Improving SAT using 2SAT

Lei Zheng **Peter J. Stuckey**

Dept. of Computer Science & Software Engineering
The University of Melbourne,
VIC 3010, Australia
Email: {zhengl,pjs}@cs.mu.OZ.AU

Abstract

Propositional satisfiability (SAT) is a fundamental problem of immense practical importance. While SAT is NP-complete when clauses can contain 3 literals or more, the problem can be solved in linear time when the given formula contains only binary clauses (2SAT). Many complete search algorithms for SAT solving have taken advantage of 2SAT information that occurs in the statement of the problem in order to simplify the solving process, only one that we are aware of uses 2SAT information that arises in the process of the search, as clauses are simplified. There are a number of possibilities for making use of 2SAT information to improve the SAT solving process: maintaining 2SAT satisfiability during search, detecting unit consequences of the 2SAT clauses, and Krom subsumption using 2SAT clauses. In this paper we investigate the tradeoffs of increasing complex 2SAT handling versus the search space reduction and execution time. We give experimental results illustrating that the SAT solver resulting from the best tradeoff is competitive with state of the art Davis-Putnam methods, on hard problems involving a substantial 2SAT component.

Keywords: SAT, propositional satisfiability, 2SAT, Krom subsumption.

1 Introduction

Propositional satisfiability (SAT) is the problem of deciding if there is an assignment for the variables in a propositional formula that makes the formula true. It is a core problem in mathematical logic and computing theory. SAT is of special concern to AI because of its direct relationship to deductive reasoning. In recent years, there have been increasing interest in this area.

Since SAT is NP-complete [Cook, 1971], this problem is unlikely to have an exact polynomial time algorithm. However, the problem becomes linear when the given formula consists of binary clauses only. An important heuristic for SAT problem solving is to first solve 2SAT clauses with fast linear time algorithms, in order to reduce the problem for subsequent solving. The simplified problem can then be tackled using a Davis-Putnam complete search procedure [Davis et al., 1962, Gu et al., 1997]. This idea has been used in many SAT solvers, such as Stamm's solver [Buro and Kleini-Büning, 1992] and Larrabee's [Larrabee, 1992]. Recently Brafman [Brafman, 2000] has also built a 2SAT simplifier for those propositional formulas containing a large fraction of binary clauses.

All the algorithms mentioned above are based on the implication graph [Aspvall et al., 1979]. Recently, a novel linear-time algorithm for 2SAT, BinSat, was proposed by Alvaro del Val [Alvaro del Val, 2000]. BinSat is based directly on unit resolution rather than the implication graph. Therefore neither the complicated graph data structures nor graph simplifying processes are required.

In this paper we explore the possibilities of using 2SAT information that is derived during the Davis-Putnam search procedure to improve the search. The first possibility is simply to maintain satisfiability of the 2SAT subproblem resulting during search. We use an incremental version of the BinSat algorithm, at each stage of the search to determine a solution to the 2SAT subproblem. A consequence of this approach is that we maintain at all times a valuation which satisfies the binary clause sub-problem of the current problem. If this also happens to satisfy all clauses with 3 or more literals we have solved the entire problem. Otherwise we can use the valuation to determine the unsatisfied clauses with 3 or more literals and use this information in directing the search.

The incremental BinSat algorithm can detect new unit clause consequences of the binary clauses. These new unit clause consequences of the binary clauses can then be used to simplify the problem via unit resolution, and in turn lead to new binary clauses being discovered.

The second possibility arises from trying to detect all unit clause consequences of the 2SAT subproblem. Although the incremental BinSat algorithm detects some unit clause consequences of the 2SAT subproblem it does not detect all. By building the resolution closure of the 2SAT subproblem we are guaranteed to detect all unit consequences. Resolution closure adds for each pair of clauses of the form $[\overline{x}y]$ and $[xz]$ the clause $[yz]$.

The third possibility arises from considering what more information can be obtained from the 2SAT subproblem. Krom subsumption resolution [Van Gelder and Tsuji, 1995] uses a binary clause $[xy]$ to reduce the size of a longer clause $[\overline{x}yzt]$, to the equivalent $[yzt]$. Krom subsumption resolution makes the SAT problem description strictly smaller in size. and allows the detection of new binary and unit clause consequences.

While each new possibility reduces the search space of Davis-Putnam procedure, each possibility also adds to the amount of bookkeeping required to efficiently perform these operations. Since this occurs in the context of the backtracking Davis-Putnam search, we need to be able to quickly update this information during forward search, and undo changes to the information on backtracking. In some cases the overhead of maintaining this information may outweigh the benefits in reducing the search space. In this paper we explore these possibilities for using 2SAT information to improve SAT search, and describe efficient data structures for maintaining the appropriate information.

The only other work we are aware of which investigates how 2SAT information derived during the Davis-Putnam search is that of Van Gelder and Yumi [Van Gelder and Tsuji, 1995]. They investigate using resolution closure and Krom subsumption information to improve the Davis-Putnam search but do not make use of a solution to the 2SAT subproblem. We show that this information can be advantageous when problems have a large 2SAT component.

