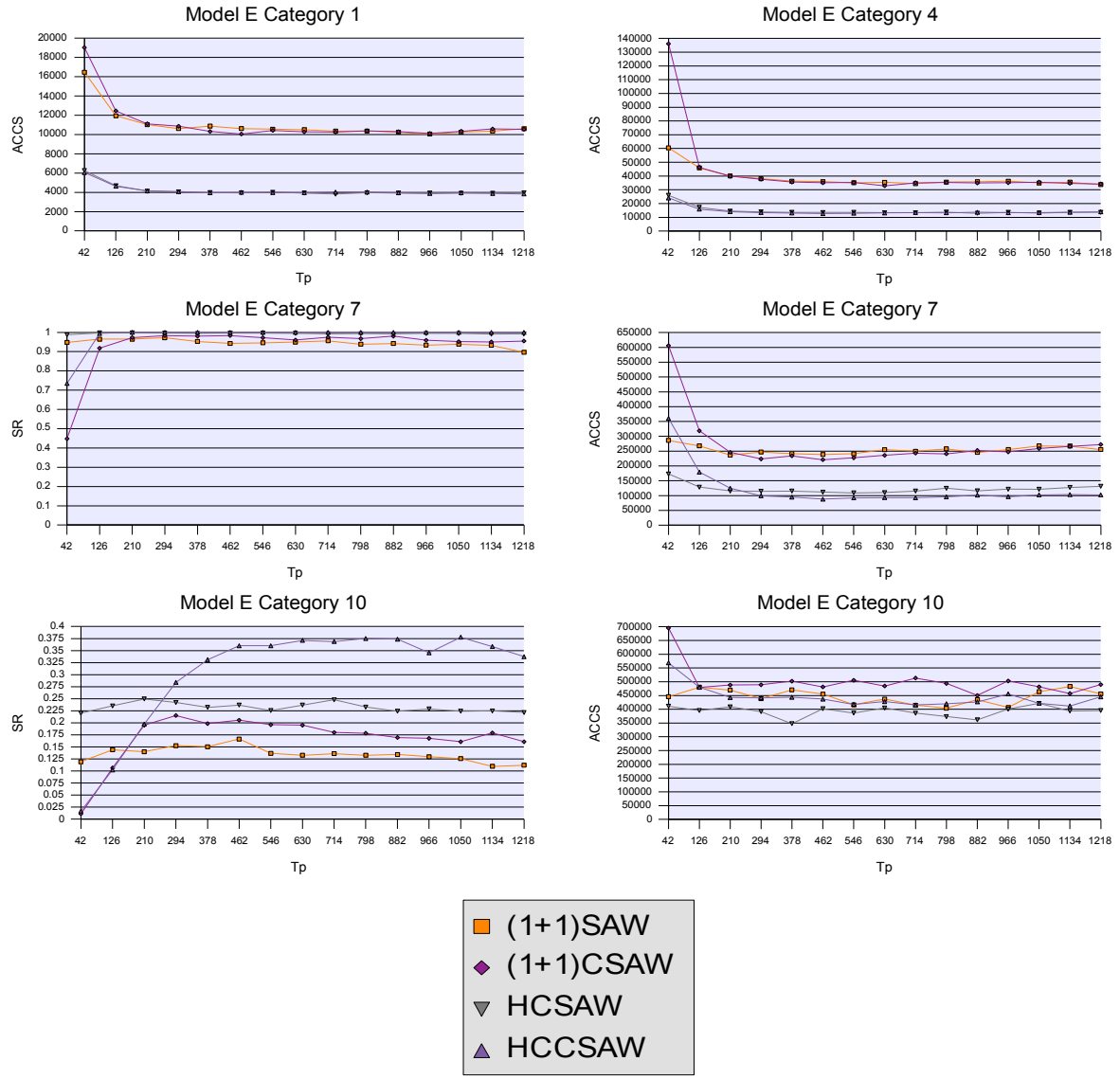

Fig. 5.3: The influence of T_p over categories E1, E4, E7 and E10, for which $n = 15$ and $m = 15$. The values of T_p are problem size dependent and range from 0.2n(m-1) to 5.8n(m-1). maxCC is set to 1,000,000.

