GSAT and Simulated Annealing — a comparison

A. Beringer and G. Aschemann and H. Hoos and M. Metzger and A. Weiß

Intellektik!

Forschungsbericht AIDA–94–01

Darmstadt,

Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstraße 10
D–6100 Darmstadt
Germany

Abstract

The question of satisfiability for a given propositional formula arises in many areas of AI. Especially finding a model for a satisfiable formula is very important though known to be $\mathcal{NP}$-complete. There exist complete algorithms for satisfiability testing like the Davis-Putnam-Algorithm, but they often do not construct a satisfying assignment for the formula. Furthermore, existing complete procedures are not practically applicable for more than 400 or 500 variable problems and in practice take too much time to find a solution. Recently, a (in practice) very fast, though incomplete, procedure, the model generating algorithm GSAT, has been introduced and several refined variants were created. Another method is Simulated Annealing (SA). Both approaches have already been compared with different results. We clarify these differences and do a more elaborate comparison using *identical* problems for both algorithms. We show that the performance of an already optimized variant of GSAT and an ordinary SA algorithm are more or less the same and that these attempts to further improve the performance of basic GSAT have lead to a procedure very similar to SA. Moreover we investigate how these algorithms can be parallelized.

1 Introduction

The question of satisfiability for a given propositional formula arises in many areas of AI, e.g. theorem proving, constraint satisfaction or non-monotonic reasoning. Especially finding a model for a satisfiable formula is very important, but unfortunately $\mathcal{NP}$-complete. There exist complete algorithms for satisfiability testing like the Davis-Putnam-Algorithm [2], but they often do not construct a satisfying assignment for the formula. Furthermore, existing complete procedures are not practically applicable for more than 400 or 500 variable problems and take too much time to find a solution for smaller problems. So the search for (in practice) faster procedures began. One such very fast, though incomplete, procedure is the model generating algorithm GSAT together with its variants, first described in [13]. Another method is Simulated Annealing (SA) [6], which is often used for optimization and constraint satisfaction problems and can be easily adapted to neural networks. Consequently, it is easy to parallelize. Both approaches have already been compared, with different results [12, 14]. At first sight, both comparisons seemed unfair in some points, so we tried to clarify these differences and do a more elaborate comparison of the amount of work done by the two algorithms. We run GSAT and SA on *identical* problems, using the *same* random number generator. We consider propositional random formulas in conjunctive normal form, each clause consisting of three literals (3-CNF), comparing problems with 100, 200, and 300 variables and a ratio of 4.3 for clauses/literals. This has been reported to be a class of very hard problems for the Davis-Putnam-Procedure as they are neither under- nor over-constrained.

We found that the performance of an already optimized variant of GSAT and an ordinary SA algorithm are more or less the same and that the attempts to further improve the performance of basic GSAT lead to an algorithm very similar to SA. As it is expected, that in the future the most powerful computers will be massively parallel ones, we also investigated how GSAT and SA can be parallelized. It seems that SA is better suited for parallelization than GSAT as it does not use global information to make its decisions. This is explained in detail in Section 6. In Section 2 we give an overview over the similarities and differences of both approaches. Our experimental results are reported and evaluated in Section 5 and related work is presented in Section 7.

2 GSAT and Simulated Annealing

GSAT and SA mainly rely on the same ideas. Both operate on a kind of *energy function* E, searching for its global minima. Unfortunately, both may get trapped in *local* minima which are not global. So strategies to escape from these local minima are necessary. We will first describe the general idea underlying both algorithms and then go into the details of both algorithms. Then we will give a unifying algorithm *GenSAT* for both methods within the GenSAT-framework introduced by [4] to show the similarities in the structure and ideas of both algorithms.

2.1 The general idea

Both algorithms, GSAT and SA, start with a random truth assignment for the variables of the energy function, a socalled *configuration*. If this is not already a global minimum, then successively variables are selected, whose assignment will be alternated (a so-called *flip* of the variable). This selection is done according to certain preferences depending on the energy change, the flip of a given variable V will cause. This difference $\Delta E(V)$ between the energy values of the new and the old configuration is called the *score* of the variable. A very low (negative) score means a deep decrease of the energy and usually is preferred. The selection of variables is repeated until a global minimum of the energy function E is found or the maximal number of iterations is reached. The search for a global minimum may then be restarted with a new random assignment.

3 Simulated Annealing

Simulated Annealing (SA) is a probabilistic optimization method which operates on an energy function E, dependent on a set of variables X_i. An assignment for these variables is called a *configuration.* The procedure walks in the space of these configurations, where the direction of the next step is dependent on the energy value of the neighbouring configurations. The neighbouring configurations are those differing only in the assignment of one variable. In general, SA has the tendancy to select configurations with lower energy. A step from configuration α to β is accepted with probability $P_T(\alpha \rightarrow \beta) = P_T(\Delta E)$ where T is a control parameter called the *temperature* and $\Delta E = E(\beta) - E(\alpha)$.

At low temperatures (near zero) improving steps (with $\Delta E < 0$) are likely to be perfomed and others are not. A high temperature gives a high probability even for bad moves and thus the procedure has the possibility to escape from local optima. One starts with a high temperature T_{max} and if

$$\frac{P_T(\alpha \rightarrow \beta)}{P_T(\beta \rightarrow \alpha)} = \exp(-\frac{E(\beta) - E(\alpha)}{T}) \tag{1}$$

holds, then it can be proved that SA always converges to a local minimum of the energy function as $T \rightarrow 0$, i.e. runs into a stable state.

Furthermore [3] proved that SA converges to a global minimum if the temperature is decreased in infinitesimally small steps. But in practical applications, a finite time approximation must be used. The procedure iteratively does several moves and then decreases the temperature according to a certain law called the annealing schedule. It terminates after reaching $T_{min} > 0$. Therefore SA is incomplete in practice and one often uses restarts of the search to reduce this problem. Many annealing schedules have been tried and some of them are listed below, where ΔT is a fixed maximal distance between two successive values of the temperature and t is the time.

1. $T(t+1) = T(t) - \Delta T$
2. $T(t+1) = T(t) - \Delta T/t$
3. $T(t+1) = T_{max}/(t+1)$
4. $T(t+1) = T_{max}/\exp(t * \Delta T)$
5. $T(t+1) = T(t) - \Delta T/\exp(t)$

The transition probabilities $P_T(\Delta E(X))$ have to be chosen such that these distributions obey equation 1. Otherwise it cannot be guaranteed that the algorithm reaches a stable state. The following very common distributions fullfill the necessary requirements.