The paper is organized as follows. In the next section, we give our notation for SAT problems and discuss two 2SAT algorithms: the standard backtrack once algorithm [Dalal and Etherington, 1992] and Alvaro's linear time 2SAT algorithm based on unit resolution [Alvaro del Val, 2000]. In Section 3 we define an incremental version of Alvaro's algorithm, IncBinSat, which we then extend to include binary clause resolution closure, and Krom subsumption resolution.

In Section 4 we defining our SAT algorithm, SATB which makes use of these incremental 2SAT algorithms, in particular discussing different choices of search strategy. In Section 5, we focus on the data structures required to allow the efficient restoration of constraint store information during backtracking. In Section 6, we present some experimental results comparing various choices in making use of 2SAT information. We then compare the best tradeoff discovered SATB against state of the art SAT solvers SATO [Zhang and Stickel, 2000] and EQSATZ [Li, 2000]. Finally in Section 7 we conclude.

2 2SAT algorithms

We use the following notation for SAT problems. A *variable* is an element of some given variable set $Vars$. We use p and q, possibly subscripted, to refer to variables. A *literal* is one of p or its negation $\overline{p}$ where $p \in Vars$. We use w, x, y and z possibly subscripted, to refer to literals. The *complement* of a literal $\overline{x}$ is $\overline{p}$ if x is a variable p, and p if x is a negated variable $\overline{p}$. A *clause* is a set of literals. We use C possibly subscripted to refer to clauses. A *unit clause* is a singleton set. A *binary clause* C is a clause where $|C| = 2$.

In order to clarify notation we use square brackets to denote sets of literals, so $[p_1]$ is a unit clause, and $[\overline{p_2}p_3]$ is a binary clause. $\Box$ represents the empty (unsatisfiable) clause. We use the notation $[x]$ to match a unit clause, $[xy]$ to match a binary clause, and $[xC]$ to match any clause of 1 or more literals with x being one literal and C the remaining set (clause). We use $[xyC]$ similarly. A *theory* is a set of clauses. We let $vars(T)$ be the set of variables $p \in Vars$ appearing in theory T, and $size(T)$ be the number of literal occurrences in T, i.e. $size(T) = \Sigma_{C \in T}|C|$. Let $occurs(x, T)$ be the number of occurrences of literal x in theory T.

```
procedure BTOSat(T)
    T := PropUnit(T)
    while (□ ∉ T and ∃p ∈ vars(T)
    such that [p] ∉ T and [p̄] ∉ T) {
        T' := PropUnit(T ∪ {[p]});
        if (□ ∈ T') then
            T := PropUnit(T ∪ {[p̄]});
        else T := T'; }
    if (□ ∈ T) then return false;
    else return true;
end

procedure PropUnit(T)
    while (∃[x] ∈ T and ∃[x̄C] ∈ T)
        T := (T − {[x̄C]}) ∪ {C};
    return T
end
```

Figure 1: Backtrack-once algorithm

2.1 The "backtrack once" 2SAT algorithm: BTOSat

The classic algorithm for 2SAT is based on "guess and deduce" and was introduced in 1976 [Even et al., 1976]. In this section we introduce an enhanced version of this algorithm that was produced by David and Etherington in 1992 [Dalal and Etherington, 1992]. This algorithm, which is usually named BTOSat (for "backtrack once"), has complexity $O(nm)$ where $n = |vars(T)|$ and $m = |T|$. BTOSat is shown in Figure 1.

In fact, BTOSat can be seen as a restricted case of the classic Davis-Putnam procedure for general SAT problems. The basic idea is quite straightforward. It chooses literals to assign, and propagates their value with unit resolution. If an assignment of p leads to a contradiction, then we switch to $\overline{p}$. The key point is that BTOSat never needs to backtrack over previous decisions. We may need to undo an assignment to p, but if the complementary assignment $\overline{p}$ fails also then the formula is unsatisfiable. Hence BTOSat backtracks at most one step [Dalal and Etherington, 1992].

Examining the BTOSat algorithm, we can easily find that the propagation of tentative values is stopped if a contradiction occurs, which results in an $O(mn)$ complexity. In fact, it is not necessary to stop at each conflict point as long as we can tell whether it is a determinate assignment or a tentative assignment that results in a contradiction. A new linear time 2SAT algorithm, BinSat, defined by Alvaro in 2000 [Alvaro del Val, 2000] reflects this idea.

```
procedure TempPropUnit(x,T,A)
    if ([x̄] ∈ A) then {
        /*temporary conflict*/
        T := PropUnit(T ∪ {[x]});
        A := {[y] ∈ A | [y] ∉ T ∧ [ȳ] ∉ T};
        return (T, A); }
    A := A ∪ {[x]};
    for each [x̄y] ∈ T {
        if ([y] ∉ A) then
            (T, A):=TempPropUnit(y,T,A);
        if (□ ∈ T) then break; }
    return (T, A);
end

procedure BinSat(T)
    T := PropUnit(T);
    A := ∅;
    while (□ ∉ T and ∃p ∈ vars(T)
            where [p] ∉ T ∪ A and [p̄] ∉ T ∪ A)
        (T, A) := TempPropUnit(p,T,A);
    if (□ ∈ T) then return false;
    else return true;
end
```

Figure 2: Linear time 2SAT algorithm [Alvaro del Val, 2000].