Figure 5.3 shows the results of running the four suggested algorithms with four model E categories, and $maxCC = 1,000,000$. The categories are E1, E4, E7, and E10. Recall that model E categories contain problems of varying hardness. E1 holds the easiest CSPs, and E10 is at the phase transition. The success rate graphs for categories E1 and E4 are omitted because for them all the algorithms have a 100% success rate for all T_p values. The results show again that the conflict based SAW variants are more sensitive for lower

values of T_p. Category E10 shows that HCCSAW might require a T_p value of $2C$ in order to reach full effectiveness. Otherwise even lower T_p values are still applicable to all algorithms.

In conclusion, 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 constraint based SAW and conflict based SAW. The value $T_p = 1.4C$, shown on the forth column from the left of all the graphs, 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, especially for the HC variants which appear to be more stable. CSAW variants outperform SAW variants at all tests performed for model F and model RB, which indicates that conflict based SAW consistently outperforms constraint based SAW on phase transition CSPs. Finally, results for the RB model show that the traditional value $T_p = 250$ may be too low 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 rSAWEA.

5.3 Comparisons for CSPs of Varying Hardness

We turn to test the implications of using conflict based SAW for soluble CSPs of varying hardness by using the model E test-set.

Tab. 5.1: Model E test-set results for $maxCC = 1,000,000$.

Category	**(1+1)SAW**		**(1+1)CSAW**		**HCSAW**		**HCCSAW**	
	SR	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**
E1	1	10801	1	10702	1	4073	1	4002
E2	1	15382	1	15605	1	5785	1	5923
E3	1	23192	1	24142	1	8816	1	8840
E4	1	37195	1	37819	1	14157	1	13354
E5	1	66797	1	61639	1	24898	1	23784
E6	1	127283	1	120672	1	48065	1	45457
E7	0.96	241927	0.98	239314	0.99	115867	1	97525
E8	0.72	352572	0.80	370575	0.89	239675	0.96	246007
E9	0.33	419616	0.44	433448	0.52	345121	0.58	374262
E10	0.15	479161	0.21	486220	0.26	360991	0.28	441589

Table 5.1 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 variants of both algorithms. However, as we get closer to the phase transition, conflict based SAW begins to outperform the constraint based SAW in terms of SR, which is the primary measure.

When the algorithms receive more time the effect of conflict based SAW becomes more apparent.

Tab. 5.2: Model E test-set results for $maxCC = 10,000,000$.

Category	(1+1)SAW		(1+1)CSAW		HCSAW		HCCSAW	
	SR	ACCS	SR	ACCS	SR	ACCS	SR	ACCS
E6	1	127283	1	120672	1	48065	1	45457
E7	1	297102	1	249181	1	118642	1	100289
E8	0.98	937903	1	605367	0.99	465234	1	292644
E9	0.72	2.09e+06	0.94	1.90e+06	0.72	2.09e+06	0.98	1.29e+06
E10	0.39	2.64e+06	0.75	3.20e+06	0.49	1.93e+06	0.90	2.74e+06

Table 5.2 shows results for the last five categories 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 conflict based SAW widen the performance gap, yet in a more modest way.

5.4 A Comparison with rSAWEA

Table 5.3 shows the results of running HCSAW, HCCSAW and rSAWEA on the model F test-set. The results for rSAWEA were originally published in [3], and taken from [5, 6]. The time allowed for HCSAW and HCCSAW to complete their runs was 1 million conflict checks. rSAWEA was allowed 100,000 evaluations of search points.

Tab. 5.3: Model F test-set results.

