

SAT
A Plugin for HeuristicLab

Florian Lonsing

Bachelor Project

*Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria*

March - July 2005

Contents

1	Introduction	3
1.1	SAT – The Satisfiability Problem	4
1.2	SAT & HeuristicLab	5
2	Implementation Details	10
2.1	Problem Representation	10
2.1.1	General Parameters	10
2.1.2	Mutation Parameters	10
2.1.3	Weight Parameters	12
2.2	Solution Representation	12
2.2.1	Properties	12
2.2.2	Methods	13
2.3	Instance Input	23
2.4	Assignment Input	25
2.5	Variable Scores	26
2.6	Evaluation Operators	28
2.6.1	Standard	28
2.6.2	ClauseWeights	29
2.7	Mutation Operators	29
2.7.1	NVarFlip	30
2.7.2	CDRW	31
2.7.3	UCDRW	31
2.7.4	NoveltyPlus	31
2.7.5	WalkSAT-SKC	32
2.7.6	RandomRestart	32
2.7.7	GSAT	33
2.8	Recombination Operators	33
2.8.1	UC	34
2.8.2	CC	35
2.8.3	CCTM-SBS and CCTM-SBS-NEW	35
2.8.4	CCTM-CBC and CCTM-CBC-NEW	35

2.8.5	FF	36
2.8.6	GASAT	36
2.9	Weight Update Operators	37
2.9.1	SAPS	37
2.9.2	WIUSC	37
2.10	Neighbourhood Operators	38
2.10.1	Simulated Annealing	38
2.10.2	Tabu Search	38
2.11	Memory Operators	39
2.11.1	Standard	39

Overview

This report presents an implementation of a plugin modelling the *Satisfiability Problem (SAT)* in HeuristicLab [10]. In the first chapter, a short introduction into SAT, the basic concepts of the HeuristicLab environment and the features of the implementation are presented. The second chapter focuses on details of the implementation: problem and solution representation of SAT in HeuristicLab, instance creation, evaluation, mutation, recombination and especially methods, which are used for efficiently evaluating solutions. Additionally to the operators for the algorithms provided in HeuristicLab, their background, if necessary for understanding, is described.

Chapter 1

Introduction

HeuristicLab [10] is a problem- and algorithm-independent heuristic optimization framework. Algorithms and problems are separated from each other such that they can freely be exchanged. Each problem implementation provides special operators, which can be used together with certain algorithms, enabling the user to optimize problems with different algorithms without having to switch to another framework or problem-dependent algorithm. A broad variety of algorithms have been implemented for HeuristicLab, ranging from pure Random Search to population-based approaches such as Genetic Algorithms and some more sophisticated variants, which are worth testing with SAT:

- Sexual Genetic Algorithm: two different selection operators may be specified for parent selection during the recombination phase.
- Island Genetic Algorithm: the population consists of subpopulations (islands) which evolve independently from each other. Individuals migrate from one population to another and thus introduce new genetic information in the populations.
- Generic Genetic Algorithm: the replacement strategy may be set to *elitism*, where the best n individuals are taken into the next population without any modification, or *random*, where either the whole population or only a specified proportion is replaced for the next generation.

Among the Problem-plugins, there is the Travelling Salesman Problem (TSP), n-Dimensional, Real-Valued Test Functions (TFND), Job Shop Scheduling Problem (JSSP), Multi Processor Scheduling Problem (MPSP) and some more.

In this report, the concepts of HeuristicLab are rather briefly touched on, the focus lies on technical details concerning the implementation of the SAT Problem-plugin. Further, more detailed information about the framework, its concepts and

motivation may be helpful for understanding and can be found in technical reports of the authors of HeuristicLab and on the website of the project [10].

1.1 SAT – The Satisfiability Problem

The *Satisfiability Problem (SAT)* is one of the most popular and thoroughly studied combinatorial problems and has, despite of its conceptual simplicity, many practically relevant applications such as verification in hardware [11] and software [12] and SAT encodings of other combinatorial problems e.g. from the domain of planning, logistics or scheduling.

A SAT instance is a formula in propositional logic. In propositional logic, propositional variables are of type Boolean and can be assigned either the value *true* or *false*. A variable occurring either in positive or negative form is called a *literal*. Literals are combined using logical operators such as conjunctions \wedge , the logical “*and*” and disjunctions \vee , the logical “*or*” to form propositional formulae.

When studying instances of SAT, e.g. in context with SAT solver evaluation and development, then these instances are restricted to a special format most of the time, the *conjunctive normal form (CNF)*. A formula F in CNF consists of disjunctions over literals, which are called *clauses*, the clauses itself are combined using conjunctions. A SAT instance in CNF is the most common form of representation. Any propositional formula can be transformed into a logically equivalent formula in CNF.

For a given propositional formula F , the *Satisfiability Problem* in propositional logic is now to determine if this formula is satisfiable or not, that is, whether an assignment of truth values to the variables appearing in the given formula can be found such that the whole formula evaluates to *true*. If this is the case, we call such an assignment a *model* of the formula which itself is then called *satisfiable*. Of course there may be more than one model for a formula. In the other case, if the formula does not have a model, the formula is called *unsatisfiable*.

The advantage of representing a SAT instance as a formula in CNF becomes obvious: a formula in CNF is satisfiable if, and only if, all clauses are satisfiable, since the clauses are combined by conjunctions. A clause can be satisfied by assigning values to the variables occurring in that clause such that at least one literal within the clause evaluates to *true*. Because literals within a clause are combined by disjunctions, one such literal will satisfy the clause.

From one point of view, SAT can be seen as a decision problem, where the goal is to decide whether a formula is satisfiable or not: only “yes” or “no” as an answer is required. This is the decision variant of SAT. From another point of view — the search variant or model-finding variant — the goal is to find a satisfying assignment for the formula. It is obvious that solving the model-finding variant

also solves the decision variant, but only when a model is found. If it is not assured by an algorithm that a formula – in case that this particular algorithm did not find a model – really does *not* have a model, then one can not be sure that there does not exist a model at all. Such algorithms are called *incomplete*, as all algorithms provided in HeuristicLab are.

The performance of complete – a decision is made whether an instance is satisfiable or not during each run – and incomplete solvers is dependent on the instance class, i.e. whether the instance is structured or generated at random. State-of-the-art SAT solvers are annually evaluated in a competition [13], where both complete and incomplete solvers are tested on instances, which stem from different areas of application such as SAT encodings of other combinatorial problems (planning, logistics), Bounded Model Checking [11] or instances which have been generated at random.

1.2 SAT & HeuristicLab

This section provides a brief overview over the functionality and the features that come with the SAT Problem-plugin in HeuristicLab. Further information on different aspects such as representing SAT instances and solutions, instance input, evaluating solutions, operators and – in general – implementation details can be found in chapter 2.

SAT is a Problem-plugin for HeuristicLab and therefore not only has to store specific problem parameters such as the number of variables and clauses in the SAT instance, but also to provide methods for generating and evaluating new solutions, i.e. `SATSolution` objects. A `SATSolution` object represents a concrete solution, i.e. a variable assignment, or – more generally spoken – a candidate solution, if the solution represented is not a model of the instance, and it provides an operator for evaluating solutions. The quality of a solution is calculated by means of this evaluation operator which basically counts the number of currently satisfied clauses (and perhaps taking into account clause weights, if specified). To be precise, the evaluation function is not always calculated from scratch each time a solution has been modified, rather this is done incrementally whenever possible.

An arbitrary algorithm in the HeuristicLab environment makes use of operators for modifying a solution or creating new solutions during a run. These operators have to be implemented in the respective Problem-plugin the algorithm is working on. There are crossover and mutation operators for Genetic Algorithms and Evolutionary Strategies, and neighbourhood operators for Simulated Annealing and Tabu Search.

The user may create a new SAT Problem instance via the SAT Problem Form. Figure 1.1 shows a typical screenshot. The instance can either be typed in manually

or loaded from a file which contains a SAT instance in DIMACS format [2]. It can be specified, if the instance is satisfiable, unsatisfiable or if satisfiability is unknown. For the latter two cases, the property `BestKnownSolutionQuality` is set to the value -1 and will be updated whenever an algorithm finds solution that is better than the current value. After the instance has been parsed, statistical information about the instance is calculated:

- the number of variables
- the number of clauses
- the ratio *clauses/variables*: this is an interesting piece information in context with *Uniform Random k-SAT* instances. Such instances consist of m clauses of length k . Each clause is constructed by adding k literals chosen uniformly at random from the set of all $2 \cdot n$ possible literals, where n is the number of variables. At a ratio of approximately 4.26, a so-called solubility phase transition can be observed: generated instances with ratios beyond that threshold are likely to be unsatisfiable.
- the number of literals
- the ratio *literals/clause*

Another way of creating an instance is to use the `SATLibImporter`. In general, instances of problems that implement an importer can be created from a file containing the problem instance and its parameters in a custom, textual representation. The advantage of this approach is, that parameters may be explicitly set in the input file together with the instance in textual format. The input file format of the SAT Problem Form and `SATLibImporter` are different with respect that the Problem Form does not accept parameters to be specified directly in the input file. Further information on instance input can be found in section 2.3.

A SAT Problem object has several parameters, some of which are dependent on specific algorithms or operators and which take influence on the performance of these. The parameters can either be set manually (figure 1.2) on the problem form or imported in the input file by means of the `SATLibImporter`.

The HeuristicLab environment provides three basic ways of visualizing the progress of the search:

1. First, a record of improvements made can be kept in form of a textual log (figure 1.5), which is updated whenever a better solution is found. A log entry in case of the SAT plugin displays the current variable assignment and a list of unsatisfied clauses under this assignment. The assignment, for example, may be copied to a textfile and used in the `SATLibImporter` as described in section 2.4.