2.2 A linear time algorithm for 2SAT: BinSat

BinSat [Alvaro del Val, 2000] is described in Figure 2. It maintains a tentative assignment A of values to variables represented as a set of unit clauses. A permanent assignment of a value to a variable p is represented by $[p]$ or $[\overline{p}]$ occurring in T.

Initially, A is empty. BinSat repeatedly selects a variable p with no permanent or temporary assignment; and temporarily assigns it a value *true* (represented by adding $[p]$ to the theory A). This temporary assignment is then propagated by temporary unit resolution to find new temporary assignments. If this process leads to assigning a temporary value to literal x of *false* which has already been assigned *true* then there is a contradiction and the algorithm permanently assigns x to *false*.

***Example* 1** *Consider the execution of BinSat on the theory* $T_0 = \{[\overline{p_1 p_2}], [p_2 p_3], [\overline{p_3} p_4], [\overline{p_3} p_5], [\overline{p_5} p_2]\}$. *Suppose the first literal selected is* p_1 *then the execution calls* $TPU(p_1, T_0, \emptyset)$, $TPU(\overline{p_2}, T_0, \{[p_1]\})$, $TPU(p_3, T_0, \{[p_1], [\overline{p_2}]\})$. *Inside this call, the call* $TPU(p_4, T_0, \{[p_1], [\overline{p_2}], [p_3]\})$ *immediately returns, then the call* $TPU(p_5, T_0, \{[p_1], [\overline{p_2}], [p_3], [p_4]\})$ *leads to* $TPU(p_2, T_0, \{[p_1], [\overline{p_2}], [p_3], [p_4], [p_5]\})$ *which detects a contradiction. The call* $PU(T_0 \cup \{[p_2]\})$ *returns* $T_1 = T_0 \cup \{[\overline{p_1}], [p_2]\}$ *and the final answer is* T_1 *and temporary assignment* $\{[p_3], [p_4], [p_5]\}$.

Note that if p_2 *is selected as the first literal the resulting solution is* T_0 *and* $\{[p_2], [\overline{p_1}], [p_3], [p_4], [p_5]\}$ *which does not detect any unit clause consequences of the theory.*

With the above description, it is clear that BinSat does not stop the unit propagation even when an inconsistency of temporary assignments is found. Instead, it searches for conflicts and yields unit clause consequences. As a result, any tentative assignment is explored in full only once, and revoked at most once. It can be shown that BinSat has $size(T)$ complexity [Alvaro del Val, 2000].

3 An incremental BinSat

BinSat only solves pure 2SAT problems. If it is applied directly to a general SAT problem, it can serve as a preprocessor for those formulas containing binary clauses by detecting unit clause consequences. Obviously, a 2SAT preprocessor can significantly reduce the search space for those problems containing many binary clauses, e.g. graph colouring and parity problems.

But consider the Davis-Putnam style complete backtracking search for a solution to an arbitrary SAT problem. Even if there are no binary clauses in the original problem formulation, eventually unit resolution occurring in the search will reduce many clauses to involve 2 or less literals. If we apply BinSat on these newly deduced binary clauses we may more quickly detect conflict (in temporary assignments), and thus yield more unit resolutions.

In order to use BinSat efficiently within a backtracking search we need to make it incremental, so that we don't repeat work unnecessarily. The pseudocode for an incremental version of BinSat, IncBinSat, is given in Figure 3.

IncBinSat maintains a tentative assignment for each variable which has no permanent assignment. Every time we add a new binary clause (by unit propagation) we need do nothing if the current temporary (or permanent) assignment satisfies the new clause. Otherwise we select one of the literals of the new clause to set temporarily *true*. This will either succeed, or lead to a contradiction causing the other literal in the clause to be permanently set to *true* (a unit clause will be added to T).

Note that in order to differentiate between contradictions arising through temporary unit propagations, and simple changes of temporary values due to solving a new incremental problem we maintain a global variable *current_time* keeping track of the number of calls to IncBinSat (in forward computation). We assume that each tentative assignment clause $[x] \in A$ has an attached time, $time([x])$, telling when the tentative assignment was made. We do not detect a contradiction unless a new tentative assignment of $[\overline{x}]$ is made at the same time as the assignment $[x]$.

The only additional complexity of the algorithm is intermingling of the processing of the new binary clauses, that arise from unit propagations discovered by BinSat.