- Glauber distribution:

$$P_T(\Delta E(X)) = \frac{1}{1 + \exp(\frac{\Delta E(X)}{T})}$$

- Metropolis distribution:

$$P_T(\Delta E(X)) = \begin{cases} 1 & ,\Delta E(X) <= 0 \\ \exp(-\frac{\Delta E(X)}{T}) & , \text{otherwise} \end{cases}$$

3.1 Propositional Logic and Energy Functions

Propositional formulas in general can be related to energy functions in the following way. We construct the energy function such that it has as many variables as the propositional formula and each variable of the energy function corresponds to one of the propositional variables of the formula. For a given 0-1-assignment of these variables the value of the energy function gives the number of unsatisfied clauses in the propositional formula. For example the formula $(A \vee B) \wedge (\neg B \vee C)$ is expressed by the energy function $1 - A + AB - BC$. It can be seen easily that its models correspond to the variable assignments with least value of the energy function.

Consequently, the energy function is positive semi-definite and has global minima with value 0 if the formula is satisfiable, corresponding to the truth assignments, where all clauses of the propositional formula are satisfied. This means that the variable assignments where the energy function has a global minimum give the models of a satisfiable formula. As GSAT and SA search for global minima of a given energy function, they can be used to find models of a given satisfiable formula. In the following we give a formal way to compute the energy function from a given propositional formula due to [8]. As we consider formulas in 3-CNF we can use a simplified procedure. A formula Φ in 3-CNF corresponds to

$$\Phi \equiv \bigwedge_{i=1}^{n} \phi_i$$

where the clauses ϕ_i are defined as

$$\phi_i \equiv (\overline{X}_{i_1} \vee \overline{X}_{i_2} \vee \overline{X}_{i_3}) \quad , \overline{X}_{i_j} \in \{X_{i_j}, \neg X_{ij}\}.$$

The *penalty* P for a clause ϕ_i is defined as follows

$$P_{\phi_i} = \prod_{j=1}^{3} \left\{ \begin{array}{ll} (1 - X_{i_j}) & , \overline{X}_{i_j} = X_{i_j} \\ X_{i_j} & , \overline{X}_{i_j} = \neg X_{i_j} \end{array} \right\}$$

and we obtain the corresponding energy function through

$$E = \sum_{i=1}^{n} P_{\phi_i}$$

Now we identify $\perp$ with 0 and $\top$ with 1. Then it is obvious that the penalty for each unsatisfied clause is 1, and the value of the energy function gives the number of unsatisfied clauses of the propositional formula for the given configuration (resp. truth assignment). The task is to find an assignment that minimizes the energy function to zero.

The formula $(A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee \neg E)$ for instance results in the energy function

$$E = (1 - A)(1 - B)C + ABE = ABC + ABE - BC - AC + C.$$

3.2 Energy Functions and Neural Networks

Following [8], each energy function describes a symmetric neural network. The energy function can be transformed in such a way that it contains only quadratic terms but has the same global minima as the original function. Unfortunately, this transformation enlarges the formula linearly in its size, i.e. for each term of order higher than quadraic at least one additional variable has to be introduced. In practice this is too much, because then a 100 variable problem may be transformed to a (significantly harder) 500 variable problem. The point is, that introducing additional variables also means introducing additional local minima. This is why we used symmetric networks with higher order connections and binary sigma-pi-units (which compute the sum of the products of the activations on their connections). The energy function can be mapped directly to a net with sigma-pi units without introducing new variables:

Each variable is represented by one unit in the network. The variable assignments are represented through the activation of the units (0 or 1). The coefficient w of a term $w \prod_i X_i$ in E gives the negative weight of the (higher order) connection between the Units corresponding to the involved variables X_i. Linear terms describe the threshold of the respective unit (which is 0 otherwise).

Suppose we are given the energy function of the last section

$$E = C - BC - AC + ABC + ABE$$

The corresponding network is shown in figure 1. If we flip variable A only the terms containing A are changed in the energy function. Therefore we will write

$$E = -AC + ABC + ABE + Rest$$

If α is the current configuration and β the succesor configuration then $\Delta E = E(\beta) - E(\alpha)$. We compute ΔE for a flip of variable A:

1. Case (A flips from 0 to 1)

$$\begin{array}{ll} E(\alpha) = Rest & \text{, as A} = 0 \text{ in } \alpha \\ E(\beta) = -C + BC + BE + Rest & \text{, as A} = 1 \text{ in } \beta \end{array}$$

$$\Delta E(A) = -C + BC + BE$$

2. Case (A flips from 1 to 0)

$$\begin{array}{l} E(\alpha) = -C + BC + BE + Rest \\ E(\beta) = Rest \end{array}$$

$$\Delta E(A) = Rest - (-C + BC + BE + Rest) = C - BC - BE$$

Generally, $\Delta E(A) = (C - BC - BE)(2A - 1)$. We see, that the first multiplicand gives the input for unit A in the network. In general for a flip of unit X we have

$$\Delta E(X) = (\text{netinput of } X_i) * (2X_i - 1).$$

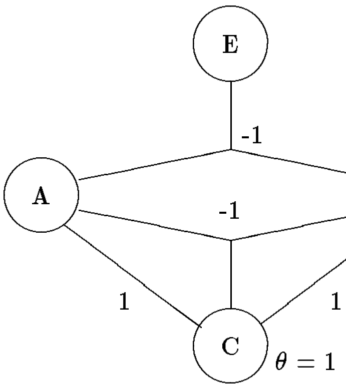

Figure 1: The network corresponding to the energy function $C - BC - AC + ABC + ABE$.

So ΔE can be computed very easily within the network, especially without consideration of variables not involved in the energy change (which reduces the effort significantly from linear to constant).

3.3 The SA Algorithm

Using the described neural network methods we can formulate the algorithm as follows:

At the beginning of the procdure the units are randomly initialized with 0 or 1. For any unit X we get the following unit function

$$X_{next} = \begin{cases} 1 - X_{current} & \text{, if } rnd(0,1) \leq P_T(\Delta E) \\ X_{current} & \text{, else} \end{cases}$$

where rnd(0,1) is a uniformly distributed stochastic variable with real values between 0 and 1. Using this unit function we perform Simulated Annealing. In the course of the procedure the temperature parameter T is decreased but it must not become zero as this would result in a divison by zero in the probability distribution. So we have to stop decreasing T when reaching T_{min}. At this time the procedure has not necessarily already exploited a local minimum. Therefore we use the following unit function after reaching T_{min}:

$$X_{next} = \begin{cases} 1 - X_{current} & \text{, if } \Delta E < 0 \\ X_{current} & \text{, else} \end{cases}$$

This unit function is used until a fixed point is reached. It is fully deterministic and behaves similar to gradient descent with continuous valued functions. At a fixed point no flip of any unit gives a decrease in the energy function and so no more flips are performed – the network is in a stable state.

```
Procedure NNSAT
    Build Net
    For i := 1 to MAXTRIES do
        initialize units with random values 0 or 1
        T := MAXTEMP (* T is the Temperature *)
        t := 0 (* t is the time *)
        While (T ≥ MINTEMP) and (t ≤ MAXCYCLES) and (E ≠ 0) do
            Step (* described below *)
            t := t + 1
            decrease temperature according to annealing schedule
        Wend
        While (E ≠ 0) and (no fixed point reached) do
            Step
        Wend
    Forend
End of NNSAT
```

The parameter MAXTRIES is the same as in GSAT but instead of MAXFLIPS we use MAXCYCLES. The correlation between these two parameters will be discussed in a later section. E is the current energy of the net. The procedure Step is defined as:

```
Procedure Step
    For each unit do
        (* with probability 1/2 update unit *)
        If rnd(0,1) ≥ 0.5
            compute ΔE
            If T ≥ MINTEMP and t ≤ MAXCYCLES
                use SA unit function
            else
                use gradient-descent-like unit function
            Endif
        Endif
    Forend
End of Step
```

We count an evaluation of ΔE plus the application of a unit function as an update. If a unit actually changes its value it is counted as a flip.