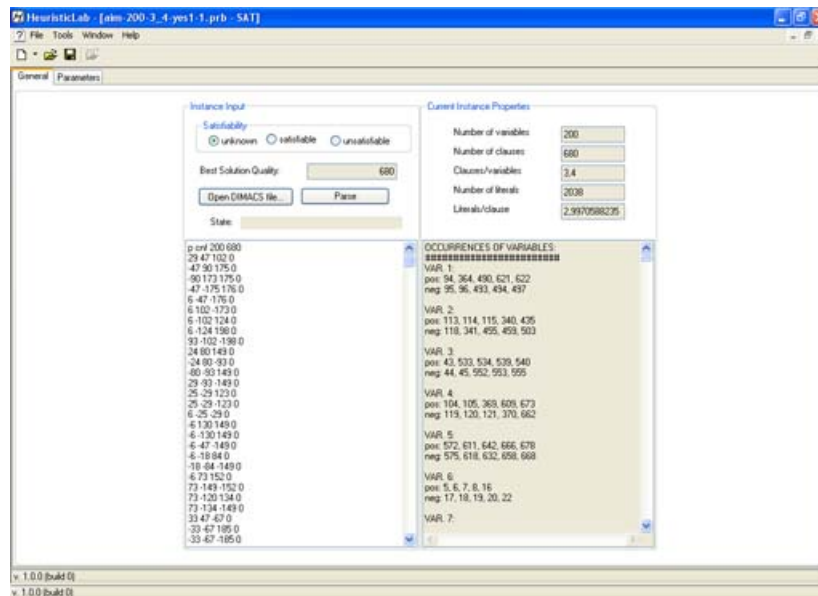


Figure 1.1: The SAT Problem Form provides information about the given SAT instance.

2. Secondly, each problem can implement a custom solution display (figure 1.3). The SAT plugin simply juxtaposes the numbers of clauses currently satisfied and unsatisfied in a piechart. When using clause weights, the number of currently satisfied clauses can *not* necessarily be derived from the evaluation function value of a solution, as it is the case if the use of weights is switched off. The reason is, that the weights of unsatisfied clauses are subtracted from the evaluation function value. However, the piechart always displays the number of currently satisfied clauses, no matter if clause weights are enabled or not.
3. Finally, every algorithm plots a quality chart (figure 1.4) displaying the quality of the best solution that has been found so far and additional information such as the average quality of the population or of subpopulations and the effect of migration phases e.g. in an Island Genetic Algorithm.

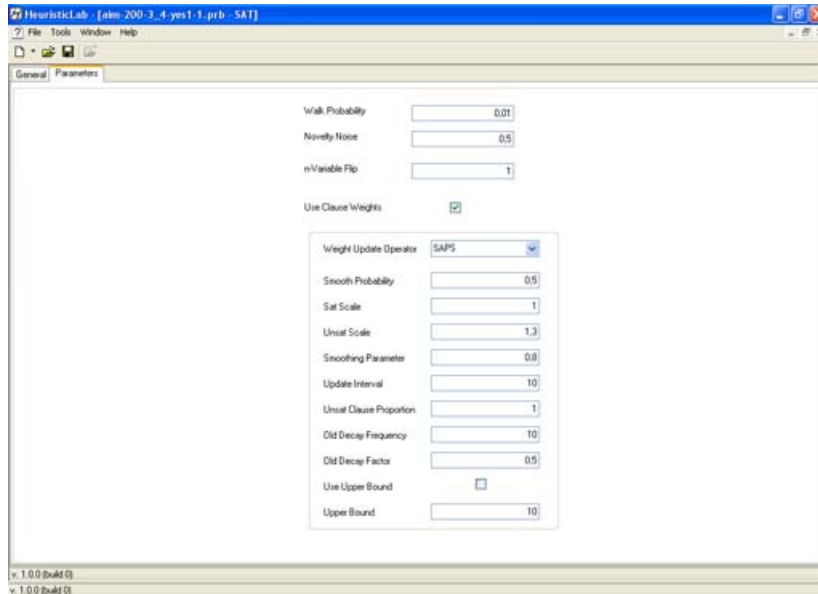


Figure 1.2: Parameters of a SAT Problem object may be set on the Problem Form.

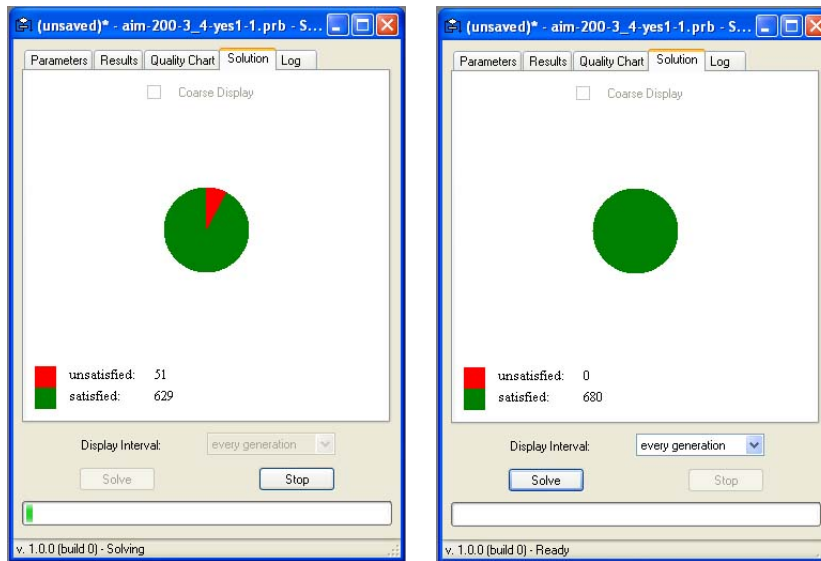


Figure 1.3: The Solution Display of the SAT plugin: a simple piechart juxtaposing the number of satisfied and unsatisfied clauses; working (left) and after a solution has been found (right).

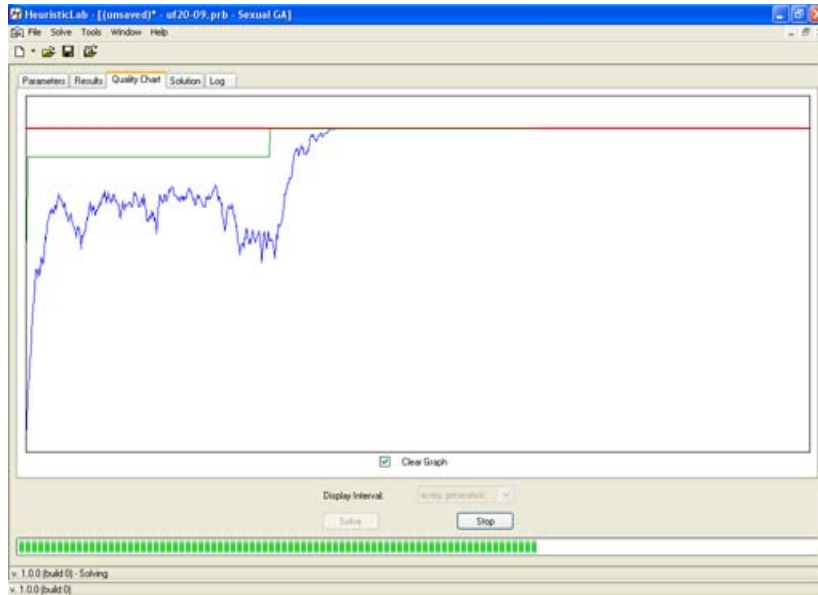


Figure 1.4: The Quality Chart of a Sexual Genetic Algorithm.

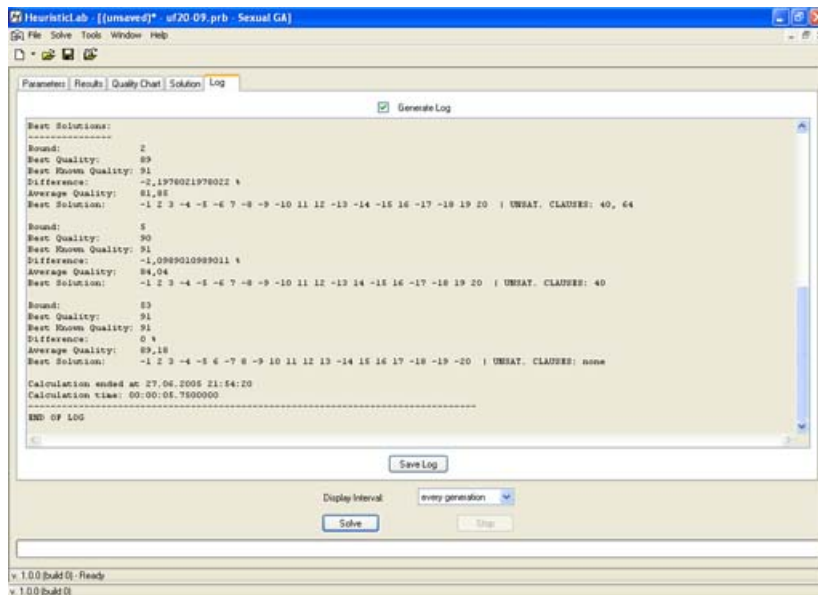


Figure 1.5: The Log of a SAT instance in HeuristicLab.

Chapter 2

Implementation Details

2.1 Problem Representation

A `Problem` object in the framework of `HeuristicLab` stores specific information about the given problem instance and its parameters set by the user. The numerous parameters of a SAT `Problem` object are listed in figure 2.1 to provide a quick overview.

2.1.1 General Parameters

These properties store instance-specific information such as the number of variables, clauses and literals in the CNF formula and the clauses itself. The indices of clauses where a variable x_i occurs positively (x_i) or negatively ($\neg x_i$) are stored in `posOccurrences[i]` or `negOccurrences[i]`. These lists are built when parsing the SAT instance and play an important role in evaluating a solution.