***Example* 2** *Consider adding the additional binary clause* $[\overline{p_5} p_2]$ *to the theory* $T_{-1} = \{[\overline{p_1 p_2}], [p_2 p_3], [\overline{p_3} p_4], [\overline{p_3} p_5]\}$, *where the current temporary assignment is* $A_0 = \{[p_1]^1, [\overline{p_2}]^1, [p_3]^1, [p_4]^1, [p_5]^1\}$ *(the superscripts denote the time of the assignment). The current time becomes 2. The new clause is not satisfied by the current temporary assignment. Suppose we choose the first literal, calling* $ITPU(\overline{p_5}, T_{-1}, A_0)$. *A*

procedure IncTempPropUnit(x,T,A)
  if ($[\overline{x}] \in A$ **and** $time([\overline{x}]) = current\_time$) **then**
    /*temporary conflict*/
    return IncPropUnit($T \cup \{[x]\}$,A);
  $A := (A - \{[x], [\overline{x}]\}) \cup \{[x]\}$;
  $time([x]) := current\_time$
  for each $[\overline{x}y] \in T$ {
    if ($[y] \notin A$) **then**
      $(T, A) :=$ IncTempPropUnit(y,T,A);
    if ($\Box \in T$) **then break**; }
  return (T, A);
end

Procedure IncPropUnit(T,A)
  $B := \emptyset$;
  while ($\exists[x] \in T$ **and** $\exists[\overline{x}C] \in T$) {
    $T := T - \{[\overline{x}C]\} \cup \{C\}$;
    if C is of the form $[y]$ **then**
      $A := A - \{[y], [\overline{y}]\}$;
    if C is of the form $[yz]$ **then**
      $B := B \cup \{C\}$; }
  foreach $C \in B$
    $(T, A) :=$ IncBinSat(C,T,A);
  return (T, A);
end

procedure IncBinSat($[xy]$,T,A)
  $current\_time := current\_time + 1$;
  if ($[x] \in T$ **or** $[y] \in T$) **then**
    return (T, A);
  if ($[\overline{x}] \in T$) **then**
    return IncPropUnit($T \cup \{[y]\}$, $A - \{[y], [\overline{y}]\}$);
  if ($[\overline{y}] \in T$) **then**
    return IncPropUnit($T \cup \{[x]\}$, $A - \{[x], [\overline{x}]\}$);
  %% insertion point
  if ($[x] \in T \cup A$ **or** $[y] \in T \cup A$) **then**
    return (T, A);
  else return IncTempPropUnit(x,$T \cup \{[xy]\}$,A);
end

Figure 3: Incremental algorithm for 2SAT.

contradiction is not detected since the current time is greater than that of literals in A_0. This leads to a call $ITPU(\overline{p_3}, T_{-1}, \{[p_1]^1, [\overline{p_2}]^1, [p_3]^1, [p_4]^1, [\overline{p_5}]^2\})$, then $ITPU(p_2, T_{-1}, \{[p_1]^1, [\overline{p_2}]^1, [\overline{p_3}]^2, [p_4]^1, [\overline{p_5}]^2\})$, then $ITPU(\overline{p_1}, T_{-1}, \{[p_1]^1, [p_2]^2, [\overline{p_3}]^2, [p_4]^1, [\overline{p_5}]^2\})$ with final tentative assignment $\{[\overline{p_1}]^2, [p_2]^2, [\overline{p_3}]^2, [p_4]^1, [\overline{p_5}]^2\}$)

Note we have not discovered all unit clause consequences of $T_{-1} \cup \{[\overline{p_5}p_2]\}$.

3.1 Binary Resolution Closure

We can detect more unit clause consequences by computing the resolution closure of the binary clause information [Leeuwen, 1990], and thus possibly discover more variables whose values must be fixed by the binary clause sub-problem.

Resolution closure on the binary clauses, ensures that all binary consequences of the binary clauses appear explicitly in the theory T. This may be able to detect literals that must be *true*, for example, given the $T = \{[pq], [\overline{q}r], [p\overline{r}]\}$, the resolution closure adds $\{[pr], [\overline{q}p], [p]\}$ and detects that p must be *true*. The procedure BinResClos given in Figure 4, assumes that

procedure BinResClos($[xy]$,T,A)
  $B := \emptyset$;
  foreach $[z\overline{x}] \in T$ {
    $B := B \cup \{[zy]\}$;
    foreach $[\overline{y}w] \in T$
      $B := B \cup \{[zw]\}$; }
  foreach $[\overline{y}w] \in T$
    $B := B \cup \{[xw]\}$;
  $T := T \cup B$;
  $(T, A) :=$ IncPropUnit(T,A);
  return (T, A);
end

Figure 4: Binary resolution closure algorithm for 2SAT.

T is resolution closed for all binary clauses except the new one, and adds the resolution closure for this new clause. To incorporate binary resolution closure in IncBinSat we add the line

$$(T, A) := \text{BinResClos}([xy], T, A);$$

into IncBinSat at the indicated point in Figure 3.

Example 3 *Consider executing the problem given in Example 2. We assume T_{-1} is resolution closed, so $T_{-1} = \{[\overline{p_1}\overline{p_2}], [\overline{p_1}p_3], [\overline{p_1}p_4], [\overline{p_1}p_5], [p_2p_3], [p_2p_4], [p_2p_5], [\overline{p_3}p_4], [\overline{p_3}p_5]\}$, The addition of the new clause $[\overline{p_5}p_2]$, generates transitive consequences $B = \{[\overline{p_1}p_2], [\overline{p_1}], [p_2], [\overline{p_3}p_2], [\overline{p_3}\overline{p_1}]\}$.*

The description in Figure 4 is deliberately simple. In fact the only purpose of the resolution closure step is to discover new unit clause consequences $[w] \in B$. The new binary clauses in B are only used for this purpose. In the implementation they are labelled separately to avoid redundant computations in IncPropUnit and IncTempPropUnit.

