```
$\underline{\varphi_1 \equiv \varphi_2}$ : for $any\ \ k \in \{1,2\}$ : $A_r$ $occurs\ in$ $\varphi_k$ do
              $UpdateScores^{\pm}(\varphi_k, A_r)$
           for $any\ \ A_i$: $A_i$ $occurs\ in$ $\varphi$ do
              $\varphi.s_i := \varphi_1.s_i \cdot \varphi_2.s_i^- + \varphi_1.s_i^- \cdot \varphi_2.s_i$      ; where $\varphi_k.s_i^{\pm}$ means $\begin{cases} \varphi_k.s^{\pm}[i]\ if\ A_i \in \varphi_k \\ \varphi_k.s^{\pm}\ otherwise \end{cases}$
              $\varphi.s_i^- := (\varphi_1.s_i^- + \varphi_2.s_i) \cdot (\varphi_1.s_i + \varphi_2.s_i^-)$
end case.
```

```
procedure $UpdateScores^{\pm}(\varphi, A_r)$
$\varphi.s := \varphi.s[r]$
$\varphi.s^{-} := \varphi.s^{-}[r]$
Case $\varphi$ of :
Literal :   $\varphi.s[r] := complement(\varphi.s[r])$
            $\varphi.s^{-}[r] := complement(\varphi.s^{-}[r])$
$\bigwedge_k \varphi_k$ :   for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
                $diff_k := \varphi_k.s[r] - \varphi_k.s$
                $ratio_k := \varphi_k.s^{-}[r]/\varphi_k.s^{-}$
            $diffs := \sum_k^{A_r \in \varphi_k} diff_k$
            $ratios := \prod_k^{A_r \in \varphi_k} ratio_k$
            for any $A_i$: $A_i$ occurs in $\varphi$ do
                $\varphi.s[i] := \varphi.s[i] + diffs$
                $\varphi.s^{-}[i] := \varphi.s^{-}[i] \cdot ratios$
            for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
                for any $A_i$: $A_i$ occurs in $\varphi_k$ do
                    $\varphi.s[i] := \varphi.s[i] - \varphi_k.s[i]$
                    $\varphi.s^{-}[i] := \varphi.s^{-}[i]/\varphi_k.s^{-}[i]$
                $UpdateScores^{\pm}(\varphi_k, A_r)$
                for any $A_i$: $A_i$ occurs in $\varphi_k$ do
                    $\varphi.s[i] := \varphi.s[i] + \varphi_k.s[i] - diff_k$
                    $\varphi.s^{-}[i] := \varphi.s^{-}[i] \cdot \varphi_k.s^{-}[i]/ratio_k$
$\bigvee_k \varphi_k$ :   for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
                $ratio_k := \varphi_k.s[r]/\varphi_k.s$
                $diff_k := \varphi_k.s^{-}[r] - \varphi_k.s^{-}$
            $ratios := \sum_k^{A_r \in \varphi_k} ratio_k$
            $diffs := \prod_k^{A_r \in \varphi_k} diff_k$
            for any $A_i$: $A_i$ occurs in $\varphi$ do
                $\varphi.s[i] := \varphi.s[i] \cdot ratios$
                $\varphi.s^{-}[i] := \varphi.s^{-}[i] + diffs$
            for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
                for any $A_i$: $A_i$ occurs in $\varphi_k$ do
                    $\varphi.s[i] := \varphi.s[i]/\varphi_k.s[i]$
                    $\varphi.s^{-}[i] := \varphi.s^{-}[i] - \varphi_k.s^{-}[i]$
                $UpdateScores^{\pm}(\varphi_k, A_r)$
                for any $A_i$: $A_i$ occurs in $\varphi_k$ do
                    $\varphi.s[i] := \varphi.s[i] \cdot \varphi_k.s[i]/ratio_k$
                    $\varphi.s^{-}[i] := \varphi.s^{-}[i] + \varphi_k.s^{-}[i] - diff_k$
```

```
procedure UpdateScores($\varphi, A_r$)
$\varphi.s := \varphi.s[r]$
Case $\varphi$ of :
    Literal :
        $\varphi.s[r] :=$ complement($\varphi.s[r]$)          ; $\varphi = A_r$ or $\varphi = \neg A_r$
    $\bigwedge_k \varphi_k$ :
        for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
            $diff_k := \varphi_k.s[r] - \varphi_k.s$          ; $\varphi_k.s_{new} - \varphi_k.s_{old}$
        $diffs := \sum_k^{A_r \in \varphi_k} diff_k$
        for any $A_i$: $A_i$ occurs in $\varphi$ do           ; main loops:
            $\varphi.s[i] := \varphi.s[i] + diffs$            ; equivalent but much faster than:
        for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do    ; for any $A_i$: $A_i$ occurs in $\varphi$ do
            for any $A_i$: $A_i$ occurs in $\varphi_k$ do      ;   for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
                $\varphi.s[i] := \varphi.s[i] - \varphi_k.s[i]$  ;      $\varphi.s[i]_{new} :=$
            UpdateScores($\varphi_k, A_r$)                   ;      $\varphi.s[i]_{old} + \varphi_k.s[i]_{new} - \varphi_k.s[i]_{old}$
            for any $A_i$: $A_i$ occurs in $\varphi_k$ do      ;
                $\varphi.s[i] := \varphi.s[i] + \varphi_k.s[i] - diff_k$   ;
    $\bigvee_k \varphi_k$ :
        for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
            $ratio_k := \varphi_k.s[r] / \varphi_k.s$         ; $\varphi_k.s_{new} / \varphi_k.s_{old}$
        $ratios := \prod_k^{A_r \in \varphi_k} ratio_k$
        for any $A_i$: $A_i$ occurs in $\varphi$ do           ; main loops:
            $\varphi.s[i] := \varphi.s[i] \cdot ratios$       ; equivalent but much faster than:
        for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do    ; for any $A_i$: $A_i$ occurs in $\varphi$ do
            for any $A_i$: $A_i$ occurs in $\varphi_k$ do      ;   for any $\varphi_k$: $A_r$ occurs in $\varphi_k$ do
                $\varphi.s[i] := \varphi.s[i] / \varphi_k.s[i]$  ;      $\varphi.s[i]_{new} :=$
            UpdateScores($\varphi_k, A_r$)                   ;      $\varphi.s[i]_{old} \cdot \varphi_k.s[i]_{new} / \varphi_k.s[i]_{old}$
            for any $A_i$: $A_i$ occurs in $\varphi_k$ do      ;
                $\varphi.s[i] := \varphi.s[i] \cdot \varphi_k.s[i] / ratio_k$   ;
    $\varphi_1 \equiv \varphi_2$ :
        for any $k \in \{1, 2\}$ : $A_r$ occurs in $\varphi_k$ do
            UpdateScores$^{\pm}$($\varphi_k, A_r$)
        for any $A_i$: $A_i$ occurs in $\varphi$ do
            $\varphi.s_i := \varphi_1.s_i \cdot \varphi_2.s_i^- + \varphi_1.s_i^- \cdot \varphi_2.s_i$   ; where $\varphi_k.s_i^{\pm}$ means $\begin{cases} \varphi_k.s^{\pm}[i] \ if \ A_i \in \varphi_k \\ \varphi_k.s^{\pm} \ otherwise \end{cases}$
end case.
```