The algorithm presented here is the one we used to do our experiments. But it is only one of many possibilities to do simulated annealing. Especially the method of giving each unit a certain probability for updating is not common at all. We might also visit all units in a predefined order as it was done for example by [14].

4 Hill-climbing Algorithms for SAT

When trying to compare different methods for solving the same problem, it often proves to be useful to look for a generalized scheme containing those specific methods as instances.

In this section, we take that approach to dicuss some hill-climbing algorithms, which take a propositional formula in conjunctive normal form (CNF) as their input and attempt to find a satisfying assignment of its variables. We present these algorithms (most of which are decribed in [4] and [12]) as instances of GenSAT, a general algorithmic frame defined by [4] and discuss their important properties.

The GenSAT-procedure can be written as:

```
procedure GenSAT(α)
  for i := 1 to maxTries do
    A := initial (α);
    for j := 1 to maxFlips do
      if A satisfies α then return A
      else
        possFlips := hillClimb (α, A);
        V := pick (possFlips);
        flip assignment of V in A;
        end if;
      end for;
    end for;
  return ''no satisfying assignment found'';
  end GenSAT;
```

where α denotes the input formula (in CNF, containing N variables), *maxFlips* and *maxTries* are natural number parameters. The function *initial* generates a truth assignment, which is used as starting point for the search, *hillClimb* computes a list variables that may be flipped, and *pick* selects the variable which will be actually flipped. We call *hillClimb* and *pick* selection-functions, because they reduce the set of possible flips by selecting some and rejecting others.

GenSAT performs local hill-climbing on the number of satisfied clauses (which is called the score), the space which is searched for a better truth assignment is restricted to the set of assignments with a Hamming-distance of 1 to the current assignment (this set consists of all assignments obtained by flipping the truth value of exactly one variable in the current assignment). Using this strategy, the algorithm generally might get stuck in local minima which are no solutions, even if satisfying assignments of α do exist. Therefore GenSAT is no decision procedure for SAT[1], but to increase the probability of existing solutions to be found, the algorithm restarts the search process a fixed number of times, we say it uses several tries to find a solution.

The parameter *maxTries* determines after how many tries GenSAT will terminate without a satisfying assignment being found, whereas *maxFlips* is the number of flips, which have to be made before the current search is aborted to start a new try. As pointed out in [4], the setting of *maxTries* is simple (increasing it will simply increase the chance of success), but *maxFlips* has to be chosen carefully to optimize the performance of the algorithm. According to [4], the optimal value of *maxFlips* appears to increase approximately as N^2.

The performance of GenSAT depends mainly on the routines *hillClimb* and *pick*. In the worst case, *hillClimb* has to take into consideration the whole local neighbourhood of the current assignment, the size of which is given by the number of variables N. Now it may happen, that all N possible flips are found to be "equally good" and therefore included in *possFlips*, so *pick* has to chose again among N flips. Therefore GenSAT's time-complexity depends strongly on four factors: the average number of flips checked by *hillClimb* (these are called tentative flips), the costs of deciding wether a flip is to be included

[1] to be precise, this holds for general GenSAT, although there are instances of the algorithm, which are decision procedures

in *possFlips* (which are the costs of scoring the flip), the average number of flips checked by *pick* and the costs of selecting one flip of *possFlips*.

For GenSAT-variants, in which *hillClimb* scores all N possible flips, the performance can be significantly improved by using the following method of scoring (similar to the one described in [1]), which we will call "score-by-need": After *initial* has generated an assignment, it evaluates the score for each flip and memorizes it. Now *pick*, after selecting the "actual" flip, updates the memorized scores. As was pointed out by [10], only the scores of those variables have to be updated, which share clauses with the flipped one (which means, that there are clauses, in which both variables appear). This means, that the number of necessary evaluations of the scoring-function depends on the number and the length of clauses in α, but is usually small w.r.t. N. This way, *hillClimb* has to do no evaluation of scores at all, but simply uses the memorized scores.

Whereas the straight-forward method of scoring (which is to evaluate the scores of all possible flips in *hillClimb*) has a time-complexity of $O(N^2)$, score-by-need needs linear time in the worst case and, for fixed clause lengths (as given for k-SAT), constant time in the average case.

4.1 Greedy Hill-climbing: Basic GSAT

The basic GSAT algorithm, as proposed in [13], can be regarded as an instance of GenSAT which performs greedy hill-climbing: in GSAT, *initial* generates a random truth assignment, *hillClimb* returns those flips which cause the maximal increase of the score and *pick* selects randomly one of these flips. Since *hillClimb* uses a global selection criterion, the score-by-need strategy greatly improves GSAT's performance.

4.2 Indifferent Hill-climbing

ISAT, a GenSAT-variant introduced by [4], is very similar to basic GSAT, except that it's *hillClimb* is indifferent in the sense of including all flips in *possFlips*, which do not decrease the score, if there are none of those, *possFlips* contains all N possible flips. I_2SAT is almost identical to ISAT, but allows only flips for *possFlips*, that increase the score or, if there are none of those, those flips which ensure minimal decrease of the score.

According to [4], ISAT performs not significantly worse than GSAT, which leads to the conjecture, that GSAT's greediness is not essential for the effenciency of the algorithm. Our own experiments show, that the same holds for I_2SAT (see section 4.6). Note, that different from GSAT, ISAT and I_2SAT use no global information for the selection of the actual flips and therefore appear to be especially suitable for parallelization.

4.3 Variants with Memory

GenSAT can use information gathered during the whole run in two different ways: for scoring flips and for selecting the actual flip from *possFlips*. In [4], the variant HSAT is introduced, which is similar to GSAT, but in which *pick* selects always randomly one of the variables, that were flipped longest ago in the current try. H_2SAT is almost identical, but counts for each variable the number of times it had been in *possFlips* but not actually flipped (this counter is reset at the beginning of each try). *pick* selects randomly one of the variables with the maximal value of that counter. The performance of HSAT is significantly better than that of GSAT; according to our own experiments, H_2SAT seems to perform slightly better than basic GSAT and slightly worse than HSAT (see section 4.6).