3.2 Krom Subsumption

Krom subsumption is a case of resolution which is guaranteed not to increase the size of the theory T. It can detect new binary clause information not detected by binary resolution closure which in turn can lead to new unit clause information.

The addition of Krom subsumption to our incremental satisfiability procedure is reasonably straightforward to express, although it will complicate the data structures required during the Davis-Putnam procedure. The procedure for Krom Subsumption resolution for a new binary clause $[xy]$ is shown in Figure 5. When used in conjunction with binary resolution closure then each binary clause generated also generates a call to KromSub.

4 Search strategy

Davis-Putnam style algorithms are based on searching for a solution, by setting each variable to *true* or *false* in a backtracking search. During each iteration, the procedure selects a variable p and generates two sub-formulas by adding one of the unit clauses $[p]$ and $[\overline{p}]$ to the current theory, effectively

```
procedure KromSub([xy],T,A)
    K := ∅;
    foreach ([x̄yC] ∈ T {
        T := (T − {[x̄yC]}) ∪ [yC];
        if |C| = 1 then
            /* new binary clauses emerge */
            K := K ∪ [yC]; }
    foreach ([xȳC] ∈ T {
        T := (T − {[xȳC]}) ∪ [xC];
        if |C| = 1 then
            /* new binary clauses emerge */
            K := K ∪ [xC]; } }
    foreach C ∈ K
        (T, A) := IncBinSat(C,T,A);
    return (T, A);
end
```

Figure 5: Krom subsumption resolution algorithm for binary clause $[xy]$.

setting the variable p to each of the two possibilities, *true* or *false*. Backtrack algorithms differ in the way they select which variable to set at each iteration [Gu et al., 1997].

A simple but effective variable selection strategy is to at each stage examine the original clauses in T that are not redundant with respect to the unit clauses currently in T. We choose the variable to branch on which has the most occurrences in these not necessarily satisfied clauses. In this manner we choose variables which are likely to satisfy many clauses or create many shorter clauses.

If we make use of the IncBinSat algorithm we have much more information available in order to make a variable selection. The temporary (and permanent) assignments give a value to each variable. We can straightforwardly determine which (non-binary) clauses are not satisfied by this valuation. If none are unsatisfied then we are finished, otherwise we simply select a literal from an unsatisfied clause as the literal to branch on. A full description of the search procedure is shown in Figure 6.

Initially the algorithm propagates any consequences of unit and binary clauses. Next we build an initial tentative assignment for all remaining free variables (which don't occur in any unit or binary clause). We simply select the form of the variable that occurs more often in the theory T. Then the search begins.

Search begins by selecting a variable to branch on using either SelectV, the default branching rule, or SelectC, using an unsatisfied clause. The use of SelectC implicitly requires that temporary unit propagation is being performed in order to be meaningful. Once we have selected a variable to branch on we set it to its two possible values. Note that in the case of using SelectC we first set the variable to the value appearing in the clause (negating its current temporary value). The procedure is performed recursively.

```
Procedure SATB(T)
    A := ∅;
    /* Preprocess unit clauses */
    (T, A) := IncPropUnit(T, A);
    if (□ ∈ T) then return false;

    /* Preprocess binary clauses */
    let B be all the binary clauses in T;
    foreach C ∈ B
        /* Process the binary clauses incrementally */
        (T, A) := IncBinSat(C,T,A);
        if (□ ∈ T) then return false;

    /* assign the unset variables in vars(T) */
    foreach p ∈ vars(T) such that [p] ∉ T ∪ A
            and [p̄] ∉ T ∪ A
        if occur(p, T) ≥ occur(p̄, T) then
            A := A ∪ {[p]};
        else A := A ∪ {[p̄]};
    /* Start branching */
    current_time := 0;
    (T, A) := Search(T,A);
    if (□ ∉ T) then return true;
    else return false;
end

Procedure Search(T0,A0)
    x := Select(T0,A0);
    if x ≠ ⊥ then {
        save_current_time := current_time;
        (T1, A1) := IncPropUnit(T0 ∪ {[x]}, A0);
        if (□ ∉ T1) then {
            (T2, A2) := Search(T1, A1);
            if (□ ∉ T2) then return (T2, A2); }
        current_time := save_current_time;
        (T1, A1) := IncPropUnit(T0 ∪ {[x̄]}, A0);
        if (□ ∉ T1) then {
            (T2, A2) := Search(T1, A1);
            if (□ ∉ T2) then return (T2, A2);
            else return ({□}, ∅); } }
    else return (T0, A0);
end

Procedure SelectC(T0,A0)
    if (∃C ∈ T0 such that C ∩ {x | [x] ∈ T0 ∪ A0} = ∅) {
        /* C is not satisfied by the current setting */
        let C be of the form [xC'];
        return x; }
    return ⊥;
end

Procedure SelectV(T0,A0)
    T' := T0 − {C | C ∈ T0, ∃[x] ∈ T0, x ∈ C};
    x := ⊥; min := |T'| + 1;
    foreach p ∈ vars(T')
        count := |{C | C ∈ T', {p, p̄} ∩ C ≠ ∅}|;
        if (count > min) {
            x := p; min := count; }
    return x;
end
```