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman and Company, New York, 1979.

[GW92] I. P. Gent and T. Walsh. The Enigma of SAT Hill-climbing Procedures. Technical Report 605, University of Edinburgh, Dept. of Artificial Intelligence, 1992.

[GW93] I. P. Gent and T. Walsh. Towards an Understanding of Hill-climbing Procedures for SAT. In *Proc. of the 11th National Conference on Artificial Intelligence*, 1993.

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and Easy Distributions of SAT Problems. In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 459–465, 1992.

[PG86] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[SK93] B. Selman and H. Kautz. Domain-Independent Extension to GSAT: Solving Large Structured Satisfiability Problems. In *Proc. of the 13th International Joint Conference on Artificial Intelligence*, pages 290–295, 1993.

[SLM92] B. Selman, H. Levesque., and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proc. of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

A The updating algorithm

Let A_r be the currently flipped variable. The computation of *UpdateScores*(φ, A_r) calls *UpdateScores*$^{\pm}(\varphi_k, A_r)$ iff φ_k occurs in φ under the scope of some $\equiv$, it calls *UpdateScores*(φ_k, A_r) otherwise. *UpdateScores*$^{\pm}(\varphi, A_r)$ is quite similar to *UpdateScores*(φ, A_r). It performs also the updating of $s^{-}(T, \varphi_k)$ values, which is dual to the updating of $s(T, \varphi_k)$. Both functions are invoked only if A_r occurs in φ.