Of course there are other ways of utilizing memory. Two more techniques are discussed in [10]: "Weights" and "Averaging In". The "Weights"-method is based on a scoring function with memory. To each clause an integer number, the weight, is associated. All weights are set to one at the start of a try, and at the end of an unsuccessful try, the weights of the unsatisfied clauses are increased by a constant. Here, the score is defined as the sum of the unsatisfied clauses' weights, so clauses that were often left unsatisfied at the end of previous tries become more and more important for the decision, which flips are to be made. This technique is especially useful when the input formula shows internal structure (like strong asymmetries) as propositional encodings of many problems do (e.g. graph-coloring, block-world-planning, n-queens).

"Averaging In" uses memory for *initial*, which returns the bitwise average of the best assignments found by the two previous tries.

4.4 Random Downward Moves

There are two different kinds of randomness in GenSAT. The first is random selection (of the initial assignment, of *possFlips* and of the actual flip), the second the possibility of random downward moves (a flip is called a "downward move", iff its score is negative). Whereas according to [4], random selection has no significant influence on GenSAT's performance, random downward moves have been found to cause real improvements. In [12], two strategies using the latter technique have been proposed. The random noise method uses a new parameter p, which is the probability that instead of selecting one flip of *possFlips*, *pick* returns a randomly chosen variable (the score of which may be negative, causing a downhill move). The random walk strategy works similarly, but allows only random flips, that appear in clauses, which are not satisfied by the current assignment. As [12] point out, random walk is somehow a more directed strategy than random noise. Experimental results show, that both methods improve GenSAT's performance[2], compared with each other, random walk is superior to random noise.

Obviously, random downhill moves are an effective means of escaping from local minima. This observation leads to a different algorithm for SAT-problems, which is also introduced in [12]: WSAT. WSAT can be regarded as a GenSAT-variant, in which *hillClimb* randomly selects an unsatisfied clause of α and returns the set of its variables as *possFlips*, and *pick* selects at random or according to a greedy heuristic one element of this set.

Experimental results show, that on certain problems WSAT outperforms GSAT with random walk (see [12] for details).

4.5 Deterministic Variants

As [4] point out, random selection-functions are not important for GenSAT's performance. This is shown by introducing GenSAT variants with deterministic selection-functions for *initial*, *hillClimb* and *pick* and comparing their behaviour with that of variants, which realize random choice at these points.

Not surprisingly, another property of the selection-functions turns out to be the important one: fairness. Unfair selection (meaning, that not all elements of a set are selected with the same probability) has a negative influence on the performance of the algorithm, which can be explained by the fact that in this case, certain variables are preferred independently of the problem structure. This potentially prevents flips, which are essential for the solution of a given problem.

We suppose, that the same argument may be applied to random downward moves, such that not randomness is the important feature, but again it is fairness of the selection, which really matters. We therefore propose a new GenSAT-variant detWSAT, which flips after each p standard moves (p being a new natural number parameter) a deterministically, but fairly chosen variable, which appears in an unsatisfied clause.

As [4] point out, it is possible to define the deterministic selection-functions in such a way, that the resulting GenSAT-variants become decision procedures for SAT (under the condition, that the parameters are chosen suitably). Since that kind of determinism has not necessarily a negative influence on the performance of the algorithm, these variants have the advantage of being complete and efficient (at least for many problems), so they seem to be good canditates for adequate procedures for SAT.

[2] more precisely: there are GenSAT-variants, as basic GSAT, the performance of which is improved by adding random noise or walk

4.6 Comparing some Variants: Experimental Results

To get a feeling for the performance of certain GenSAT-variants, we did some experiments with an extended version of the GSAT-program by Selman & Kautz (version 24c). Since we were particularly interested in the performance of our new GenSAT-variants I_2SAT and H_2SAT, we added two options `i` and `h2` realizing these variants. A third option `h1` implementing HSAT was also added to facilitate comparisons between HSAT and H_2SAT.

For our experiments, we used two sets of randomly generated, satisfiable 3-CNF-formulas with a clauses/variable ratio of 4.3. The first test-set consisted of 100 formulas with 200 variables, the second contained 50 300-variable formulas. We did experiments with *maxFlips*-values of $8N$ and $100N$, the *maxTries*-Parameter was always set to 10.[3]

We compared the success-rates (which is the total number of solutions found in an experiment divided by the total number of tries) and the flips/assignment-ratio, which is the total number of flips that were made during an experiment divided by the number of satisfying assignments found. The results are given in the following tables, where "exp.ID" is an internal identification of the experiments, "variant" describes which options have been selected ("w" = random walk with $p = 0.5$, others as defined above), "assign" and "F/A" denote the measured success-rate and flips/assignment-ratio.

$N = 200$, *maxFlips* $= 8N$

exp.ID	variant	assign	F/A
200-11	w+h1+i	70.5%	1472
200-12	w+h2+i	68.1%	1569

$N = 200$, *maxFlips* $= 100N$

exp.ID	variant	assign	F/A
200-31	w+i	99.9%	1636
200-21	w+h1+i	99.9%	1626
200-22	w+h2+i	99.7%	1768

$N = 300$, *maxFlips* $= 8N$

exp.ID	variant	assign	F/A
300-60	basic	23.8%	9120
300-41	h1	28.2%	7543
300-20	h2	25.0%	8582
300-40	h1+i	27.4%	7741
300-42	h2+i	24.8%	8599
300-51	w	67.0%	2503
300-52	w+i	62.4%	2688
300-22	w+h1	66.8%	2446
300-21	w+h2	68.4%	2429
300-71	w+h1+i	67.2%	2538
300-72	w+h2+i	62.8%	2784

$N = 300$, *maxFlips* $= 100N$

exp.ID	variant	assign	F/A
300-80	w	100%	2462
300-81	w+i	99.8%	2586
300-31	w+h1	100%	2456
300-30	w+h2	100%	2621
300-50	w+h1+i	100%	2610
300-32	w+h2+i	100%	2455

These results show quite clearly, that the most significant improvement is achieved by the "walk"-option, which implements the random-walk-strategy described in section 4.4. We also observed a slight advantage of the variants using memory (`h1` and `h2`) over basic GSAT, which confirms the results of [4]. This advantage could not be observed in combination with random walk. We also found, that there was no significant difference in performance between the `h1`- and `h2`-variants.

Furtheron our results show, that using indifferent instead of gready hillclimbing does not significantly decrease the performance (as was also stated by [4]), although we observed it to cause small decreases in performance when random walk was enabled.