2.1.2 Mutation Parameters

A mutation operator modifies a small part of a solution which can either be carried out directed, i.e. explicitly trying to improve the evaluation function value, or undirected by introducing random modifications such as flipping one or more (`nVarFlip`) variables. Directed mutation operators used in the SAT `Problem`-plugin such as `NoveltyPlus` or `WalkSAT-SKC` may select a variable to be flipped at random with probability `walkProbability`, `NoveltyPlus` additionally makes use of the noise setting in order to do an age-biased variable selection. Details on mutation operators can be found in section 2.7.

<i>Property</i>	<i>Description</i>
General Parameters	
varCount	number of variables in the CNF formula
clauseCount	number of clauses in the CNF formula
litCount	number of literals in the CNF formula
posOccurrences	lists storing positive occurrences of variables
negOccurrences	lists storing negative occurrences of variables
clauses	the clauses in the formula
Mutation Parameters	
walkProbability	probability of random walk step during mutation
noveltyNoise	noise parameter for NoveltyPlus
nVarFlip	number of variables to be flipped during mutation
Weight Parameters	
useWeights	use clause weights?
WeightUpdateOperator	performs clause weight modification
WeightUpdateFrequency	interval of weight updates
oldDecayFrequency	interval of weight decay
oldDecayFactor	factor used for decaying weights
unsatClauseProportion	see text
useUpperBound	use an upper bound on clause weights?
upperBound	upper bound on clause weights
smoothProbability	SAPS: probability for smoothing weights
smoothParameter	SAPS: parameter used for smoothing weights
satScale	SAPS: factor for scaling satisfied clauses
unsatScale	SAPS: factor for scaling unsatisfied clauses

Figure 2.1: Properties of a SAT Problem object.

2.1.3 Weight Parameters

When associating weights with each individual clause to influence the progress of the search, there are many possibilities of updating the weights over time. The weight parameters are part of weight updating policies which have been implemented in the SAT plugin. In general, the use of clause weights can be switched on or off, weights can be decayed before increasing by multiplying them with a factor `oldDecayFactor` after a certain number of generations have been generated or rounds performed (`oldDecayFrequency`), and a weight update is done every `weightUpdateFrequency` generations (in a population-based algorithm) or rounds according to the updating policy specified in `weightUpdateOperator`. Typically, the weights of unsatisfied clauses are increased. In a population-based algorithm such as a Genetic Algorithm, it has to be decided some way if a particular clause is satisfied or unsatisfied, since the satisfaction status can be different from individual to individual. This decision can be influenced by specifying a proportion `unsatClauseProportion` $\in [0, 1]$ of the individuals in a population, where a clause is considered to be unsatisfied, if it is satisfied in less than `#Individuals` \cdot `unsatClauseProportion` individuals. Clause weights can be bounded, which prevents them from taking values above the specified bound. However, limiting the range of clause weights during a run of an algorithm may also be achieved by decaying, by setting `unsatClauseProportion` to a smaller value or by larger update intervals.

Probabilistically smoothing clause weights towards the average clause weight is done in the SAPS-inspired weight update operator (Scaling and Probabilistic Smoothing). SAPS is a Dynamic Local Search Algorithm for SAT presented in [8]. In the operator, the weight of a clause is scaled depending on its satisfaction status by the factor `unsatScale` or `satScale`. The implemented weight update operators are described in section 2.9.

2.2 Solution Representation

2.2.1 Properties

An algorithm in HeuristicLab works on so-called `Solution` objects or populations of such objects. Depending on the problem that a `Solution` object belongs to, it has different properties and methods which are closely related to characteristics of the problem. In case of SAT, the properties of a `SATSolution` object are given in figure 2.2.

The two core properties of a `SATSolution` object are `varAssignment` and `curClauseSatesInt`. The property `varAssignment` represents the current variable assignment of the `SATSolution` object as an array of boolean values numbered

<i>Property</i>	<i>Description</i>
<code>varAssignment</code>	the current variable assignment
<code>curClauseStatesInt</code>	the satisfaction status of the clauses
<code>varScores</code>	the current score of the variables
<code>unsatList</code>	a list of currently unsatisfied clauses
<code>score</code>	the evaluation function value (sat. clauses, weights)
<code>ages</code>	the age list of the variables

Figure 2.2: Properties of a `SATSolution` object

from 0 to `varCount` – 1 and `curClauseStatesInt` is an integer array with index range from 0 to `clauseCount` – 1 which stores the value 0 at position i if clause c_i is currently unsatisfied, or the value 1 if the clause is *critically* satisfied, that is, if it is satisfied by exactly one literal, and finally a value which is greater than 1 if the clause is satisfied. The indices of all clauses currently unsatisfied are kept in a dynamically changing list in the property `unsatList` for efficiently accessing them. The score of each variable (see section 2.5) in a solution is dynamically updated after each variable flip and stored in the floating point number array `varScores`, analogously the age of a variable in `ages` which is defined as the number of variable flips that have been performed on a solution since that variable has been flipped last. After variable x_i has been flipped, its age is reset to zero and the age of all other variables is increased by 1. The evaluation function value of a solution is given by `score`. Because this property stores the number of currently satisfied clauses (if clause weights are disabled), SAT can be seen as a maximization problem in the implementation for HeuristicLab.

There is one static property of the `SATSolution` class which is not listed in the table: `clauseWeights`. It is an array of double precision floating point numbers with length `clauseCount`, which stores the current (global) weights for each clause.

2.2.2 Methods

Figure 2.3 shows important methods of a `SATSolution` object. They realize the continuous updating of the variable scores and age, the satisfaction status of the clauses and the evaluation function value after each variable flip and weight modification.

FlipVar(int flipIndex)

The method `FlipVar(int flipIndex)` performs a variable flip on the specified variable. It simply inverts the current value of the given variable and then calls the

<i>Methods</i>	<i>Description</i>
<code>updateAges(int flipIndex)</code>	managing the age of variables
<code>InitializeEvalValues</code>	evaluation and variable scores
<code>UpdateClauseStates(int flipIndex)</code>	partial eval. and var. scores
<code>FlipVar(int flipIndex)</code>	performs variable flip
<code>UpdateWeightedEvalValues(double[] deviations)</code>	updating after weight modification

Figure 2.3: Methods of a `SATSolution` object

methods `UpdateClauseStates(int flipIndex)` and `UpdateAges(int flipIndex)`.

UpdateAges(int flipIndex)

`UpdateAges(int flipIndex)` performs the age updating of the variables. The age of the variable that has been flipped is reset to zero and the age of all other variables is increased by one. For example, the mutation operator `NoveltyPlus` makes use of the stored age when selecting the variable to be flipped.

Two of the listed methods – `InitializeEvalValues` and `UpdateClauseStates(int flipIndex)` – can in some way be considered as the “heart” of the plugin implementation because they perform the initialization and continuous updating of the evaluation function value and the variable scores of a `SATSolution` object after each variable flip. Runtime aspects have been taken into account when developing these methods and they shall therefore be explained in detail.

InitializeEvalValues

InitializeEvalValues is needed whenever a new `SATSolution` object has been created or, in general, whenever one intends to evaluate a `SATSolution` object from scratch. The method performs the following actions:

- traverse the list of clauses
- check all literals in each clause c_i
- for each literal in clause c_i , determine if this literal has the value *true* under the current variable assignment
- if so, increase the satisfaction status of clause c_i by one
- update the evaluation function value and variable scores according to the satisfaction status of each clause

```

InitializeEvalValues() {
    reset score, curClauseStatesInt, varScores, unsatList;

    FOR all clauses c {
        int satVar; // single pos. var. in clause if clause is crit. sat.
        FOR all literals lit in c {
            IF (lit is negative) {
                IF (varAssignment[var(lit)] = false) {
                    curClauseStatesInt[index(c)]++;
                    satVar = var(c);
                }
            }
            ELSE {
                IF (varAssignment[var(lit)] = true) {
                    curClauseStatesInt[index(c)]++;
                    satVar = var(c);
                }
            }
        } // END FOR all literals in c

        // check satisfaction status of clause c
        IF (curClauseStatesInt[index(c)] == 0) { // clause is unsat.
            // increase score of all occuring variables in clause c
            IF (useWeights)
                score = score - clauseWeights[index(c)];
            unsatList.Add(index(c));
            FOR all literals in c {
                varScores[var(lit)]++;
                IF (useWeights)
                    varScores[var(lit)] = varScores[var(lit)] + clauseWeights[index(c)];
            } // END FOR all literals in c
        } ELSE IF (curClauseStatesInt[index(c)] == 1) { // clause is critically sat.
            // decrease score of satisfying variable
            score++;
            varScores[satVar]--;
            IF (UseWeights)
                varScores[satVar] = varScores[satVar] - clauseWeights[index(c)];
        } ELSE { // clause is sat.
            // scores of variables do not change
            score++;
        }

    } // END FOR all clauses

} // END InitializeEvalValues

```

Figure 2.4: Pseudo code of the method `InitializeEvalValues` in a `SATSolution` object

The pseudo code of this method can be found in figure 2.4. Some annotations to the given pseudocode: `UseWeights` is the boolean flag indicating whether clause weights should be used or not during the update of the evaluation function value and the variable scores. Weighted updating will be described later on in this section. Details on the idea and background of the usage of clause weights can be found in section 2.6.2. The problem representation is described in section 2.1. In the pseudocode, two functions are used for the sake of simplicity: `var(lit)` and `index(c)`. The first one returns the variable index for a given literal, the second one the index of the given clause.

At the beginning of `InitializeEvalValues`, the properties `score`, `curClauseStates`, `varScores` and `unsatList` are reset to their respective initial values (zero, null etc.). The whole information, which is held by these properties, will be built up from scratch in this method.