In order to handle "$x/0$" situation we must represent scores as pairs (n, s) such that $score = 0^n \cdot s$, where $s \neq 0$ and $0^0 := 1$. Thus $(n, s)/(m, t) = (n - m, s/t)$. By this way $(s_i \cdot s_j)/s_j = s_i$ even if $s_j = 0$.

clausal formulas. In fact, if Ψ is the result of applying one of such algorithms to a non-clausal formula φ, then the number of clauses of Ψ is $O(|\varphi|)$ and $|\Psi|$ is $O(|\varphi|^2)$ [dlT90]. Nevertheless this result is valid only if no "$\equiv$" occurs in φ, as the number of clauses of Ψ grows exponentially with the number of "$\equiv$" in φ. More important, the introduction of new variables much enlarges the search space, as the "assignment searching" algorithms like GSAT are worst-case exponential in the number N of the variables of the input. Finally, Ψ is not logically equivalent to the inputs φ. In fact Ψ implies φ, but the converse is not true: we only know that Ψ is satisfiable iff φ is satisfiable [PG86]. For any truth assignment T' for Ψ, we have that $T' \models \Psi$ implies $T' \models \varphi$ but not vice versa, so that the assignments satisfying Ψ are a (generally very strict) subset of the possible extensions to the new variables of the assignments satisfying φ.

The second observation is that the solution proposed in this paper can be applied to the variants of GSAT which do not need a direct access to all single clauses. [GW92, GW93] propose many possible variants to GSAT. In "CSAT" (Cautious SAT) *hill-climb* returns all the variables which cause a decrease of the score. This can be implemented by a simplified version of *NC-hill-climb()*. In "DSAT" (Deterministic SAT) the function *pick()* performs a deterministic choice. In "RSAT" (Random walk SAT) the variable is picked randomly among *all* the variables. Finally, in "MSAT" (Memory SAT) *pick()* remembers the last flipped variable and avoids picking it. All these variants can be transposed into NC-GSAT as well, as they are independent of the structure of the input formula. [SK93] suggests some variants to the original GSAT algorithm which improve the performance and/or overcome some problems like, for instance, escaping local minima. The strategy "*Averaging in*" suggests a different implementation of the function *initial()*: instead of a random assignment, *initial()* returns the bitwise average of the best assignments of the two latest cicles. This is independent of the form of the input formula. In the strategy "*random walk*" the sequence *hill-climb() - pick()* is substituted with probability p by a simpler choice function: choose randomly a variable occurring in some unsatisfied clause. This idea can be transposed in NC-GSAT: choose randomly a branch passing only for nodes whose score is different from zero, and pick the variable at the leaf.

References

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *3rd Annual ACM Symposium on the Theory of Computation*, pages 151–158, 1971.

[dlT90] T. Boy de la Tour. Minimizing the Number of Clauses by Renaming. In *Proc. of the 10th Conference on Automated Deduction*, pages 558–572. Springer-Verlag, 1990.

Proposition 4.1 *If φ is in clausal form, i.e. $\varphi = cnf(\varphi)$, then for any last-flipped variable V NC-UpdateScores(φ, V) has the same computational complexity as UpdateScores(φ, V).*

In fact the computations described by Schemas (1) and (2) perform the same number $\overline{C}_\varphi$ of external loops, one for any clause/main subformula in which V occurs. Moreover, for any index k, they perform the same number $\overline{l}_\varphi$ of internal loops, one for any variable/literal occurring in φ_k. If V occurs in a clause φ_k, then *NC-UpdateScores(φ_k, V)* requires only one updating, as it updates only the value of the literal containing V. Thus Schema (2) requires $\overline{C}_\varphi \cdot (1 + \overline{l}_\varphi)$ updatings versus the $\overline{C}_\varphi \cdot \overline{l}_\varphi$ updatings of Schema (1). Thus both computations are $O(\overline{C}_\varphi \cdot \overline{l}_\varphi)$.

Proposition 4.2 *Let φ be an INF formula. Let $\phi = cnf(\varphi)$. The computation complexity of NC-UpdateScores(φ, V) grows linearly with the number of literals of φ. The computation complexity of UpdateScores(ϕ, V) grows exponentially with the number of literals of φ.*