[3] For a further discussion of the parameter-settings and the formulas used for these experiments, see section 5.

As the number of our experiments and the size of the test-sets were rather small, these results should be interpreted *cum grano salis*; many minor differences between the measured performance of the variants are certainly due to statistical effects, but we believe the general observations described above to be quite correct.

4.7 Simulated Annealing from a GenSAT point of view

Seen from a GenSAT point of view, we find, that many features of simulated annealing (SA) algorithms can be easily embedded in the general GenSAT-frame.

SA algorithms typically use only local information to select the next flip; in this respect, they behave similar to indifferent GenSAT variants. Randomness plays an important role in simulated annealing, usually probabilistic methods as the Glauber- or the Metropolis-Test are used to decide, wether a given variable is actually to be flipped or not. The probability commonly depends on two factors: the temperature parameter and the score. Of course, this kind of randomness can be easily achieved in GenSAT; one only has to chose a suitable selection-function for *hillClimb*.

Note, that during one execution of GenSAT's inner loop exactly one variable is flipped, whereas annealing-algorithms typically flip a random number of variables per step[4]. This discrepancy can be avoided, if GenSAT's *hillClimb* is designed in such a way, that it generates only *possFlips*-sets of cardinality 1, giving *pick* only one possible flip to select. If we now take care, that *hillClimb* checks the variables in a fair way (ensuring that given a large number of executions of the inner loop, all variables are checked with the same probability), we achieve a behaviour, which is very similar to that of certain simulated annealing algorithms.

To modify the temperature-parameter during the run, SA algorithms typically use an annealing schedule. This can also be realized by GenSAT variants: again, *hillClimb* has to be defined in a suitable way, using GenSAT's loop-variables i and j as additional parameters.

To show, how a GenSAT-Variant realizing simulated annealing would look like, we introduce SAnSAT, which is defined by the GenSAT-frame with the following procedures:

```
procedure initial(α)
  A := rndAssignment (α); % (a randomly chosen assignment)
  return A;
  end initial;

procedure hillClimb(α, A)
  P := rndPermutation (0, N − 1); % (a randomly chosen permutation of 0, ..., N − 1)
  k := 0;
  repeat
    T := temperature (i, j, k); % (implements annealing schedule)
    k := k + 1;
    x := score (A, P(k mod N) ); % (score of variable P(k mod N) )
    accept := glauber (x, T); % ( = true with probability 1/(1 + e^(-x/T)) )
  until accept; % (variable P(k mod N) was accepted)
  return {P(k mod N)}; % (returns a set containing only the selected variable)
  end hillClimb;

procedure pick(possFlips)
  return rndElement (possFlips); % (returns a randomly selected element of possFlips)
  end pick;
```

[4] here, a step is a period, during which all variables are updated exactly once

5 Experimental Results

We compared GSAT and Simulated Annealing (SA) on random problems with 100, 200, and 300 variables with a ratio of 4.3 for clauses/variables. This characterizes a class of very hard problems for the Davis-Putnam-Procedure, as these problems seem to be neither under- nor over-constrained. We used GSAT with a random walk probability of -0.5 because [12] reported that this parameter setting performs best. It means, that random walk occurs approximately every second flip and is performed even if there exist flips decreasing the score. We set MAXCYCLES $= 1000$ for SA and computed an appropriate setting for GSAT determining the maximal number of flips allowed. In one cycle of SA, about half of the variables are updated ($N/2$). By doing the experiments we found an almost constant ratio of updates per flip of about 5.5. This means, that per cycle about $N/(2*5.5) = N/11$ flips are performed. Allowing 1000 cycles then means that up to $1000*N/11 \approx 100*N$ flips are possible. We therefore used a varying MAXCYCLES parameter of $100*N$ for GSAT.

We ran both algorithms on the same 100 instances of the 100 and 200 variable formulas and the same 50 instances of the 300 variable problem. The formulas were generated with the random formula generator obtained from Bart Selman, and they were constructed such that they were known to be satisfiable. One should note that this may lead to easier problem instances than filtering out unsatisfiable formulas after generation[5]. But nevertheless the performance of both algorithms remains comparable. This just explains why our results for GSAT are much better than those reported in [12]. Initial assignments were created with the same random generator for both, GSAT and SA. For the SA algorithm we used the neural network simulator ICSIM written in the object oriented language SATHER which was developed at ICSI [9].

For Simulated Annealing, we compared the performance of several annealing schedules for the 100 variable problem to determine which schedule to use. The problem is to find a "good" approximation of the infinitesimally slow annealing required for the theoretical proof that SA finds a global minimum. The annealing schedule usually depends on the number of iterations t through the algorithm and a certain maximal difference ΔT between two values of the temperature. It is necessary to find a compromise between very slow annealing (which improves the probability that a global minimum will be found), and keeping the time for the search short.

The different schedules are as follows:

$$\begin{aligned}
\text{Schedule 1}: &\quad T(t+1) = T(t) - \Delta T \\
\text{Schedule 2}: &\quad T(t+1) = T(t) - \Delta T/t \\
\text{Schedule 3}: &\quad T(t+1) = \text{MAXTEMP}/(t+1) \\
\text{Schedule 4}: &\quad T(t+1) = \text{MAXTEMP}/\exp(t*\Delta T) \\
\text{Schedule 5}: &\quad T(t+1) = T(t) - \Delta T/\exp(t)
\end{aligned}$$

Together with the Glauber distribution we then obtained the following results for the five schedules which are also given in Table 1.

1. The first schedule was not too bad compared with the second one. Obviously the value $\Delta T = 0.0005$ caused an annealing that was neither too slow nor too fast.

2. The second schedule scaled up very well for a parameter setting of MAXTEMP $= 0.3$, MINTEMP $= 0.01$, and $\Delta T = 0.01$. It found an assignment in *each* try for the 100 variable problem. Other parameter settings, esp. a higher initial temperature or ΔT did not scale up as well.

3. The third schedule showed a rather poor performance. We tried it with MAXTEMP $= 5$, 1, and 0.2, resp. For all initial temperatures it found less assignments than schedule 2, with a higher amount of updates and flips per assignment. It seems that in these schedules on the one hand the decrease of the temperature was too fast, on the other hand beginning with a relatively high temperature resulted in wasted time for a search in the beginning, where the energy still is rather high.

[5] Bart Selman, personal communication

Schedule	%	U/F	U/A	F/A
1, ΔT=0.0005	99	9.34	4117	441
2	100	9.04	3736	413
3, MAXTEMP=0.2	92	13.44	9767	727
3, MAXTEMP=1.0	69	10.87	5029	463
3, MAXTEMP=5.0	94	9.3	5680	610
4	100	7.27	3657	503
5	41	9.1	3877	426

Table 1: Testing the performance of different annealing schedules with Glauber distribution. If not mentioned otherwise, MAXTEMP=0.3, MINTEMP=0.01 and $\Delta T = 0.01$. For schedule 3 we used MINTEMP=0.0.