For all clauses c in the problem instance, all literals lit in c are checked right in the order they occur in the clause. If lit is a positive literal, then the current value of the variable of lit is checked. If this value is *true*, the counter of clause c in `curClauseStatesInt` can be increased, meaning that one literal has been found which satisfies clause c . In case that the counter of a clause has value 1, that clause is called *critically satisfied* by the particular variable whose literal is *true* in that clause. Negative literals are treated analogously: now it is checked, if the respective variable has the value *false*.

After all literals of clause c have been traversed, a decision can be made – by inspecting the counter value for clause c – whether the clause is satisfied, critically satisfied or unsatisfied.

If the clause is *unsatisfied*, the index of the clause is added to the list of unsatisfied clauses. The property `score` is not decreased – it has been initially reset to zero – except if clause weights are used. In this case, the weight of the unsatisfied clause is subtracted from `score`. Next, the scores of all variables which occur in the clause are increased, because flipping one of these variables will satisfy the clause. Further details on variable scores can be found in section 2.5. Again, if clause weights are used, the respective weight of the clause is added to the variable scores.

If the clause is *critically satisfied*, `score` is increased by one. Clause weights have an impact on the evaluation function value only, if clauses are unsatisfied. A satisfied clause always contributes positively to the evaluation function value by one, therefore no weights are added when a satisfied clause is encountered. The only variable, whose score has to be updated, is the one variable which critically satisfies the clause. Its score has to be decreased, since flipping that variable will immediately break the critically satisfied clause.

Finally, if the clause is *satisfied*, the only action to be done is to increase **score** by one. The variable scores do not have to be updated, because there is more than one variable satisfying the clause and flipping one of these will not cause the clause to become unsatisfied.

The runtime complexity of the method `InitializeEvalValues` is $O(m \cdot CL(n))$, where m is the number of clauses in the problem instance, $CL(n)$ an upper bound on number of literals per clause, and n the number of variables in the instance. One must be aware, that the performance will be the weaker, the larger the population size is, e.g. in a Genetic Algorithm. Especially when used on large problem instances, the initialization phase may cause a noticeable pause right after the algorithm has started.

UpdateClauseStates(int flipIndex)

The second core method of the plugin implementation is `UpdateClauseStates(int flipIndex)`. By means of this method, the evaluation function value and the scores of the variables are updated whenever a variable has been flipped. The key idea behind the process of updating these values is to consider only those clauses which are dependent on the variable that has been flipped.

The implementation of the method with respect to handling the scores of the variables and the evaluation function value is based on the description in [1].

Before starting with a detailed explanation of how the method `UpdateClauseStates(int flipIndex)` works, the following terminology about variable and clause dependencies has to be introduced as a basis for the understanding of the description:

- In a CNF formula F , two variables x and x' are *dependent* on each other if, and only if, there is a clause which contains both x and x' . For a variable x , the set $V_{dep}(F, x)$ denotes the set of all variables which depend on x .
- A clause c is *dependent* on variable x if, and only if, it contains x . Analogously, the set $C_{dep}(F, x)$ denotes the set of clauses which depend on x .
- A clause c is *critically satisfied* by a variable x under the current variable assignment if, and only if, x appears in c , c is currently satisfied and flipping the variable x would make clause c become unsatisfied. Note that this is exactly the case when `curClauseStatesInt[index(c)] = 1` as described above.
- A variable x' is *critically dependent* on a variable x if, and only if, there is a clause c that contains both x and x' , and flipping the variable x makes clause c change its satisfaction status from critically satisfied to unsatisfied

(or vice versa), or from satisfied to critically satisfied by x' (or vice versa). Note that the first case corresponds to a change in the counter value in `curClauseStatesInt[index(c)]` from 1 to 0 or vice versa, and the second case to a change from 2 to 1 or vice versa.

When flipping the value of a variable x , the satisfaction status of all clauses dependent on x , i.e. those which contain x , might change. For example, if the value of x before the flip was *true*, then the flip might potentially have broken clauses where x occurs positively (x) and satisfied clauses where x occurs negatively ($\neg x$). All other, non-dependent clauses, of course, do not have to be taken into account because they can not have been affected by the flip. Thus, updating the evaluation function value can be restricted to clauses from set $C_{dep}(F, x)$.

For updating the variable scores, only the variables that are *critically* dependent on the flipped variable have to be considered, i.e. those variables that occur in the same clauses as the flipped variable and where the clauses change their satisfaction status from satisfied to unsatisfied (or vice versa) or from critically satisfied to satisfied (or vice versa).

The pseudo code of this method for handling a variable flip from *false* to *true* can be found in figures 2.5 and 2.6. For handling a flip from *true* to *false*, only the roles of the two lists `posOccurrences` and `negOccurrences` have to be switched, the actions performed are the same.

The method `UpdateClauseStates(int flipIndex)` gets as an argument the index of the flipped variable. It is assumed that the value of the variable in `varAssignment[flipIndex]` has already been flipped and that the two sets `posOccurrences[flipIndex]` and `negOccurrences[flipIndex]` are disjoint. The latter assumption is in fact crucial for the correctness of the update process. See section 2.3 for details on instance input and preprocessing techniques performed which guarantee to keep this important assumption.

At the beginning, the score of the flipped variable can be inverted. This is a consequence of the definition of a variable's score. It simply changes its sign and does not have to be taken into account during the rest of the method. Details on the score of a variable can be found in section 2.5. Next, a decision is made whether the value of the variable with index `flipIndex` has been flipped from *true* to *false* or vice versa. This can be done by checking the current value of `varAssignment[flipindex]`, which already stores the new value after the flip.

Assume, the value has been flipped from *false* to *true*, as it is described in figures 2.5 and 2.6. Now, all clauses dependent on the flipped variable can be checked.

First, all clauses c , where the flipped variable occurs negatively, are checked (figure 2.5). The satisfaction status of such clauses might have changed from satisfied to critically satisfied or from critically satisfied to unsatisfied by the flip.

```

UpdateClauseStates(int flipIndex) {
    // invert score of flipped variable
    varScores[flipIndex] = varScores[flipIndex] * (-1);

    IF (varAssignment[flipIndex] == true) { // flip from false to true

        FOR all clauses c in negOccurrences[flipIndex] {
            curClauseStatesInt[index(c)]--;
            IF (curClauseStatesInt[index(c)] == 0) { // clause now unsat
                score--;
                IF (UseWeights)
                    score = score - clauseWeights[index(c)];
                unsatList.Add(index(c));
                FOR all literals lit in clause c { // check literals in clause
                    IF (var(lit) != flipIndex) {
                        varScores[var(lit)]++;
                        IF (UseWeights)
                            varScores[var(lit)] += clauseWeights[index(c)];
                    }
                } // END FOR all literals in clause c
            } ELSE IF (curClauseStatesInt[index(c)] == 1) { // clause now crit. sat
                int satVarIndex = index of var. that crit. sat. clause c
                varScores[satVarIndex]--;
                IF (UseWeights)
                    varScores[satVarIndex] -= clauseWeights[index(c)];
            } //END IF (curClauseStatesInt[index(c)] == 1)
        } // END FOR all clauses c in negOccurrences[flipIndex]

        FOR all clauses c in posOccurrences[flipIndex] {
            ...
        }

    } // END IF varAssignment[flipIndex] == true
    else {
        ... // analogously with roles of posOccurrences and negOccurrences switched
    }
} // END UpdateClauseStates

```

Figure 2.5: Pseudo code of part of the method `UpdateClauseStates(int flipIndex)` in a `SATSolution` object. Only actions undertaken in clauses where the flipped variable (*false* to *true*) occurs negatively are shown

```

UpdateClauseStates(int flipIndex) {
  // invert score of flipped variable
  varScores[flipIndex] = varScores[flipIndex] * (-1);
  IF (varAssignment[flipIndex] == true) { // flip from false to true

    FOR all clauses c in negOccurrences[flipIndex] {
      ...
    }

    FOR all clauses c in posOccurrences[flipIndex] {
      curClauseStatesInt[index(c)]++;
      IF (curClauseStatesInt[index(c)] == 1) { // clause now crit. sat.
        score++;
        IF (UseWeights)
          score += clauseWeights[index(c)];
        unsatList.Remove(index(c));
        FOR all literals lit in clause c {
          IF (var(lit) != flipIndex) {
            varScores[varIndex]--;
            IF (UseWeights)
              varScores[var(lit)] -= clauseWeights[index(c)];
          }
        } // END FOR all literals lit in clause c
      } ELSE IF (curClauseStatesInt[index(c)] == 2) { // clause now sat.
        int satVarIndex = index of var. that has crit. sat. clause c before the flip
        varScores[satVarIndex]++;
        IF (UseWeights)
          varScores[satVarIndex] += clauseWeights[index(c)];
      }
    } // END FOR all clauses c in posOccurrences[flipIndex]

  } // END IF varAssignment[flipIndex] == true
  else {
    ... // analogously with roles of posOccurrences and negOccurrences switched
  }
} // END UpdateClauseStates

```

Figure 2.6: Pseudo code of part of the method `UpdateClauseStates` in a `SATSolution` object. Only actions undertaken in clauses where the flipped variable (*false to true*) occurs positively are shown

For each clause c , the counter at `curClauseStatesInt[index(c)]` is decreased by 1. Then the new counter value is inspected. If the counter has the value 0, then the clause has been broken. The evaluation function value `score` is decreased by 1, if clause weights are used, then the current clause weights are subtracted as well and the index of the currently inspected clause is added to `unsatList`, the list of all clauses currently unsatisfied. Next, the scores of all variables that occur in the – now unsatisfied – clause are increased, except the score of the flipped variable, which has already been handled at the beginning. Again, if clause weights are used, they are added. If the counter has value 1 – the clause is now critically satisfied – then only the score of the variable that critically satisfies the clause is decreased. All other, still satisfied clauses do not have to be handled.