More precisely, the complexity of *UpdateScores(ϕ, V)* grows exponentially with the number of "$\vee$" and "$\equiv$" occurring in φ. The first statement follows immediately the fact that in Schema (2) the number of the variables occurring in φ_k is less or equal the number of literals in φ_k. For the second statement, we have seen before that *UpdateScores(ϕ, V) is* $O(\overline{C}_\phi \cdot \overline{l}_\phi)$, where $\overline{C}_\phi$ is the average number of clauses of ϕ which V occurs in, $\overline{l}_\phi$ is the average length of such clauses. Suppose $p_v(\varphi)$ is the number of clauses of ϕ (i.e. cnf(φ)) in which V occurs and $p(\varphi)$ is the global number of clauses of ϕ. Then we have:
$$p_v(\varphi_1 \vee \varphi_2) = p_v(\varphi_1) \cdot p_v(\varphi_2) + p_v(\varphi_1) \cdot [p(\varphi_2) - p_v(\varphi_2)] + p_v(\varphi_2) \cdot [p(\varphi_1) - p_v(\varphi_1)].$$
Thus $\overline{C}_\phi$ (i.e. the average value of $p_v(\varphi)$) grows exponentially with the number of "$\vee$" occurrences. Moreover $\overline{l}_\phi$ grows linearly with the number of "$\vee$" occurrences, as $\overline{l}_{cnf(\varphi_1 \vee \varphi_2)} = \overline{l}_{cnf(\varphi_1)} + \overline{l}_{cnf(\varphi_2)}$. Similar considerations follow for "$\equiv$" occurrences, as $A \equiv B$ is logically equivalent to $(\neg A \vee B) \wedge (A \vee \neg B)$.

These results should not be a surprise. Although the updating algorithm of GSAT [SLM92, SK93] is extremely efficient, generally $|\phi| \gg |\varphi|$, as $|cnf(\varphi)|$ grows exponentially with the number of "$\vee$" and "$\equiv$" inside φ. For instance, the CNF conversion of the formula of Figure 2 is long 360 clauses. Therefore any updating algorithm which is able to handle φ *directly* can behave much more efficiently than *UpdateScores(ϕ, V)*.

Two more observations are worth doing. The first is about other clausal-form conversion algorithms [PG86, dlT90]. Such methods are based on the introduction of new variables, each of them representing a subformula of the original input φ. For instance, in [PG86] a new variable is introduced for any non-atomic subformula of φ. Such methods have been introduced to overcome the exponential growth of the algorithm for a non-clausal version of GSAT. An accurate schema of such an algorithm is presented in appendix A. In order to distinguish it from that used in GSAT, we refer to it as "*NC-UpdateScores(φ, V)*". We consider only "$\wedge$" and "$\vee$" connectives, leaving the description of the "$\equiv$" case to the appendix. The updating of the $s^-(T, \varphi_k)$ scores is dual.

The input formula φ can be stored in a tree-like data structure, like the one given in Figure 2. We associate further information to any node/subformula φ_k: the current value of $s(T, \varphi_k)$ ($\varphi_k.s$ from now on), a list containing both the names of the variables A_i occurring in φ_k and their current values $s(T_i, \varphi_k)$ ($\varphi_k.s[i]$ from now on). *NC-UpdateScores(φ, V)* works recursively on the tree-structure of φ, from the leaves to the root.

If $\varphi = \bigwedge_k \varphi_k$, then the core of the computation of *NC-UpdateScores(φ_k, V)* is coarsely described by the following schema (see appendix A):

```
(2)   for any $\varphi_k$ main subformula of $\varphi$ : $V$ occurs in $\varphi_k$ do
          NC-UpdateScores($\varphi_k, V$)
          for any variable $A_i$: $A_i$ occurs in $\varphi_k$ do
              $\varphi.s[i] := \varphi.s[i] + \varphi_k.s[i]_{new} - \varphi_k.s[i]_{old} - diff_k$
```

where the $diff_k$'s are previously-computed parameters. If $\varphi = \bigvee_k \varphi_k$, then the schema is dual: [5]

```
(3)   for any $\varphi_k$ main subformula of $\varphi$ : $V$ occurs in $\varphi_k$ do
          NC-UpdateScores($\varphi_k, V$)
          for any variable $A_i$: $A_i$ occurs in $\varphi_k$ do
              $\varphi.s[i] := \varphi.s[i] \cdot \varphi_k.s[i]_{new} / \varphi_k.s[i]_{old} / ratio_k$ .
```

The scores must be updated only for those subformulas φ_k where the last-flipped variable V occurs. *NC-UpdateScores(φ, V)* computes the new values of $s(T_i, \varphi)$ for any variable A_i, returning the same values as *UpdateScores($cnf(\varphi), V$)*. Therefore in *NC-GSAT(φ)* the function *hill-climb()* returns exactly the same variable sets as in *GSAT(cnf(φ))*, so that the two procedures perform the same flips and return the same result.