4. Schedule 4 is a so-called geometric schedule. It corresponds to the one Spears [14] used. We tried it also with MAXTEMP $= 0.3$, MINTEMP $= 0.01$, and $\Delta T = 0.01$ as suggested by [14]. This schedule also scaled up very well it showed approximately the same behavior as schedule 2, with a slightly higher amount of flips.

5. Schedule 5 performed very poorly as here the decrease of temperature was too slow. The number of updates per assignment was less than 4000 but it found an assignment in only 41% of the tries, because it reached the limit of iterations through the network and therefore was forced to do gradient descent in the end.

Finally we chose schedule 2 with the parameters stated above. In our experiments, we additionally used a minimal temperature MINTEMP and stopped the annealing if the temperature reached this value. Beyond this point, gradient descent was performed until the nearest local minimum was reached (which is equivalent to setting the temperature to 0). The distribution for the choice of the flipping variable was a combination of the so-called Metropolis distribution [7] and gradient descent, if the minimal temperature MINTEMP or the maximal number of cycles MAXCYCLES was reached. During the experiments, we also tested the performance of SA with the Glauber distribution [5], where even the probability for *improving* flips depends on the score (i.e. a more greedy choice of the variables), but we found the performance of Metropolis to be superior. It found as much assignments with a significantly lower amount of updates and also less flips. This supports the experimental results of [4] which suggest that greediness is not an important feature for the performance of GSAT-like algorithms. But as this distribution accepts all improving flips, in our experiments the number of flips was about twice the number of flips GSAT used. On the other hand the number of updates was only about half the number of updates GSAT had to perform. This may clarify the results of [12], that GSAT outperformed their version of SA, if only comparing the amount of flips. Our overall probability distribution then was

$$P_T(\Delta E) = \begin{cases} 1 & \Delta E < 0 \\ \exp(-\frac{\Delta E}{T}) & \Delta E \geq 0 \,\&\, T \geq \text{MINTEMP} \,\&\, t \leq \text{MAXCYCLES} \\ 0 & \text{otherwise} \end{cases}$$

This probability distribution means, that variables with negative score, i.e. whose flip will *decrease* the energy, are *always* flipped if they are selected, whereas variables increasing the energy are flipped only with a probability depending on the amount of increase they cause (or never in the third case). This kind of choice is indifferent towards the extent of improvement a flip causes, in contrast to GSAT which greedily chooses always a variable with least score, if no random walk is performed. Above all, our SA algorithm also contains two methods to escape from local minima: random upwards moves, which is a feature inherent to all SA algorithms, and restarting the search after a given time. Restarting the search is a feature not usually used within SA. But as explained above, in practice SA remains incomplete as the infinitesimally slow annealing can be only approximated. This means that SA also may get stuck in a local minimum. Consequently, we included restarts in our SA algorithm.

Our SA algorithm also does not contain refinements to reduce the number of ΔEs to be recomputed like GSAT, as our choice which variable to flip only depends on local information. There is no global decision which variable to flip like in GSAT, and consequently it is not necessary to update all variables after a flip is made. But nevertheless the "trick" of updating the ΔEs instead of recomputing them could also be used within the SA algorithm.

Overall, our SA algorithm is not implemented very efficiently. So we did not compare the CPU–time needed to solve a problem (it was much higher for SA). We rather compared the amount of updates and flips necessary to find an assignment with SA and with GSAT. We also compared the success rate, i.e the percentage of tries where an assignment was found, averaged over 10 tries per problem and 100 formulas for the 100 and 200 variable problems and 50 formulas for the 300 variable problems.

We let GSAT and SA do 10 tries per problem and find as much assignments as possible, furthermore the number of flips in SA was only bound by the number of iterations through the network, according to the annealing schedule. The maximal number of iterations through the network was set to 1000 per try and in one iteration each unit has a chance of 50% of being updated. After reaching this limit or MINTEMP without having found an assignment, the algorithm performed gradient descent until the nearest local minimum was reached. Interestingly, if gradient descend had to be performed (because MINTEMP or MAXCYCLES was reached) a solution was almost never found.

This way of comparison is different from the one in [14], as they count only the tries until the first assignment is found. [12] used this way of comparison only for the SA algorithm. Our way of comparison avoids too much influence of a certain initial assignment. By averaging the amount of work to find an assignment over 10 tries, we lower the possibility that an actually very difficult problem (even the problems with ratio 4.3 seem to vary very much in their difficulty) is solved very quickly only because the initial assignment was near to the solution, and vice versa for the easier problems.

Then we computed the number of updates and flips performed per assignment, averaged over 100 problems. We found a much lower ratio of updates/flips for simulated annealing than the one reported for GSAT (see Table 2). This becomes clear if one considers the differences in the algorithms for GSAT and SA: Whereas GSAT does all necessary updates after a variable has flipped and before another variable is chosen to be flipped, SA randomly chooses a variable after a flip, updates it and, depending on the resulting energy change, possibly flips this variable immediately. This shows a problem if we want to compare the amount of work performed by both algorithms to find an assignment in the following way: At first thought, it seems to be correct to take the updates into account to compute the amount of work, as [14] argued. But as GSAT uses several algorithmic features reducing the time complexity of an update significantly, the comparison becomes difficult. First of all, GSAT uses some data structure which prevents it from doing unnecessary updates. If a variable is flipped, not all other variables have to be updated, but only those variables, which share a clause with the previously flipped variable. As [14] explains, in a formula with fixed clause length L and a ratio of variables to clauses of C each variable shares clauses with about $L * (L - 1) * C$ other variables. So for 3SAT–problems with a ratio of $C = 4.3$ GSAT consequently computes about $3 * 2 * 4.3 = 25.8$ new ΔEs for each flip, which is independent of the problem size, instead of recomputing the ΔEs for all variables. As we use a neural network for the SA algorithm, this feature is inherent to our algorithm: Each unit of the network is only connected to those other units which represent variables sharing a clause with the variable represented by this unit. So it is easy to avoid updating units which are not connected to a flipped unit and vice versa each units considers only connected units if it has to be updated.