Figure 6: SATB

5 Data Structures

In order to support 2SAT satisfiability the SATB algorithm requires the ability to quickly recognize (a) when a new unit clause results from unit resolution (just like any Davis-Putnam style procedure), (b) when a new binary clause results from unit resolution, and (c) when a clause is not satisfied by the current temporary assignment. In order to support binary resolution closure the algorithm additionally requires the ability to recognize (d) when a new unit clause results from binary resolution closure. In order to support Krom subsumption resolution the algorithm additionally requires the ability to recognize (e) when a new binary clause results from Krom subsumption.

We use the following (standard) fixed data structures to represent the problem.

- An array of clauses (lists of literals), to represent the theory.
- For each literal a list of clause numbers in which the literal occurs (to enable fast unit propagation).
- A two dimensional array of lists of clause numbers which contain both (indexed) literals (to support Krom subsumption)

We have standard mutable data structures

- An array of permanent values: *true*, *false*, *unknown* representing the clauses $[x]$ occurring in T.
- For each clause (number) a *remaining size* after unit propagations. Whenever we add a new permanent unit clause $[x]$ we subtract 1 from the size of all clauses in the list for $[\overline{x}]$.

The first two requirements (a) and (b) are supported by the remaining size data structure. When the remaining size reaches 2 we have discovered a (possibly new) binary clause.

We use the following mutable data structures in addition for SATB:

- An array of temporary values: *true* or *false* representing the clauses $[x]$ occurring in A.
- For each clause (number) a *tcount* (the number of variables contained in the clause that are identical to the current temporary assignments). Whenever we add a new permanent unit clause $[x]$, we subtract 1 from the *tcount* of all clauses in the list for $[\overline{x}]$; whenever a temporary value for x changes, we subtract 1 from the *tcount* of all clauses in the list for $[x]$ and add 1 to the *tcount* of all clauses in the list for $[\overline{x}]$.
- For each literal a count of the number of original clauses it appears in which are not known to be satisfied. Whenever we perform unit propagation we visit each clause containing the new unit clause and, if it were not already satisfied, subtract one from the count of all literals occurring in the clause. This data structure is only to support the default variable selection strategy.
- For each literal, a list of partner literals in binary clauses occurring in T originally or by unit resolution, implemented as an array and size. For example T_0 (from Example 1) generates a list $[p_3, \overline{p_5}]$ for the literal p_2. This is the data structure used by IncPropUnit and IncTempPropUnit.
- A matrix of all binary clauses in T whether via unit resolution, or resolution closure. The set represented is resolution closed. This data structure is used for resolution closure, and determining whether a new binary clause created by unit propagation needs to be added to the previous data structure or is redundant with respect to this.

In order to support Krom subsumption we use the matrix of clause numbers to look up candidates for Krom subsumption resolution. If Krom subsumption applies we subtract 1 from the remaining size of the clause. Note that unit resolution does not change the clause where unit resolutions occur, it simply reduces the size, and when this reaches 2 calculates the remaining binary clause. The same approach to Krom subsumption is incorrect. For example if we use binary clause $[xy]$ to simplify $[\overline{x}yzt]$ to $[yzt]$ and only reduced the remaining size to 3 we might later believe that the clause $[xz]$ could apply to the clause to reduce its size to 2. In order to maintain eliminated literal information by Krom subsumption we modify the order of the literals in the clause so that the eliminated literal appears last. Then simply restoring the remaining size restores the clause. For example the subsumption above causes the clause to change form to $[yzt\overline{x}]$. This together with the remaining size 3 indicates the clause remaining.

During the search we need to restore the mutable data structures to a previous state. To do so we keep a trail of changes. Whenever a call to Search is made we add an entry *choicepoint* to the trail indicating a backtrack point. Whenever a new permanent value $[x]$ for variable p is created we add an entry $perm([x])$ to the trail. Whenever a temporary value for p first changes after a choicepoint we add an entry $temp(p)$ if p changes to *false* and $temp(\overline{p})$ if it changes to *true*, i.e. we store the old temporary value of p. Whenever a new binary clause $[xy]$ is detected via unit resolution or Krom subsumption we add an entry $new([xy])$. Whenever a new binary clause $[xy]$ is detected via resolution closure we add an entry $res([xy])$.