Next, all clauses c where the flipped variable occurs positively, are checked (figure 2.6). In this case, the satisfaction status of such clauses might have changed from unsatisfied to critically satisfied or from critically satisfied to satisfied by the flip. Recall, that the variable has been flipped from *false* to *true*. For each clause c , the counter at `curClauseStatesInt[index(c)]` is now increased by 1. Again, this value is inspected afterwards. If the counter has value 1, then the clause has been critically satisfied by the flip. The evaluation function value is increased (additionally by the current weight, if weights are enabled), and the index of the clause is removed from the list of currently unsatisfied clauses. The scores of all variables in the clause, except the one of the flipped variable, have to be decreased. A counter value of 2 means, that the satisfaction status of the clause has changed from critically satisfied to satisfied. Thus, only the score of the variable that has critically satisfied the clause *before* the flip has to be updated. The index of that variable can easily be determined, since there are exactly two satisfying literals in the clause: one literal that has become *true* by the flip and one that has already been *true* before the flip. The latter has to be searched for.

After having performed these steps – checking all clauses where the flipped variable occurs positively and negatively and undertaking the certain actions – the variable scores, the evaluation function value and the list of currently unsatisfied clauses are up to date.

The method `UpdateClauseStates(int flipIndex)` has been explained for a flip from *false* to *true*. For the other case, the roles of the two lists `posOccurrences[flipIndex]` and `negOccurrences[flipIndex]` simply have to be switched: now clauses, where the variable occurs negatively, might have been satisfied by the flip and clauses, where the variable occurs positively, might have been broken. The actions that have to be performed in the clauses in each list are the same as described.

As already mentioned, it is very important that the two lists `posOccurrences[flipIndex]` and `negOccurrences[flipIndex]` are pairwise disjoint. Other-

wise, the calculation of the variable scores and the evaluation value would not work the way as it is described. This condition is checked when the problem instance is parsed. The detailed description of how this is done can be found in section 2.3.

The runtime of the method `UpdateClauseStates(int flipIndex)` depends on the number of clauses that contain the flipped variable and on the length of these clauses. Section 2.5 discusses some aspects, why calculating variable scores from scratch each time they are needed is a problem in context with runtime.

UpdateWeightedEvalValues(double[] deviations)

There is an important issue that has been left out so far: what has to be done, if the clause weights are modified after a generation has been generated or a round performed? In the pseudo code of `UpdateClauseStates(int index)`, clause weights are added and subtracted depending on the satisfaction status of the currently inspected clause. After the method has been run, all values are up to date with respect to the *current* clause weights. In a Genetic Algorithm, one might want to update the clause weights used for subsequent generations, e.g. based on the current satisfaction status of the clauses in the whole population. This could, for example, be done whenever a certain number of generations have been generated. If the clause weights are modified, then the values, which have been calculated in the update method, are not correct any more since they are based on the old clause weights. Note, that there would not be any problem if the variable scores and the evaluation function value were calculated from scratch each time as in the method `InitializeEvalValues`. Due to the continuous updating of these values, this problem has to be dealt with each time the clause weights change: the method `UpdateWeightedEvalValues(double[] deviations)` is called, which updates the variable scores and evaluation function value of each individual in the given population. The pseudo code of this method can be found in figure 2.7. The argument that is passed to the method is a double array of deviations of the old clause weights from the new ones which is calculated as

$$deviations[i] = oldClauseWeights[i] - newClauseWeights[i]$$

Clone

Finally, there is a method for cloning `SATSolution` objects, which is used in some mutation and recombination operators to duplicate a solution without reinitializing the evaluation function value and the clause states in order to save computation time.

```

UpdateWeightedEvalValues(double[] deviations) {
  FOR all clauses c {
    IF (curClauseStatesInt[index(c)] == 0) { // clause is unsat.
      score += deviations[i];
      FOR all literals lit in c {
        varScores[var(lit)] -= deviations[i];
      }
    } ELSE IF (curClauseStatesInt[index(c)] == 1) { // clause is crit. sat.
      int satVar = index of var. that crit. sat. clause c;
      varScores[satVar] += deviations[i];
    }
  } // END for all clauses c
}

```

Figure 2.7: Pseudo code the method `UpdateWeightedEvalValues(double[] deviations)` in a `SATSolution` object

2.3 Instance Input

In order to facilitate the process of evaluating algorithms for SAT, the so-called *DIMACS format* has been introduced [2]. It defines the structure of input files for SAT solvers and is commonly accepted. The SAT plugin does also make use of this input format. The user can either type in a SAT instance manually – this will rather rarely be the case – or load a text file containing an instance.

A file in DIMACS format may contain comments before the actual SAT instance is specified. Comments are marked with the character “c” at the beginning of the line. The specification starts with the line: `p cnf numVars numClauses`, where `numVars` is the number of variables occurring in the instance and `numClauses` the number of clauses. Then each clause is written in a single line by entering a positive or negative integer out of range $[1, numVars]$ or $[-numVars, -1]$ to denote the positive or negative literal of the variable with the index specified by that integer. The number zero must not occur as an integer because zero is used as a clause termination character: each line of a clause has to end with a zero. Furthermore, it has to be assured that all the indices come from the given range.

Additionally, the user can – in case he wants to import a SAT instance using the `SATLibImporter` – specify parameter values such as noise and walk probability in the input file before the section with the actual SAT instance. Parameter values have to be set manually, if the instance is created by means of the SAT Problem Form.

Figure 2.8 shows the format of the input file for the `SATLibImporter`. Note, that `#` is a special separator character between the parameter section and the

```

SAT PARAMS
WalkProbability = 0.01
NoveltyNoise = 0.5
NVarFlip = 1
UseWeights = false
SmoothProbability = 0.5
SatScale = 1
UnsatScale = 1.3
WeightUpdateFrequency = 10
SmoothParameter = 0.8
UnsatClauseProportion = 1
OldDecayFrequency = 10
OldDecayFactor = 0.5
UseUpperBound = false
UpperBound = 10
SAT PARAMS
#
p cnf numVars numClauses
1 -3 5 0
-4 7 9 0
...

```

Figure 2.8: File format for importing SAT instances.

DIMACS section of the file. This character must occur exactly once in the file. The parameter section may also be missing (including the separator character), which leads to an input file in pure DIMACS format as described above. As already mentioned, the SAT Problem Form can read files in pure DIMACS format only, the parameters have to be set manually in that case.

When parsing the input file, information about the instance is collected which is listed in figure 2.9. The two arrays of lists `posOccurrences` and `negOccurrences` store the indices of clauses where a variable x_i occurs as positive or negative literal at position `posOccurrences[i]` or `negOccurrences[i]` in the respective array. The indices of the clauses are zero-based. The literals in each clause c_i are stored – in the ordering they occur in the instance – in the array of lists `clauses` at position `clauses[i]`. It is important to note, that the literals are *not* stored zero-based in `clauses`. Storing them zero-based would lead to ambiguities with the literals x_1 and $\neg x_1$ of variable x_1 , which both would then have to be inserted as 0 into `clauses` and could not be distinguished any more.

<i>Name</i>	<i>Description</i>
varCount	number of variables in the instance
clauseCount	number of clauses in the instance
posOccurrences	array of lists storing the positive occurrences of a variable
negOccurrences	array of lists storing the negative occurrences of a variable
clauses	an array of lists storing the literals in each clause
removeClauses	list of clauses that are trivially satisfied and removed
trivSatClause	index of the trivially satisfied clause that has been found least recently
trivSatClauseCount	number of trivially satisfied clauses
litCount	total number of parsed literals in the instance

Figure 2.9: Collected information and lists built when parsing an input file.

Two very simple techniques of preprocessing are performed during the parsing process: elimination of duplicate literals and trivially satisfied clauses. If one and the same literal is encountered twice within the same clause, it is inserted into `clauses` only once. If both a literal and its complementary literal are contained within the same clause, then the whole clause is discarded, since it is satisfied anyway.

In principle, more sophisticated preprocessing techniques could be applied in order to simplify the instance. Among them, maybe the most important is *unit propagation*, which is a crucial part in complete SAT solvers. If a clause contains a single literal (*unit clause*), then this clause and all clauses containing the same literal can be removed and all remaining occurrences of the corresponding variable. Another technique is eliminating *subsumed clauses*: a clause c_1 is subsumed by another clause c_2 if, and only if, the set of literals appearing in c_2 is a subset of the set of literals appearing in c_1 .

After the whole instance has been parsed, some statistical information is calculated such as the ratio `clauses / variables` or `literals / clause`. See section 1.2 for further details. The parser itself has been generated using a parser generator [9] and extended with semantical actions.

2.4 Assignment Input

In HeuristicLab, the user has the possibility to log a run of an algorithm in a log file, where solution-specific information is put out in textual format. In case of the SAT plugin, the current variable assignment and a list of unsatisfied clauses is printed. The assignment obtained in such way could be specified when importing the problem instance using the `SATLibImporter`. The imported problem instance will have set its property `BestKnownSolution` to the `SATSolution` created from

```

TYPE: VAR_ASSIGNMENT
NUM_VARS: n
ASSIGNMENT
1 -2 3 4 -5 6 ... n
ASSIGNMENT

```

Figure 2.10: File format for importing variable assignments using the `SATLibImporter`.

the specified variable assignment. The assignment has to be copied from the log in `HeuristicLab` into a text file which has the format as defined in figure 2.10. The variables are numbered from 1 to n , “minus” indicates that the respective variable has the value *false*. The variables are separated by exactly one space character.

2.5 Variable Scores

The concept of a score of a variable, as it is used in the SAT plugin implementation, stems from GSAT, one of the first stochastic local search algorithms for SAT [1]. In order to understand, what is the idea of variable scores, first a brief description of the GSAT algorithm is given.