Category	HCSAW		HCCSAW		rSAWEA	
	SR	ACCS	SR	ACCS	SR	ACCS
F1	0.74	376	1	722	1	9665
F2	0.80	8317	1	3160	0.98	350789
F3	0.87	31421	1	7861	0.95	763903
F4	0.90	52897	1	15847	0.97	652045
F5	0.91	53486	1	18337	1	557026
F6	0.91	70938	1	24980	1	715122
F7	0.88	92896	1	49671	1	864249
F8	0.86	102332	1	47927	1	1012082
F9	0.96	70457	1	37387	1	408016

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 [2]. Luckily, the problem does not occur when HCCSAW is used, and performance in terms of ACCS is further improved. A comparison of HCCSAW with rSAWEA shows that HCCSAW is over ten times, and sometimes 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.

Chapter 6

Conclusions

In this thesis 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 turned out to be over ten times faster than the best SAWing EA for CSPs. This means that weight adaptation, when used correctly, is more powerful than previously 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. The results in this thesis were submitted to the IEEE-CEC conference of 2009.

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 to 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.

Further research may include a comparison of HCCSAW with better EAs for CSPs in order to determine if conflict based SAW provides a superior EA for CSPs. Since the conflict based SAW algorithms presented here are simple, better conflict based SAW algorithms for CSPs may be sought. The contribution of randomness is unclear so a non EA approach might prove useful. Local search algorithms that use weight adaptation should be compared with efficient non local search algorithms for CSPs. Since weight adaptation was found here to be more powerful than usually perceived, some form of a weight adaptation heuristic might also prove useful for non local search algorithms.

Bibliography

[1] 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*, pages 107–120, 1997.

[2] B. Craenen and A. Eiben. An experimental comparison of SAWing EAs for a new class of random binary CSPs. In *Proceedings of 2002 Congress on Evolutionary Computation*, pages 878–883. IEEE Computer Society, 2002.

[3] B. G. W. Craenen. *Solving Constraint Satisfation Problems with Evolutionry Algorithms*. PhD thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, 2005.

[4] B. G. W. Craenen and A. E. Eiben. Stepwise adaption of weights with refinement and decay on constraint satisfaction problems. In 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, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 291–298, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.

[5] B. G. W. Craenen and B. Paechter. A tabu search evolutionary algorithm for solving constraint satisfaction problems. In T. P. Runarsson, H.-G. Beyer, E. K. Burke, J. J. M. Guervós, L. D. Whitley, and X. Yao, editors, *PPSN*, volume 4193 of *Lecture Notes in Computer Science*, pages 152–161. Springer, 2006.

[6] B. G. W. Craenen and B. Paechter. A conflict tabu search evolutionary algorithm for solving constraint satisfaction problems. In J. I. van Hemert and C. Cotta, editors, *EvoCOP*, volume 4972 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2008.

[7] A. E. Eiben and Z. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In *International Conference on Evolutionary Computation*, pages 258–261, 1996.

[8] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[9] 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.

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

[11] J. Frank. Weighting for godot: Learning heuristics for GSAT. In *AAAI/IAAI, Vol. 1*, pages 338–343, 1996.

[12] J. Frank. Learning short-term weights for GSAT. In *IJCAI (1)*, pages 384–391, 1997.

[13] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[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*, pages 621–630, London, UK, 2000. Springer-Verlag.

[15] E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. Random constraint satisfaction: Theory meets practice. In M. J. Maher and J.-F. Puget, editors, *CP*, volume 1520 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 1998.

[16] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[17] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[18] F. Rossi, C. J. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *ECAI*, pages 550–556, 1990.

[19] B. Selman and H. Kautz. Domain-independent extensions to GSAT: solving large structured satisfiability problems. In *Proceedings of theInternational Joint Conference on Artificial Intelligence(IJCAI-93)*, Chambry, France, 1993.

[20] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, pages 440–446, 1992.

[21] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In D. J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. San Francisco, CA: Morgan Kaufmann, 1989.

[22] K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 12:93–103, 2000.