In order to make a comparison between *UpdateScores($cnf(\varphi), V$)* and *NC-UpdateScores(φ, V)*, we first show that they are computationally equivalent if the input formula φ is already in clausal form. Then we show that the second is more efficient in the general case: the farther from CNF is the structure of φ, the greater is the performance gap.

N is held constant [MSL92, SK93], so that the updating step requires a constant time. However in the situation we are considering, where ϕ is the result of converting in some way a non-clausal formula φ, the length of the clauses is not fixed and there is no general relation between C and N.

[5] As we shall notice in appendix A, we must be careful in handling the possible $x/0$ situations. In particular, we want that $(s_i \cdot s_j)/s_j = s_i$ even if $s_j = 0$.

- **base**: If φ is a single literal, then evaluating its truth value under T requires constant time. In fact, if we represent T as an array of booleans, then the truth value of the A_k $[\neg A_k]$ is $T[k]$ $[\neg T[k]]$. Thus $Time(s^{\pm}(\varphi, T))$ is constant.
- **step**: φ is in INF, so that φ is in the form $\varphi_1 \diamond \varphi_2$, with $\diamond \in \{\wedge, \vee, \equiv\}$. Suppose for inductive hypothesis that $Time(s^{\pm}(\varphi_i, T)) \leq a_i|\varphi_i| + b_i$. Then $Time(s^{\pm}(\varphi, T)) \leq \cdot max_i(a_i) \cdot |\varphi| \ + 2 \cdot max_i(b_i) \ + 6$. □

If ϕ is a clausal formula, then the computation of the score of an assignment T is linear in the number of literals of ϕ. Thus, if $\phi = cnf(\varphi)$, then computing the score of T for ϕ would be $O(2^{|\varphi|})$, while $s(T, \varphi)$ performs the same result directly in linear time.

4 GSAT for non-clausal formulas

We are now ready to modify GSAT to work for non-clausal formulas. We shall generically call "NC-GSAT" (non-clausal GSAT) any modification of GSAT such that *NC-GSAT(φ)* computes *GSAT(cnf(φ))* directly. We could conceive NC-GSAT in many possible ways. We shall describe a version which behaves similarly to GSAT when the input formula is in clausal form.

Following the description of GSAT in Section 2, we note that the functions *initial()*, *hill-climb()*, *pick()* and *flip()* are independent of the structure of the input formula ϕ. As a consequence there is no need to redefine them, and we can focus our attention on the internal representation of the data and on the way to implement *UpdateScores()*. In GSAT [SLM92, SK93] such procedure is implemented very efficiently, as it is coarsely described in the following schema: [3]

```
(1)       for any clause $c_k$ in $\phi$ such that $V$ occurs in $c_k$ do
              for any variable $A_i$: $A_i$ occurs in $c_k$ do
                  $\Delta s_i := \Delta s_i \pm \ldots$
```

This requires analizing only the clauses which the last-flipped variable V occurs in. Thus updating all the Δs_i values requires, on average, $O(\overline{C}_\phi \cdot \overline{l}_\phi)$ operations, where $\overline{C}_\phi$ is the average number of clauses of ϕ which V occurs in, $\overline{l}_\phi$ the average length of such clauses [4]. In this section we describe the ideas underlying the updating

[3] These considerations come from the analysis of the source code of GSAT.

[4] It could be argued that normally GSAT is tested by random problems where both the length of the clauses and the ratio C/N between the total number of clauses C and the number of variables

The correctness and the linearity of $s(T,\varphi)$ are respectively asserted by the following results.

Theorem 3.1 *Let φ be an INF formula and T a truth assignment for the propositional variables of φ. Then the function $s(T,\varphi)$ computes the score of T for φ.*

Proof [Sketch] By induction on the structure of φ.

- **base**: If φ is a literal the result is trivial, as *cnf(φ)* $= \varphi$ is a single clause with a single literal.
- **step**: $s^-(T,\varphi_i)$ is $s(T,\neg\varphi_i)$, thus we can consider the only function $s(T,\varphi)$ and apply it to all $\{\wedge,\vee,\equiv,\neg\wedge,\neg\vee,\neg\equiv\}$ [2] cases. For instance, $s^-(T,\varphi_1\equiv\varphi_2) = s(T,\neg(\varphi_1\equiv\varphi_2))$.