As most stochastic local search solvers for SAT, GSAT is based on a 1-exchange neighbourhood in the search space of all complete truth assignments. Two solutions are neighbours if, and only if, they differ in the truth assignment of exactly one variable. In each search step, one variable in the assignment is flipped and a new solution is retrieved. The quality of a solution is evaluated by counting the number of currently unsatisfied clauses in the formula – this is different from the evaluation function used in the SAT plugin, where the number of currently *satisfied* clauses are counted. The goal is to find a solution whose evaluation function value equals zero (in GSAT or `clauseCount` in the SAT plugin), that is, a model of the given formula. GSAT tries in a “greedy” manner to improve a solution by flipping the variable that, when flipped, leads to the maximum decrease in the number of unsatisfied clauses. Note, that this is different from the number of clauses that become satisfied by the flip, because a flip might also break some clauses. In context of GSAT, the decrease as an outcome of a variable flip is called *score* of that variable and can be computed in a naive, but straight-forward way by subtracting a solution’s quality value after the flip from its quality value before the flip.

The score of variable in context of the SAT plugin is the *increase* in the number of *satisfied* clauses, due to the different evaluation function used (recall, that SAT

is considered as a maximization problem in HeuristicLab). The value is actually the same as if the score described in GSAT would be used, because the score within the SAT plugin could in a straight-forward way be calculated by subtracting the number of satisfied clauses before the flip from the number of satisfied clauses after the flip. The following equations hold:

$$unsat_{beforeFlip} - unsat_{afterFlip} = sat_{afterFlip} - sat_{beforeFlip}$$

And in general:

$$unsat = clauseCount - sat$$

$$sat = clauseCount - unsat$$

It is obviously, that the score of a particular variable can also be negative, which indicates a worsening step when flipping that variable.

Variable scores are used in many of the recombination operators described in section 2.8 as well as in some mutation operators. Therefore, it pays to update the scores of the variables whenever a solution is modified rather than to calculate them from scratch, e.g. during the mutation phase. See section 2.2.2 for details on the update procedure.

Now, a closer look shall be taken on the situations where the variable scores have to be increased or decreased. The role of clause weights is explained in section 2.6.2 and is left out here for the sake of simplicity.

Assume, the variable scores have been reset to 0 and are initialized from scratch as shown in figure 2.4. If a clause c is currently unsatisfied, i.e. `curClauseStatesInt[index(c)]` has value 0, then the scores of all variables within that clause are increased, because flipping one of them will satisfy the clause. In critically satisfied clauses (`curClauseStatesInt[index(c)] = 1`), the score of the single variable whose literal is positive has to be decreased, since flipping that variable will break the clause. This can be seen as a kind of “penalty” for this variable. Finally, if a clause is satisfied, all scores remain unchanged, because flipping one of them will not cause the clause to become unsatisfied.

Now a variable is flipped. Some clauses where the flipped variable occurs might have changed their satisfaction status. In any case, the score of the flipped variable changes its sign and must not be taken into account during further calculations. The following actions have to be undertaken, depending on the change of the satisfaction status of a clause:

- critically satisfied \rightarrow unsatisfied: the scores of all variables within that clause are increased
- satisfied \rightarrow critically satisfied: the score of the variable whose literal critically satisfies the clause is decreased (“penalty”).

- unsatisfied \rightarrow critically satisfied: the scores of all variables within the clause are decreased, since flipping on of them can not satisfy the clause any more.
- critically satisfied \rightarrow satisfied: the score of the so far “penalized”, critically satisfying variable is increased, because flipping that variable will not break the clause.

The time complexity of calculating all variable scores from scratch (together with the evaluation function value) is $O(m \cdot CL(n))$, where m is the number of clauses, $CL(n)$ an upper bound on the length of the clauses, i.e. the number of literals in a clause, and n the number of variables in the CNF formula.

On the contrary, the method `UpdateClauseStates(int flipIndex)` described in section 2.2.2 performs an update of the evaluation function value and all variable scores that have been affected by the flip of a variable in time $O(CD(n) \cdot CL(n))$, where $CD(n)$ is an upper bound on the cardinality of the sets $C_{dep}(F, x)$, the sets of clauses dependent on the flipped variable x (section 2.2.2).

2.6 Evaluation Operators

2.6.1 Standard

When solving an instance of a combinatorial optimization problem, the goal is to find a solution which is globally optimal. During the process of searching or building a solution, it is necessary to evaluate the current solution in some way in order to measure its quality. The decision about further steps in the search are then based on the quality. In population-based approaches such as Genetic Algorithms, the quality of an individual, i.e. a solution, determines the fitness and in a broader sense the probability for selecting that individual for recombination.

In context with SAT, an obvious evaluation method is counting the number of clauses which are currently satisfied or unsatisfied. Since the first – counting satisfied clauses – is used in the SAT plugin implementation, SAT can be considered as a maximization problem in that case. The globally optimal solution – a model – for a given, satisfiable SAT instance therefore has evaluation function value `clauseCount`, which is the number of clauses in the problem instance.

In figure 2.4, the basic concept of evaluating a solution has been shown. The satisfaction status of a clause is determined by the truth values of the literals within that clause. In the implementation, an additional, third satisfaction state has been introduced: a clause is *critically satisfied*, if it is satisfied by exactly one literal. This extension is necessary, because a solution is *not* evaluated from scratch each time, rather the evaluation function value is updated whenever a solution is modified. Note, that evaluating from scratch requires traversing all clauses and

checking all literals within each clause – a tremendous overhead, especially, when using a population-based algorithm, where evaluation has to be done for each individual.

Together with the evaluation function value, the scores of all variables are continuously updated, which reduces the the required computation time at a large extent.

See section 2.2.2 and 2.5 for details on the update procedure of the evaluation value and the variable scores.

2.6.2 ClauseWeights

Although listed in this section, this evaluation operator is not a standalone operator, but rather an extension of the standard evaluation operator. As shown in the explanations of the methods `InitializeEvalValues` and `UpdateClauseStates(int flipIndex)` in sections 2.4 and 2.2.2, the individual weight of a particular clause has to be added to or subtracted from the evaluation function value and the variable scores depending on the satisfaction status of that clause.

The idea of introducing clause weights stems from the concept of *Dynamic Local Search Algorithms (DLS)*. In a DLS algorithm, solution components are penalized in order to allow the algorithm to escape from local optima. The penalties affect the evaluation function value of the solution. Whenever the algorithm gets stuck in a local optimum, the penalty weights of some solution components are increased, which leads to a degradation – in context with SAT as a maximization problems as presented in this report – of the evaluation function value of the solution. Thus, improving steps will become possible, since the neighbours of the solution might not have been affected at the same extent.

Generalizing the concept of clause weights to population-based approaches such as Genetic Algorithms is not that easy as it might seem. The solution components that are penalized in SAT are the currently unsatisfied clauses. In a population of a Genetic Algorithm, clauses might be unsatisfied in one individual and satisfied in the other, so a way has to be found in order to classify clauses as satisfied or unsatisfied and to modify the clause weights based on that information. One way of doing this has been shown in section 2.1.3.

2.7 Mutation Operators

Generally, a mutation operator in a Genetic Algorithm introduces small modifications of a solution with small probability, called the *mutation rate*. This is very important when thinking of the basic idea of a Genetic Algorithm: the individuals of a population should evolve towards the optimal solution by combining informa-

tion of each individual during the recombination stage. Without mutation, some alleles might drift to a fixed value after a number of generations have been generated, which causes the algorithm to prematurely converge if the fixed value is not part of the optimal solution. Mutation, despite of its relatively small impact on an individual, helps to bring back lost alleles and to maintain diversity in the population.

All mutation operators described in the following – except `NVarFlip` and `RandomRestart` – choose one variable in the individuals’ variable assignment and flip it. Some operators such as `CDRW` or `UCDRW` are conflict-directed, which means that a variable appearing in a currently unsatisfied clause is chosen and flipped, which forces at least the one particular clause to become satisfied. The operators `NoveltyPlus` and `WalkSATSKC` are in fact local search algorithms and can be considered as sort of “sophisticated” mutation operators which select the variable to be flipped in a more or less greedy manner in order to improve the mutated individual or – as `WalkSAT-SKC` does – not to cause damage in terms of breaking clauses at random. There are also two pure random mutation operators, `NVarFlip` and `RandomRestart`. The latter one does at first glance not seem to make sense: it simply reinitializes *all* variables of the individual during mutation at random. In context of SAT and population-based algorithms, such an operation is not that far-fetched. Initializing a variable assignment at random often leads to the satisfaction of at least two third of all clauses of the given problem instance. The probability to generate an individual by `RandomRestart` mutation that differs in the assignment of all variables from the individual before mutation is very low. However, it will have to be done empirical research to find out on which problem instances this can be used profitably, and which recombination operators may show improved performance together with that mutation operator.

The mutation operators are described within the context of a Genetic Algorithm (GA), but have also been implemented for Evolutionary Strategies (ES) in HeuristicLab. The algorithms, which have inspired some of the mutation operators, are have been presented in several publications, but are also covered in [1], together with a general introduction into local search techniques.

2.7.1 NVarFlip

The “N” in the name of this operator stands for the number of variables that will be flipped and can be specified by the user. The default value of the parameter is 1. If it is set to a value greater than 1, then the implementation ensures that N distinct variables will get flipped, so it will never happen that one and the same variable is selected and flipped twice or even more often at random.

2.7.2 CDRW

CDRW stands for *Conflict-Directed Random Walk*. The variable to be flipped is selected in a two-stage process. In the first, a clause that is unsatisfied under the current variable assignment is chosen uniformly at random. In the second stage, one of the variables appearing in the clause which has been chosen in the first stage is selected uniformly at random and flipped. The flip will at least satisfy the particular chosen clause, maybe some more, but can also break clauses.