But GSAT also uses the same programming "trick" as the parallel SA algorithm of [1]. This trick profits from the fact that the energy change for the flip of a certain variable need not be really *recomputed* after each flip performed but can be updated in a certain predefined way. Consequently the update of all the ΔEs can be done in linear time instead of quadratic time[6]. This feature also can be easily used within the SA algorithm. But here lies the problem in comparing the amount of updates performed by GSAT and SA: GSAT does about $n = 25.8$ updates per flip and each of those updates can be done in constant time c_1. So every flip causes $n * c_1$ steps of work. During the SA algorithm, between two flips about 5.5

[6] As [1] point out, this trick speeds up the algorithm by about 100 times. It is not clear why they report a constant speed up, it should be linear in the number of variables of the problem.

updates were performed (in the metropolis case). But each of these updates means much more work: We have c_1 times the number of flips affecting the unit's ΔE since its last update was made. This is exactly the same as if we had counted c_1 for each flip affecting the ΔE, or, as if we had updated the unit after each flip affecting it. In the worst case a unit is updated after all the units it is connected with have flipped once (a second flip of a unit will remove its effect on other ΔEs). So, as a unit is connected to about $n = 25.8$ other units (due to the considerations above) its update needs at worst $n * c_1$ steps. So we also could have multiplied the number of flips c of SA by $n = 25.8$ to determine the amount of work done with the updates. This means that in the worst case we have the same amount of work per flip as in the GSAT algorithm. But it may be less as those units which were not flipped again before a minimum was reached, the last update was unnecessary. Due to this considerations we could rely on comparing the average number of flips performed by both algorithms to find an assignment, which may be a little bit dis-advantageous for SA.

In order to do a better comparison, we have to compute the average number of steps, an update in SA needs, given that one update in GSAT needs c_1 steps. Be $f = F/A$ the number of flips per assignment and $u = U/A$ the number of updates per assignment, V the number of variables (units) in the problem. Then each unit is updated about u/V times per assignment and flipped f/V times. The total number of variables affecting a single other unit is about n due to the above considerations. So the total number of flips affecting a unit is $f/V * n$ per assignment. Consequently for each update of a unit we must consider about $\frac{f/V*n}{u/V} = f/u * n$ flips of connected units. This means that each update needs about $1/5.5 * 25.8 * c_1 = 4.7 * c_1$ steps. To compare GSAT with SA we therefore should multiply the number updates of SA by 4.7 or, very simply, compare the number of flips instead of updates, as in any case we end up with an amount of work of 25.8 steps per flip!

Comparing the results of both algorithms, we find that both show about the same success rate (number of assignments found per try). SA needs some more flips, and the ratio of flips between both algorithms seems to increase with the number of variables of the problem. So indeed GSAT with random walk scales up better. But nevertheless one has to keep in mind that the GSAT version is already a very improved one. A comparison with GSAT with *random noise* [12] (which is comparable to the randomness used in SA) shows, that this scales up better that basic GSAT (without any random upwards moves) but is still significantly worse than GSAT with random walk. So it might be possible, that indeed a SA variant using random walk would outperform GSAT. [14] already did experiments with such a SA variant, but unfortunately we do not know how the problems he used were generated, i.e. if his results are comparable with ours. Nevertheless if comparing his results for SA and SA with random walk, we find that SA with random walk shows about the same success rates as simple SA but needs much less flips per assignment. Nevertheless, the ratio between the numbers of flips of both variants decreases with the number of variables of the problem, although staying below 1. On the other hand in our experiments the ratio between the numbers of flips of ordinary SA and GSAT with random walk *increase* with the number of variables. So it is likely that, even when including random walk in both approaches, SA will stay slightly worse than GSAT. In addition, when using a neural network representation for SA, the inclusion of random walk is rather difficult, because the clause structure of the underlying formula gets lost. One would have to provide a special data structure only for this purpose. On the other hand, we gain the advantage of massive parallelism by using neural networks.

6 Parallelization of GSAT and SA

The question of parallelizability of both algorithms is interesting for at least two reasons. First of all, in the future presumably the most powerful computers will be massively parallel ones. Second, problems occurring in practice will not be always those lying in the "hard" area. As the satisfiability problem is $\mathcal{NP}$-complete, one cannot expect to gain anything by parallelization, if the problem considered really is one of the hard ones. But if it is an easier one, it might be possible, that the performance of parallel algorithms significantly improves above sequential ones (e.g. if we touch a subclass of problems lying in the class $\mathcal{NC}$ of efficiently parallelizable problems). Experiments in this direction still have to be done, but this is why we should take the parallelizability of these algorithms into account.

	GSAT, random walk, MAXFLIPS =				
	$V * 8$			$V * 100$	
V	Assigns	U/A	F/A	Assigns	F/A
100	82.6%	12,158	471	100%	520
200	70.8%	37,415	1450	99%	1622
300	68.4%	61,917	2400	99.8%	2394

	SA, Glauber				SA, Metropolis			
Vars	Assigns	U/A	F/A	U/F	Assigns	U/A	F/A	U/F
100	100%	7,357	690	10.66	99.6%	3,540	642	5.51
200	95.6%	27,674	2601	10.64	99.1%	14,119	2547	5.54
300	95.0%	53,325	5069	10.52	99.6%	26,763	4889	5.47

Table 2: Comparison of GSAT with random walk, our results and the results of [12] and SA with schedule 2, Glauber and Metropolis distribution, in the number of flips (F), updates (U) and assignments (A) found.

The parallelization of SA is straight forward, if a neural network representation is used. As each unit in the network (resp. each variable) decides if it is updated or flipped only by means of *local* information, updates can be performed simultaneously in parallel and flips can be performed at least asynchronously. Another possibility would be to use a parallel SA algorithm like the one presented in [1]. Here multiple (locally) possible flips are summarized to a "multi-flip", which is accepted by means of a global decision considering the global score of a synchronous flip of all accepted variables, e.g. by means of the metropolis test. If at each time only a (always changing) portion of the variables is chosen, the problem of getting trapped in a cycle of recurring states is avoided. This algorithm seems to scale up significantly better than general SA [1]. Unfortunately, up to this moment no theoretical results about its convergence behavior exist, which means that it cannot be guaranteed that the algorithm converges even to a *local* minimum. Nevertheless, experimental results lead to the conclusion that the algorithm indeed shows convergent behavior and also converges to a *global* minimum in the limit.

There are also several possibilities of parallelizing GSAT. Perhaps the most obvious (and naive) idea is to evaluate several tries simultaneously using MAXTRIES processors. But this possibility excludes many variants of GSAT using a kind of memory, e.g. *averaging in* and *clause weights* as described in [11], because those use information about previous unsuccessful tries to choose a new initial assignment or change the energy surface, respectively. Clearly, variants using "history" as HSAT and H_2SAT as described in Section 4.3 are still possible. But it is also possible to parallelize scoring using N processors and straight-forward scoring or score-by-need and a number of processors independent of N. Unfortunately, as GSAT needs global information to decide which variables are elements of the `possflips` list, a very high amount of communication would be necessary, slowing down the whole procedure. This could be avoided if an indifferent variant of GSAT is used. As reported by [4], indifference does not lead to significantly worse behavior. They even conjecture that greediness is not important for the performance of GSAT. A third approach (analogous to the parallel SA algorithm introduced in [1]) would be to combine the flips of a subset of *possFlips* to a multi-flip, the execution of which is probabilistically dependent of its score (which of course can be totally different from the scores of the single flips).