  Now let φ_i be the main subformulas of φ, let $\bigwedge_{k_i} A_{ik_i}$ be *cnf(φ_i)* for any i. We call $F(\varphi,T)$ the set of all the clauses of *cnf(φ)* which are false under T and $|F(\varphi,T)|$ its size. For inductive hypothesis suppose $s(T,\varphi_i)$ computes $|F(\varphi_i,T)|$.

  If φ is $\varphi_1\wedge\varphi_2$, then *cnf(φ)* is *cnf(φ_1)* $\wedge$ cnf(φ_2). This implies that $F(\varphi,T)$ is $F(\varphi_1,T)\bigcup F(\varphi_2,T)$, so that $s(T,\varphi)$ is $s(T,\varphi_1)+s(T,\varphi_2)$.

  If φ is $\varphi_1\vee\varphi_2$, then *cnf(φ)* is the conjunction of the clauses in the form $A_{1k_1}\vee A_{2k_2}$, for all possible values of k_1 and k_2. Such clauses are false if and only if both their components are false. This means that $F(\varphi,T)$ is the cross product $F(\varphi_1,T)\times F(\varphi_2,T)$, so that $s(T,\varphi)=s(T,\varphi_1)\cdot s(T,\varphi_2)$.

  Now we note that all the other forms $\{\equiv,\neg\wedge,\neg\vee,\neg\equiv\}$ can be rewritten in terms of $\{\wedge,\vee,\neg\}$. Thus the proof of the $\wedge$ and $\vee$ cases are sufficient to prove all the others. For instance, $s^-(T,\varphi_1\equiv\varphi_2) = s(T,\neg(\varphi_1\equiv\varphi_2)) = s(T,(\varphi_1\wedge\neg\varphi_2)\vee(\neg\varphi_1\wedge\varphi_2)) = (s(T,\varphi_1)+s(T,\neg\varphi_2))\cdot(s(T,\neg\varphi_1)+s(T,\varphi_2)) = (s(T,\varphi_1)+s^-(T,\varphi_2))\cdot(s^-(T,\varphi_1)+s(T,\varphi_2))$. $\square$

Theorem 3.2 *Let φ be a INF propositional formula and T a truth assignment for the propositional variables of φ. Then the number of operations required for calculating $s(T,\varphi)$ is $O(|\varphi|)$.*

Proof [Sketch] By induction on the structure of φ. We call $Time(s^{\pm}(\varphi,T))$ the number of operations required for calculating both $s(T,\varphi)$ and $s^-(T,\varphi_i)$.

[2] For instance, by "$\neg\equiv$" we mean formulas like $\neg(\varphi_1\equiv\varphi_2)$.

φ	$s(T,\varphi)$	$s^-(T,\varphi)$
φ *literal*	$\begin{cases} 0 & if\ T \models \varphi \\ 1 & otherwise \end{cases}$	$\begin{cases} 1 & if\ T \models \varphi \\ 0 & otherwise \end{cases}$
$\bigwedge_k \varphi_k$	$\sum_k s(T,\varphi_k)$	$\prod_k s^-(T,\varphi_k)$
$\bigvee_k \varphi_k$	$\prod_k s(T,\varphi_k)$	$\sum_k s^-(T,\varphi_k)$
$\varphi_1 \equiv \varphi_2$	$\begin{cases} s^-(T,\varphi_1)\cdot s(T,\varphi_2)+ \\ s(T,\varphi_1)\cdot s^-(T,\varphi_2) \end{cases}$	$\begin{cases} (s(T,\varphi_1)+s^-(T,\varphi_2))\cdot \\ (s^-(T,\varphi_1)+s(T,\varphi_2)) \end{cases}$

Notice that $s^-(T,\varphi_k) = s(T,\neg\varphi_k)$. The distinction between $s(T,\varphi_k)$ and $s^-(T,\varphi_k)$ is due to the *polarity* of the current subformula φ_k. During the computation of $s(T,\varphi)$ a call to the function $s(T,\varphi_j)$ [$s^-(T,\varphi_j)$] is invoked if and only if φ_j is a positive [negative] subformula of φ. Thus $s^-(T,\varphi_j)$ will be computed only for those subformulas φ_j which occur inside φ under the scope of some "$\equiv$" connective.