In order to restore to the previous state we run back through the trail of entries removing entries and handling them until we remove a *choicepoint* entry. For an entry $perm([x])$ we add 1 to the remaining size of clauses in the list for $[\overline{x}]$, and reset the permanent value of the variable in x to *unknown*; also we restore the temporary value of x, the time stamp of time stamp $time([x])$, and the related *tcount*s value. For an entry $temp(x)$ we reset the temporary value of the variable in the list for x appropriately. We subtract 1 from the *tcount* of all clauses in the list of $[x]$ and add 1 to the *tcount* of all clauses in the list of $[\overline{x}]$. Again, the time stamp $time([x])$ has to be restored. For each entry $new([xy])$ we subtract 1 from the counts of the partner lists for x and y (since this will be the last one added), and remove $[xy]$ from the matrix of binary clauses. For each entry $res([xy])$ we remove $[xy]$ from the matrix of binary clauses. All of these trail actions are unit time.

In order to support Krom subsumption for each entry $new([xy])$ or $res([xy])$ we check whether they causes Krom subsumption steps and add one to remaining size of literals that were modified.

6 Experimental evaluation

SATB is written in standard C. It takes the standard conjunctive normal form (CNF) file as its input. All the experiments described here were conducted on a 300MHz Alpha Server 8400 running Digital Unix 3.2F with 8GB memory. However, it is limited to 90MB memory for each user. We perform 20 runs for each problem and report the average processing time as the result in the following tests.

We examine the performance of SATB on the benchmark groups AIM, Parity-8, flat graph colouring and uniform Random-3-SAT (these benchmarks are available on DIMACS web page [SATLIB, 2001]. The AIM family are highly structured ternary clause problems, which have either exactly one or no solution. The Parity problems are again highly structured problems with a significant number of binary clauses in the original formulation. The flat graph colouring problems are made up of a large fraction of binary clauses in each instance. The uniform random-3-SAT problems consist of only ternary clauses in each instance, and the clause-variable ratio of these benchmark are 4.3 which is proved to be hard.

6.1 Comparing the possibilities of using 2SAT information

In the first experiment we examine each of the possible uses of 2SAT information for improving the Davis-Putnam search: maintaining a 2SAT solution (s), binary resolution closure (c) and Krom subsumption (k). We compare each combination on the AIM-100 problems from the SATLIB benchmark suite.

The comparison is shown in Table 1, where we illustrate each possibility by a code showing which optimizations are used (it is straightforward to remove the 2SAT solution maintenance from IncBinSat by simply replacing the code for IncTempPropUnit by code that immediately returns). We consider the simple variable selection strategy SelectV for all possibilities and the unsatisfied clause strategy SelectC when the 2SAT solution is available (the optimizations include s). We give the average number of branches and times (in milliseconds) for finding a solution for the AIM-100 problems which involves 16 single-solution instances.

From the results of Table 1 we can easily see that binary resolution closure and Krom subsumption reduce the number of branches required and the search time. The use of 2SAT solution maintenance (s) is not beneficial when used in conjunction with one of these techniques when we use default variable selection. The only possible gain is when we determine

	default		clause	
Solvers	Ave. Branches/Time			
	>1,000,000	>10 min	—	—
s	4899.1	1568.0	9.9	8.5
c	99.8	52.9	—	—
k	94.4	45.1	—	—
sc	99.3	53.9	8.9	5.3
sk	84.8	43.1	3.7	12.5
ck	41.5	48.0	—	—
sck	41.5	53.9	1.6	13.7

Table 1: 2SAT solving methods on AIM-100 benchmarks

Benchmark			sc		sk		sck	
Name	n	m	Branches/Time					
par8-1-c	64	254	2	0.8	2	4.2	1	15.0
par8-1	350	1149	13	9.2	8	18.4	7	69.2
par8-2-c	68	270	4	0.8	2	3.3	1	18.3
par8-2	350	1157	13	11.7	11	23.3	9	117.5
par8-3-c	75	298	2	0.8	2	2.5	1	15.0
par8-3	350	1171	32	20.8	14	31.6	10	168.3
par8-4-c	67	266	53	14.2	22	15.9	7	41.7
par8-4	350	1155	11	8.3	4	15.9	4	62.6
par8-5-c	75	298	28	11.7	25	20.0	14	139.1
par8-5	350	1171	15	14.2	6	20.0	4	99.2

Table 2: 2SAT solving methods on Parity-8 benchmarks

Suite			sc		sk		sck	
Name	n	m	Ave. Branches/Time					
uf-20	20	91	3.3	0.8	3.0	0.4	1.4	2.2
uf-50	50	218	13.6	5.3	16.8	11.9	7.2	21.8
uf-75	75	325	41.5	28.5	57.1	29.5	24.2	70.0
uf-100	100	430	164.2	122.6	240.5	138.0	110.1	292.6

Table 3: 2SAT solving methods on uniform random 3SAT benchmarks

that the 2SAT solution solves the entire problem before fixing each variable. This does occur, as illustrated by the reduction in branches from k to sk.

The advantage of a 2SAT solution becomes clear when we compare the unsatisfied clause variable selection strategy that is enabled by having the solution. Clearly this greatly improves upon the simple variable selection strategy. Although the complete combination sck reduces the number of branches systematically when using unsatisfiable clause variable selection, the overhead of maintaining information for Krom subsumption negates the advantage in search. The most time efficient combination is sc.

To verify this comparison we also examined the tradeoffs between the possibilities sc, sk and sck for the Parity-8 benchmarks and uniform random 3SAT using the unsatisfiable clause variable selection. The results are shown in Tables 2 and 3 respectively.