2.7.3 UCDRW

The *Uniform Conflict-Directed Random Walk* operator first builds a list of all variables appearing in currently unsatisfied clauses. Then a variable from this list is selected uniformly at random and flipped. Again, this flip will satisfy at least the clauses where the variable occurs, but might break other clauses. Note that the set of variables which are possibly selected for flipping is exactly the same as in CDRW, but the choice in UCDRW is biased towards variables that occur more than once in unsatisfied clauses.

2.7.4 NoveltyPlus

Novelty+ is a local search algorithm based on the WalkSAT architecture. A search step in the local search algorithm works as follows: first, a currently unsatisfied clause is chosen uniformly at random. When choosing a variable to be flipped, the ages and scores of the variables appearing in the selected clause are taken into account. The age of a variable is simply the number of search steps, i.e. variable flips, that have been performed since that variable has been flipped last. Whenever a variable is flipped, the age is reset to 0 and increased by 1 with every search step. The score of a variable is the improvement in the evaluation function value when flipping that variable: the number of clauses that become unsatisfied by the flip subtracted from the number of clauses that become satisfied. This score will be negative, if more clauses are broken than satisfied. This scoring function has been used in the GSAT algorithm, one of the first local search algorithms for SAT, and will also play an important role in the recombination operators described in section 2.8. See section 2.5 for further details on the score of a variable.

Within the clause that has been chosen during the first stage of the selection process, the scores of the variables appearing in that clause are calculated as described. Now the ages come in: if the best-score variable does not have minimal age, then this is the best choice: the variable is flipped. Else, if it has minimal age it is still flipped with probability $1 - p$, p is the *noise* parameter (`noveltyNoise`) of the algorithm. In the other case the second best variable is chosen with probability

p. At the beginning of each search step, Novelty+ decides whether to perform a search step as described with probability $1 - wp$, or to do a conflict-directed random walk step as in UCDRW with probability wp . The random walk parameter wp (`walkProbability`) can be set by the user.

When using Novelty+ as a mutation operator, the algorithm can be applied without any changes on the selected individual. The process of mutating an individual is in this case simply performing a local search step on the individual. As mentioned above, this operator can be considered as to perform a “greedy” mutation. By setting the parameters – random walk probability and noise – the user may take influence on the mutation process.

2.7.5 WalkSAT-SKC

Like NoveltyPlus, WalkSAT-SKC is a local search algorithm based on the WalkSAT architecture. Again, a clause currently unsatisfied is chosen during the first stage of the variable selection process, then a variable within that clause is chosen according to some heuristic function. Different from NoveltyPlus, the age of a variable does not play a role in the selection process. The scoring function used in the variable selection process counts the number of clauses that will get broken when flipping that variable. The goal is to flip a variable that has a score of 0, because flipping such variable does not break any clauses but satisfies at least the one clause where the variable occurs. If no such variable exists, then the variable with minimal score is flipped with probability $1 - \text{walkProbability}$, in the other case a variable from the clause is chosen uniformly at random and flipped. WalkSATSKC can therefore be seen as less greedy than NoveltyPlus.

As with Novelty+, the WalkSAT-SKC algorithm can be applied without any changes for the process of mutation.

2.7.6 RandomRestart

This mutation operator is inspired by the method of statically restarting the search process in local search algorithms after a number of search steps have been performed. While in local search algorithms the use of such method might be influenced by observing the progress of the search, it is performed as any conventional mutation operator based on the specified mutation rate in this implementation.

The variable assignment of the individual that has been selected for mutation is completely reinitialized at random: each variable takes the value *true* or *false* with equal probability. As mentioned at the beginning of this section, it may seem that such a heavily randomized mutation operator disturbs the process of converging towards the optimal solution in the population of a Genetic Algorithm. Indeed one will have to be careful when using that operator with high mutation

rates, but the combination with particular recombination operators might also influence the performance of this mutation operator.

2.7.7 GSAT

The GSAT mutation operator performs a local search step as the GSAT algorithm, which is described in section 2.5.

2.8 Recombination Operators

Recombination is the primary operator in Genetic Algorithms. The development of suitable recombination operators is dependent on the problem representation (binary, real-valued, discrete) and the problem itself. Ideally, an offspring should benefit from both parents at an extent as large as possible.

The recombination operators described in the following and some aspects mentioned such as diversity behaviour have been taken from [3] and [4].

During the descriptions of the recombination operators, the following terminology will be used: two parents A and B, which have been selected from the current population, produce one offspring C. As with the mutation operators, the recombination operators are described within the context of a Genetic Algorithm, but are also available for Evolutionary Strategies.

A recombination operator used in context with SAT solving should take into account as much as possible the structure of the given problem instance and the current satisfaction status of the clauses in the selected parents in order to produce an offspring that benefits from this information. An offspring should in the best case have clauses satisfied that have previously been unsatisfied in the parents, but should also maintain previously satisfied clauses. To achieve this, the clauses will have to be compared pairwise in the parents with respect to their satisfaction status. The variables in the offspring can then be assigned values based on the collected information. There are three possible types of relation between the satisfaction status of a clause c in the parents A and B:

- clause c is satisfied in both parent A and parent B
- clause c is unsatisfied in both parent A and parent B
- clause c is satisfied in either parent A or parent B, but not in both

If a clause c is *unsatisfied* in both parents, then the clause can be satisfied in the offspring by assigning a variable occurring in c the complementary value with respect to the parents. The decision which variable to choose in clause c can be

guided by the scoring function of the variables as described in section 2.5. Thus, the improvement achieved by the assignment in the offspring can be maximized.

If a clause is *satisfied* in both parents, it might seem obvious to simply copy the values of the variables in one parent to the offspring. But the variables in the parents can have different values, and one parent should not be neglected. A solution could be to select the one variable that, when get flipped, has the smallest impact and to assign that variable such that the corresponding literal is true in the offspring.

Finally, if a clause has different satisfaction status in both parents, it might be obvious to take the values from the parent where the clause is satisfied. The last three situations have been taken into account in the operators **CC**, **CCTM** and **FF**.

One of the operators described in the following – the classical *Uniform Crossover UC* – is undirected: the variables in the generated offspring take values either from parent A or parent B with equal probability. All other recombination operators take into account the current satisfaction status of the clauses in both parents and make use of this information when assigning values to the variables in the offspring. The aspects the operators focus on when generating an offspring – e.g. satisfying as many clauses as possible or maintaining satisfied clauses – are different in each operator.

In each of the operators except **UC**, the clauses have to be traversed. In the given implementation, they are traversed as they occur in the problem instance.

Empirical studies of the behaviour with respect to the diversity in the population have shown that the quality of a good crossover operator can not only be measured by the amount of improvement that is achieved in the evaluation function value of the individuals. It is important to find a compromise between solution quality and diversity of the population. Diversification allows the algorithm to explore a larger area of the search space. Another measurement for the quality of a recombination operator is the entropy of the population, which can take values between 0 and 1. The smaller the entropy, the more similar the individuals in the population. It has been observed that **CC** and **CCTM** lead to a better diversity than **UC** or **FF**. Furthermore, these two operators have a better entropy.

Nevertheless, the performance of the recombination operators together with certain mutation operators is likely to be dependent on the given class of instance.

2.8.1 UC

The classical *Uniform Crossover*. Variable x_i in the offspring takes the value of the respective variable either from parent A or parent B with equal probability. This operator is completely undirected, the structure of the instance and the current satisfaction status of the clauses in the parents is in no way taken into consideration.

2.8.2 CC

In *Corrective Clause Crossover*, the intention is to improve the offspring by inspecting the current satisfaction status of the clauses in both parents.

For each clause c such that c is unsatisfied both in parent A and parent B, and for each variable x_i in clause c , where i is the position of the variable in clause c , the sum of the scores of variable x_i in parent A and parent B is computed. The variable with maximal sum of scores is then assigned the flipped value in the offspring. See section 2.5 for further details on the score of a variable. Since only clauses that are unsatisfied in both parents are inspected, it is necessary to flip that variable in the offspring to make the currently inspected clause satisfied. By selecting the best variable with respect to the variable scores in the parents, the offspring will be improved as much as possible. Once a variable in the offspring has been assigned, it is excluded from further assignments.

All variables in the offspring that have not yet taken values as described take the value either from parent A or parent B with equal probability.

2.8.3 CCTM-SBS and CCTM-SBS-NEW

The *Corrective Clause and Truth Maintenance Crossover* focuses on clauses that are unsatisfied or satisfied in both parents. “SBS” stands for *step by step*: first all clauses that are unsatisfied in both parents are inspected and treated exactly as in the *Corrective Clause Crossover*. Again, variables that once have been assigned are of course excluded from multiple assignments. Next, each clause c that is satisfied in both parents is inspected. For each variable x_i in clause c , the sum of the scores of variable x_i in both parents is computed, where i is the position of the variable in clause c . Variable x_k in the offspring is then assigned the value of variable x_k in parent A, where k is the position of the variable where the sum of the scores is minimal.

Again, all variables in the offspring that have not yet taken values as described take the value either from parent A or parent B with equal probability.

The difference between the operators CCTM-SBS and CCTM-SBS-NEW – the functionality of the latter has been described above – is, that in CCTM-SBS only variables are taken into account during the second phase, i.e. when inspecting clauses which are satisfied in both parents, whose literal has the value *true* in at least one parent.

2.8.4 CCTM-CBC and CCTM-CBC-NEW

This operator works exactly like CCTM-SBS, but traverses the list of clauses one after the other. “CBC” stands for *clause by clause*. The reason for different behaviour of CCTM-SBS and CCTM-CBC stems from the fact that variables which have already

been assigned are excluded from further assignments. In many situations CCTM-SBS will show a greedier performance, because in the first step all clauses which are unsatisfied in both parents are taken into consideration. In CCTM-CBC, variables in such clauses could have already been assigned during the inspection of prior clauses and are therefore excluded and can not be used a second time for satisfying a clause in the offspring.