7 Related work

There are two other papers dealing with a comparison of the performance of GSAT and SA on the problem of finding satisfying assignments for random 3CNF-formulas (3SAT). One is the work done by B. Selman, H. Kautz, and B. Cohen [12], the other one is by W. Spears [14]. The two comparisons are totally different as they use different SA algorithms, different variants of GSAT and different ways of

	Kautz/Selman	Spears
GSAT-Variant	random walk	basic
SA-Schedule	constant temperature	geometric
max_temp	0.2	0.3
Distribution	Metropolis	Glauber
Comparison	Flips	Updates
"Winner"	GSAT	SA

Table 3: The two comparisons of GSAT and SA

comparison. Consequently, also their results differ. An overview of the differences in both comparisons is given in Table 3. Selman, Kautz, and Cohen [12] use GSAT with random walk for the comparison, which scales up significantly better than basic GSAT, whereas Spears [14] uses only basic GSAT for the comparison. In addition, [14] uses the results of [13] which report a significantly higher amount of flips per assignment than new experiments with basic GSAT reported in [12], and he compares basic GSAT with a variant of SA performing also random walk, which scales up even better than SA itself. So the comparison of [14] is unfair in so far as the very poor basic variant of GSAT is compared with a rather "tuned" version of SA including random walk. Interestingly, also [12] report that SA performs significantly better than basic GSAT, but is worse than GSAT with random walk, but they compare only the simple amount of flips necessary to find an assignment. Instead, [14] compares also the amount of updates necessary, as they need also a certain amount of work. This may be correct, if a naive method of *computing* the new ΔEs after a flip is used. But due to the algorithmic improvements used within GSAT, the comparison of updates is still more complicated as we explained in the previous section. In our experiments comparing the simple number of updates for GSAT and SA would also have lead to the conclusion that SA scales up significantly better than GSAT.

To summarize the conclusions drawn in both papers, we may say that [12] see GSAT as the "winner" whereas [14] conjectures that SA scales up better. But at least the comparison in [14] seems to be preferential for SA. One could argue that we should compare basic GSAT with SA, as then we have two "original" un-tuned versions of both algorithms. But by comparing GSAT with random walk and SA we see that the improvements of GSAT only lead to an approach to the principles of SA.

8 Conclusion

Seen from a SA point of view, we find, that many features already inherent in SA algorithms are embedded in the refined version of the GSAT–frame we used. Seen from a GSAT standpoint, SA is nothing else than a GSAT variant. We have shown this by formulating both algorithms within the GenSAT–framework in Section 2.

SA algorithms typically use only local information to select the next flip; in this respect, they behave similar to indifferent GenSAT variants [4] which consider all variables with negative score as elements of the *possFlips* list. As [4] argue, greediness is not an important feature for the performance of GenSAT, which may be confirmed by our results showing very little difference in the performance of the greedy GSAT algorithm and the indifferent SA algorithm. Randomness plays an important role in SA, usually probabilistic methods as the Metropolis or the Glauber test [5] are used to decide, whether a given variable is actually to be flipped or not. The probability commonly depends on two factors: the temperature parameter and the score. Of course, this kind of randomness can be easily achieved in GenSAT; one only has to choose a suitable selection-function for *hillClimb*, as we have shown in Section 4.7. Note, that during one execution of GenSAT's inner loop exactly one variable is flipped, whereas annealing-algorithms typically flip a random number of variables per cycle. This discrepancy can be avoided, if GenSAT's *hillClimb* is designed in such a way, that it checks the variables in the *possFlips* list in a fair way (ensuring that given a large number of executions of the inner loop, all variables are checked with the same probability). We then achieve a behavior, which is very similar to that of certain SA algorithms. To modify the temperature-parameter during the run, SA algorithms typically use an annealing schedule. This can also be realized by GenSAT variants: again, *hillClimb* has to be defined in a suitable way, using GenSAT's loop-variable j as additional parameter.

On the other hand many improvements of GSAT to simply speed up the algorithm are already realized within SA: The avoidance of updating the score of variables which are not affected by the previous flips is inherent to neural networks because of their connectionist structure. The possibility of random upwards moves on the energy surface is one of the main features of SA, introduced later on to GSAT to greatly improve its performance.

Above all, there are many features included into GSAT to improve its performance by the time as experience with this algorithm grew up. Most of them are already inherent to each SA algorithm in general, especially if a neural network representation is used. Some other improvements may still be included into SA. So it seems worth to investigate SA further and try to find further improvements as here already the basic algorithm is very powerful compared to the basic GSAT variants (as shown in [14]). We have shown that the principles necessary to improve GSAT lead to an algorithm very similar to the well known SA algorithm, which is not adapted to special problems classes in any way. It therefore will be interesting to further investigate how powerful SA still can be made. In addition, for SA there exists a theoretical convergence result. We do not know about a similar result for GSAT.

As explained in section 6, the question of parallelizability is also very important. As SA is much better suited for parallelization than GSAT, we should investigate further the possibilities of SA. There still remain some very interesting questions to be answered: Can we provide a convergence results for the parallel SA algorithm introduced in [1]? What would be an optimal way to parallelize GSAT? What about the behavior of parallel SA vs. parallel GSAT? We anticipate that SA would scale up better, but experiments still have to be done. How do both approaches behave, if we do *not* consider the hardest problems, but those that are under- or over-constraint? What can we do, if we really do not know if a given formula is satisfiable or not?

References

[1] N. Boissin and J.-L. Lutton. A parallel simulated annealing algorithm. *Parallel Computing*, 19:859–872, 1993.

[2] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[3] S. Geman and D. Geman. Stochastic relaxation, Gibbs distribution, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[4] I. P. Gent and T. Walsh. Towards an understanding of hill–climbing procedures for SAT. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 28–33. MIT press, 1993.

[5] R. J. Glauber. Time-dependent statistics of the ising model. *Journal of mathematical physics*, 4:294–307, 1963.

[6] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. Technical Report Research Report RC 9335, IBM Thomas j. Watson Center, Yorktown Heights, N.Y., 1982.

[7] W. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of states calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[8] G. Pinkas. Symmetric neural networks and logic satisfiability. *Neural Computation*, 3, 1991.

[9] H. W. Schmidt. ICSIM: Initial design of an object-oriented net simulator. Technical Report TR-90-055, International Computer Science Institute, 1990.

[10] B. Selman and H. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 46–51. MIT press, 1993.

[11] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In R. Bajcsy, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 290–295. Morgan Kaufmann Publishers Inc., 1993.

[12] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiabiliy testing. Technical report, AT&T Bell Laboratories, 1993.

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

[14] W. M. Spears. Simulated annealing for hard satisfiability problems. Technical report, Naval Research Laboratory, Washington D.C., 1993.