Example 3.1 Figure 2 represents the computation tree of the score of a truth assignment T for the INF formula φ:

$$(((\neg A \wedge \neg B \wedge C) \vee \neg D \vee (\neg E \wedge \neg F)) \wedge \neg C \wedge ((\neg D \wedge A \wedge \neg E) \equiv (C \wedge F)))\vee$$
$$(D \wedge \neg E \wedge B) \vee (((D \wedge \neg A) \vee (\neg F \wedge D \wedge \neg B) \vee \neg F) \wedge A \wedge ((E \wedge \neg C \wedge F) \vee \neg B)).$$

T assigns "true" to all the variables of φ. The information in square brackets associated to any subformula φ_j represents $[s(T,\varphi_j), s^-(T,\varphi_j)]$. For instance, if we consider the small subtree in the left of Figure 2, then the score is computed in the following way:

$$\begin{array}{ll}
s(T,\ (\neg A \wedge \neg B \wedge C) \vee \neg D \vee (\neg E \wedge \neg F)\ ) = & ;\ s(T, \bigvee_k \varphi_k) = \prod_k s(T,\varphi_k) \\
s(T,\neg A \wedge \neg B \wedge C)\cdot s(T,\neg D)\cdot s(T,\neg E \wedge \neg F) = & ;\ s(T, \bigwedge_k \varphi_k) = \sum_k s(T,\varphi_k) \\
(s(T,\neg A)+s(T,\neg B)+s(T,C))\cdot s(T,\neg D)\cdot(s(T,\neg E)+s(T,\neg F)) = & ;\ literals \\
(1+1+0)\cdot 1\cdot(1+1) = 4. &
\end{array}$$

□

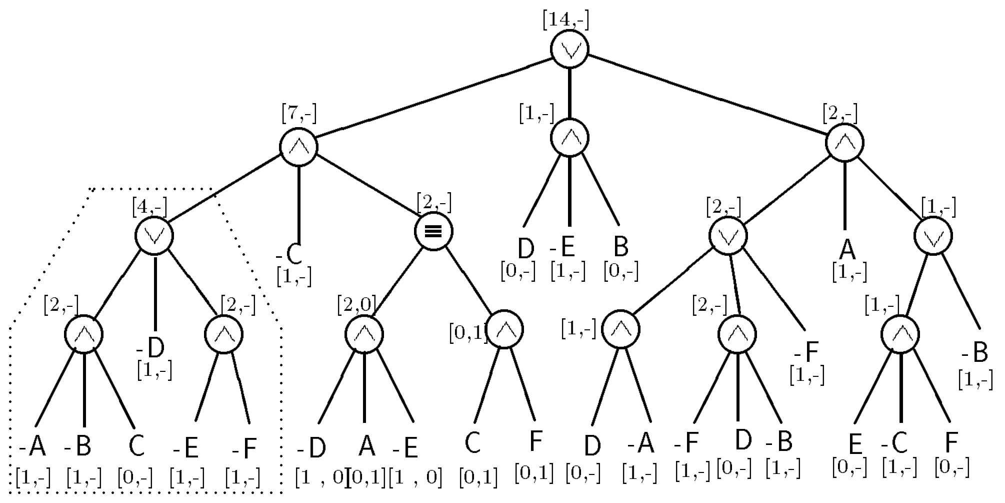

Figure 2: The computation tree of $s(T,\varphi)$.

```
procedure $GSAT(\phi)$
    for $i := 1$ to $Max\text{-}tries$ do
        $T := initial(\phi)$
        for $j := 1$ to $Max\text{-}flips$ do
            if $T \models \phi$
                then return $T$
                else $Poss\text{-}flips := hill\text{-}climb(\phi, T)$
                    $V := pick(Poss\text{-}flips)$
                    $T := flip(V, T)$
                    $UpdateScores(\phi, V)$
        end
    end
    return "no satisfying assignment found".
```

Figure 1: A general schema for GSAT.

3 An extended notion of score

In order to both simplify the explanation and improve the efficiency of our computations, we shall assume that any non-clausal formula φ is formed only by $\wedge$,$\vee$,$\equiv$ connectives and positive/negative literals. We say it is in "Iff Negative Form" ("INF" from now on). This assumption is not restrictive. All the definitions and theorems of this section can be easily extended to propositional formulas. Moreover, any propositional formula φ can be converted into INF in linear time with respect to the number of literals of φ ($|\varphi|$ from now on) and the resulting formula has the same number of literals of φ.

Let *cnf(*φ*)* be the result of converting a propositional formula φ into clausal form with the standard method (i.e. by applying the rules of De Morgan). The following definition introduces a generalized notion of score.