Again, the difference between CCTM-CBC and CCTM-CBC-NEW is the same as described in the preceding section.

It is clear that the behaviour of these two implementations of the CCTM crossover is dependent on the given problem instance.

2.8.5 FF

The *Fleurent Ferland Crossover* inspects clauses with different satisfaction status in both parents, i.e. clauses that are satisfied in parent A and unsatisfied in parent B or vice versa. For each variable x_i such that x_i appears in a clause with different satisfaction status, x_i in the offspring takes the value from that parent where the currently inspected clause is satisfied.

This operator requires relatively little computational effort because clauses that do not have different satisfaction status can be skipped during the traverse of the clause list and no variable scores have to be taken into account.

All variables in the offspring that have not yet taken values as described take the value either from parent A or parent B with equal probability.

2.8.6 GASAT

GASAT is a hybrid evolutionary algorithm for SAT which has been presented in [5]. In *GASAT*, new solutions are generated by means of a recombination operator which are then improved by a tabu search mechanism.

This recombination operator works step by step like CCTM-SBS and considers clauses which are unsatisfied and satisfied in *both* parents, but the selection of the variable in the latter case is different.

In the *GASAT* crossover operator, the first step is performed analogously to the CC crossover. In the second step, all clauses c that are satisfied in both parents are inspected: for each clause c and for each variable x_i which appears in c , where i is the position of the variable in clause c , the variable x_i in the offspring takes the value *true* (respectively *false*) if, and only if, the variable has the value *true* (respectively *false*) in both parents.

Different from the operators described so far, the variables that have not yet been assigned a value, are assigned a random value independently from the parents A and B.

2.9 Weight Update Operators

A weight update operator should analyze a solution or, in case of population-based algorithms, a whole population of solutions and modify the clause weights based on the gathered information. The idea is to focus the search on particular clauses. Most of the time, the weights of currently unsatisfied clauses are increased. As mentioned in section 2.1, the weights can be decayed over time by multiplying them with factor `oldDecayFactor` every `oldDecayFrequency` generations. The decay is performed *before* the weights are modified. The idea is to emphasize clauses which have been unsatisfied least recently. Decaying weights is not a specific feature of a particular weight update operator, rather this can be done in any implemented operator as with using an upper bound for clause weights as defined by `upperBound`. If bounded weights are used, then weights are not increased such that the value exceeds the bound – the value is rather set to `upperBound`. An alternative would be to decay the weights if their value is beyond the bound after increasing. However, the actual strategy of weight updating which performs best and most robust on a variety of instances will have to be determined by testing.

2.9.1 SAPS

Scaling and Probabilistic Smoothing is a Dynamic Local Search Algorithm for SAT [8] which has inspired this weight update operator. A brief introduction to Dynamic Local Search algorithms can be found in section 2.6.2. In the original algorithm SAPS, the weights of unsatisfied clauses are scaled by factor `unsatScale` and then smoothed with probability `smoothProbability` towards the average clause weight after scaling: $clw(c) := clw(c) \cdot r + (1 - r) \cdot \bar{w}$ where r is the smoothing parameter and \bar{w} the average clause weight after scaling.

In the weight update operator SAPS, additionally clauses which are currently satisfied are scaled by factor `satScale`, the smoothing phase is implemented as in the algorithm SAPS. However, by setting the factor `satScale` and `oldDecayFactor` to 1, the behaviour of the weight update scheme as in the SAPS algorithm can be achieved. The decision, if a clause is unsatisfied in a population-based algorithm, is made as described in section 2.1.3.

2.9.2 WIUSC

In the operator *Weight Increase of Unsatisfied Clauses*, the decision about the satisfaction status of a clause is made in the same way as in the SAPS operator. The weights of clauses which have been considered as unsatisfied are increased by 1, weights of satisfied clauses remain unchanged. A decay of all weights before

increasing may be performed by setting the parameters `oldDecayFrequency` and `oldDecayFactor`.

2.10 Neighbourhood Operators

A local search algorithm tries to improve a given solution by modifying some solution components, most of the time in a greedy manner, in order to maximize the gain in the solution quality. Modifying a solution corresponds to moving through the search space of the problem instance from one solution to a neighbouring solution. The neighbourhood of a solution, i.e. the set of all solutions which are considered to be neighbours, is determined by the amount of modification applied to a solution. In SAT, typically a 1-exchange neighbourhood is used: two solutions are neighbours if they differ in the value of exactly one variable in the assignment.

This section lists the neighbourhood operators for the metaheuristic approaches of Simulated Annealing and Tabu Search. Performance studies will have to show, which operator is favourable. For now, these operators are mainly intended to be used for experimental purposes.

2.10.1 Simulated Annealing

NoveltyPlus

This operator simply returns the solution chosen during a search step according to NoveltyPlus as described in section 2.7.4.

WalkSAT-SKC

The same as above, but the solution is chosen according to WalkSAT-SKC as described in section 2.7.5

CDRW

A solution is returned based on a conflict-directed random walk as described in section 2.7.2.

2.10.2 Tabu Search

Different from Simulated Annealing, a neighbourhood operator for Tabu Search returns a *set* of solutions, where a solution is then chosen from by the Tabu Search algorithm.

SimpleOneExchange

A simple 1-exchange neighbourhood as described above: the set of all solutions that differ in the value of exactly one variable.

SimpleCDOneExchange

The set of all solutions, that differ in the value of exactly one variable which occurs in a currently unsatisfied clause. CD stands for “conflict directed”.

2.11 Memory Operators

A Tabu Search algorithm uses a memory to influence the search process. Whenever a search step has been performed, the solution or some relevant information is stored for later diversifying, i.e. restarting the search in areas of the search space which have not been searched so far at a large extent, or intensifying the search, i.e. restarting from elite solutions.

Since the Tabu Search algorithm in HeuristicLab is currently under development, these operator has been added for the sake of completeness and should be considered as a test implementation.

2.11.1 Standard

When adding a solution into memory, the values of the variables are inspected. For each variable, two counters `trueCount` and `falseCount` are updated depending on the value of the variable in the solution which should be added into memory. When diversifying the search, a new solution is created by assigning a variable the value *true* if the counter value in `trueCount` is smaller than the counter value in `falseCount` and *false* in the other case. When intensifying the search, the two cases are applied vice versa.

These techniques do not take into account the quality of the solution constructed in such way. More refined versions of intensification and diversification techniques will still have to be developed.

Conclusion

In this report, a plugin implementation modelling the *Satisfiability Problem (SAT)* in HeuristicLab has been presented. The focus has been put on technical details concerning (weighted) solution evaluation and techniques for influencing variable selection, because these are the core parts of the plugin and play an important role in runtime performance.

HeuristicLab is a framework where algorithms and problems are separated from each other, which makes it suitable for testing different – already implemented or new – algorithms on SAT on a broad basis. It does not require a lot of effort, e.g. to introduce new recombination or mutation operators for Genetic Algorithms or new local search heuristics due to the operator concept of HeuristicLab.

The algorithms provided in HeuristicLab should rather not be expected to, right away, be competitive with highly optimized, state-of-the-art SAT solvers when run on a SAT instance, especially when compared in terms of runtime. At the moment, the intention should be to test various (population-based) algorithms on certain instance classes for finding out suitable parameter settings by empirical analysis. Any observations made during the testing phase, which, when translated into new ideas and concepts for algorithms or operators, are promising to improve performance, can conveniently be introduced in the plugin.

Bibliography

- [1] H.H. Hoos and T. Stützle, “Stochastic Local Search: Foundations and Applications”, Elsevier/Morgan Kaufmann Publishers, San Francisco, CA, USA 2005
- [2] ”Satisfiability - Suggested Format”, taken from SATLIB [6]
- [3] Frédéric Lardeux, Frédéric Saubion and Jin-Kao Hao, “Recombination Operators for Satisfiability Problems”, Proc. of the 5th Metaheuristics International Conference (MIC’03), August 2003, Kyoto, Japan
- [4] Frédéric Lardeux, Frédéric Saubion and Jin-Kao Hao, “Recombination Operators for Satisfiability Problems”, Proc. of the 6th International Conference on Artificial Evolution, Lecture Notes in Computer Science, Springer 2003, pages 103-114, Marseille, France
- [5] Jin-Kao Hao and Frédéric Lardeux, “Evolutionary Computing for the Satisfiability Problem”, Applications of Evolutionary Algorithms, 2003, vol. 2611, series LNCS, pages 258-267, University of Essex, England, UK
- [6] H.H. Hoos and T. Stützle, “SATLIB: An Online Resource for Research on SAT.” In: I.P. Gent, H. v. Maaren, T. Walsh, editors, SAT 2000, pp.283-292, IOS Press, 2000. SATLIB is available online at www.satlib.org
- [7] Dave A. D. Tompkins, Holger H. Hoos, “UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT.”, LNCS 3542: Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), pages 305-319, 2004.
- [8] Frank Hutter, Dave A. D. Tompkins, Holger H. Hoos, “Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT”, LNCS 2470: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP’02), pages 233-248, Springer Verlag, 2002.

- [9] Mössenböck, H.: Coco/R for various languages – Online Documentation. <http://www.ssw.uni-linz.ac.at/Research/Projects/Coco>
- [10] S. Wagner, M. Affenzeller ”The HeuristicLab Optimization Environment”. Technical Report. Institute of Formal Models and Verification, Johannes Kepler University Linz, Austria. 2004, Website: www.heuristiclab.com
- [11] E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design (FMSD), vol. 19, number 1, Kluwer 2001.
- [12] Y. Xie, A. Aiken, “Scalable Error Detection using Boolean Satisfiability”, POPL’05, January 12–14, 2005, Long Beach, California, USA.
- [13] SAT Solver Competition, www.satcompetition.org