Definition 3.1 *The score of T for a propositional formula φ is the number of the clauses of cnf(φ) which are false under T.*

cnf() is the "natural" clausal form conversion. Unlike the results of other conversion algorithms, *cnf(*φ*)* has the same number of propositional variables of φ and is logically equivalent to φ. The problem with *cnf()* is the exponential size growth of *cnf(*φ*)*. Definition 3.1 overcomes such problem, as it is possible to introduce a function $s(T, \varphi)$ which computes in linear time the score of T for an INF formula φ *directly*, i.e. without converting φ into clausal form. We define such function recursively as follows:

the "standard" clausal conversion is used, as such conversion can be time and space exponential in the length of the input formula. A maybe even worse problem may arise with step (ii), as GSAT is given a input formula whose length is exponential in the length of the former input φ. This causes a huge increase of both space and time complexity. The increase of complexity is not avoided by using other conversion algorithms [PG86, dlT90], as they much enlarge the search space.

In this paper we define a particular "score" function which allows (a slightly-modified version of) GSAT to be applied *directly* to non-clausal formulas without performing the clausal form conversion. In Section 2 we give a brief description of GSAT. In Section 3 we define a new notion of "score" of a truth assignment T for a formula φ. We also introduce a particular function which is able to compute it in linear time. Section 4 describes how to modify GSAT to make it work with non-clausal input formulas. We then present a computational analysis, comparing the efficiency of the modified algorithm with the standard GSAT when applied to a clause-converted formula. Finally we show how most variants of GSAT can be adapted to non-clausal formulas as well. One possible schema of (the core of) a modified GSAT algorithm is reported in appendix A.

2 GSAT

Suppose ϕ is a clausal formula. The number of clauses of ϕ which are false under a truth assignment T is called the *score* [1] of T for ϕ. The score is considered a measure of the goodness of the choice of T. In fact T satisfies ϕ iff its score is zero, and the less the score is the closer to a solution is T.

The schema of Figure 1 describes both GSAT and many of its possible variants. We use the notation from [GW93]. *initial()* returns a random truth assignment T, *hill-climb()* returns the set of the variables whose single flipping causes the greatest decrease of the score, *pick()* chooses randomly one of these variables, *flip()* returns the truth assignment T with V's assignment flipped (inverted), *UpdateScores()* updates the internal data after each flip. Let T_i be the assignment T with the value of the variables A_i flipped. The function *hill-climb*(ϕ, T) needs to know the scores of T_i for ϕ, for all variables A_i. If we store, for all A_i, the variations of score caused by the flipping (i.e. $\Delta s_i = s(T_i, \phi) - s(T, \phi)$) then *hill-climb*$(\phi, T)$ will simply have to select the variables A_i with the best Δs_i. Thus the core of the internal loop consists in the updating of the values Δs_i after each flip, which is performed by *UpdateScores()*.

This paper exploits the observation that in the schema of Figure 1 the computation of the scores is the only step where the input formula ϕ is required to be in clausal form.

[1] The score is usually defined as the number of clauses of ϕ which are *true* under T. Our definition is equivalent.

Applying GSAT To Non-Clausal Formulas [*]

Roberto Sebastiani

Mechanized Reasoning Group,
DIST, via Opera Pia 11/a, 16145 Genova, Italy
Mechanized Reasoning Group,
IRST, loc. Pantè, 38050 Povo, Trento, Italy.
rseba@dist.unige.it

March 4, 1994

Abstract

In this paper we describe how to modify GSAT so that it can be applied to non-clausal formulas. The idea is to use a function which computes the number of clauses of the CNF conversion of a formula which are false under a certain truth assignment, without constructing the conversion itself. The proposed methodology applies to most variants of GSAT.

1 The goal

Many algorithms have been devised to solve the well-known NP-Complete SAT problem [Coo71, GJ79]. Among them, GSAT (Greedy SAT) [SLM92, SK93], an incomplete search algorithm for SAT, is now state of the art. Though incomplete, GSAT has been shown to be generally much faster than the traditional complete Davis-Putnam algorithm [DP60].

Like most SAT algorithms, GSAT requires that the input formula be in clausal form. A possible way to find a truth assignment for a non-clausal formula φ is (i) to convert φ into clausal form, and then (ii) to apply GSAT to the converted formula. Step (i) causes an extra amount of computation time, which can be extremely heavy if

[*]Fausto Giunchiglia and Enrico Giunchiglia have given substantial and continuous feedback during the whole development of the research. Toby Walsh provided important feedback about a former version of the paper. Bart Selman, Henry Kautz and the AT&T company are thanked for making the source code of GSAT freely accessible via FTP.

Applying GSAT to Non-Clausal Formulas

R. Sebastiani

January 1994
Technical Report # 94-0018