Although the sc combination leads to more branching, it almost uniformly improves upon the sk and sck combinations in time because of the overhead in Krom subsumption. Given these experiments, the remainder of the paper considers SATB using 2SAT solution maintenance (via temporary unit propagation) and binary resolution closure, using the unsatisfied clause variable selection strategy.

6.2 Comparing against state of the art SAT solvers

To illustrate the performance of SATB we compare against two state of the art SAT solvers, SATO [Zhang and Stickel, 2000] and EQSATZ [Li, 2000]. SATO and EQSATZ are acknowledged as very efficient for solving SAT problems [SATLIB, 2001]. SATO use a highly optimized *trie* data structure, together with a special search strategy to make the Davis-Putnam search extremely efficient. EQSATZ detects set of original clauses that can be replaced by equivalence clauses (exclusive ors) and represents and treats these clauses specially. It also employs a powerful, but expensive, look-ahead heuristic to maximize the reduction of search space when branching. Hence EQSATZ usually has a very low number of branches.

In the following tables, n denotes the number of distinct variables of the formula T, and m the number of clauses.

6.2.1 Performance on AIM problems

The AIM instances are all generated with a particular Random-3-SAT instance generator [Asahiro et al., 1996]. All the yes-instances have exactly one solution. These are notably difficult problems for local search methods such as GSAT, as well as complete Davis-Putnam search methods.

The results in Table 4, show that while EQSATZ always finds the solution "without search" by using its expensive look-ahead branching scheme, the cost is prohibitive compared to SATO and SATB. Only for the instance aim_200_3_4_yes2 does the reduced search payoff. Even though there is no initial 2SAT information, the structured nature of the problem means that 2SAT clauses are generated quickly in the search. This means SATB usually does very well compared to SATO, both in terms of branches and time. The very different search strategies sometimes lead to cases where one improves over the other by a significant degree.

Benchmark		SATO		EQSATZ		SATB	
Name	m	Branches/Time					
aim_50_1_6_yes1	80	6	3.3	1	40.8	6	0.0
aim_50_1_6_yes2	80	3	3.7	1	30.0	2	0.0
aim_50_1_6_yes3	80	4	3.9	1	18.4	3	0.0
aim_50_1_6_yes4	80	3	3.4	1	16.7	1	0.0
aim_50_2_0_yes1	100	7	3.8	1	16.7	1	0.0
aim_50_2_0_yes2	100	10	4.1	1	33.3	3	0.0
aim_50_2_0_yes3	100	6	4.2	1	18.4	1	0.0
aim_50_2_0_yes4	100	4	4.8	1	17.5	3	0.0
aim_50_3_4_yes1	170	19	6.9	1	25.8	9	0.0
aim_50_3_4_yes2	170	8	6.4	1	57.5	4	0.8
aim_50_3_4_yes3	170	11	6.4	1	35.8	6	0.8
aim_50_3_4_yes4	170	10	6.0	1	33.3	7	0.8
aim_50_6_0_yes1	300	6	7.2	1	530.0	4	1.7
aim_50_6_0_yes2	300	5	7.6	1	618.4	3	2.5
aim_50_6_0_yes3	300	5	6.6	1	149.2	4	4.2
aim_50_6_0_yes4	300	4	7.1	1	1325.8	4	0.8
aim_100_1_6_yes1	160	5	6.0	1	66.7	3	0.0
aim_100_1_6_yes2	160	1	4.8	1	75.8	1	0.8
aim_100_1_6_yes3	160	10	6.5	1	74.2	2	0.0
aim_100_1_6_yes4	160	6	6.3	1	71.7	1	0.0
aim_100_2_0_yes1	200	64	12.3	1	60.9	5	3.3
aim_100_2_0_yes2	200	11	6.8	1	66.7	8	7.5
aim_100_2_0_yes3	200	7	6.3	1	83.3	10	4.2
aim_100_2_0_yes4	200	13	6.2	1	65.9	2	3.3
aim_100_3_4_yes1	340	26	11.3	1	33.3	36	11.7
aim_100_3_4_yes2	340	49	14.8	1	33.3	6	0.8
aim_100_3_4_yes3	340	47	15.0	1	33.3	16	6.7
aim_100_3_4_yes4	340	55	16.2	1	35.0	44	14.2
aim_100_6_0_yes1	600	5	11.8	1	118.4	5	15.0
aim_100_6_0_yes2	600	4	12.4	1	129.1	7	12.5
aim_100_6_0_yes3	600	3	11.6	1	100.0	4	5.8
aim_100_6_0_yes4	600	5	12.2	1	207.5	2	5.0
aim_200_1_6_yes1	320	8	8.6	1	134.1	2	0.0
aim_200_1_6_yes2	320	4	7.3	1	265.9	7	5.0
aim_200_1_6_yes3	320	11	10.8	1	135.8	12	9.2
aim_200_1_6_yes4	320	12	12.7	1	101.7	15	7.5
aim_200_2_0_yes1	400	43	15.2	1	132.5	210	53.3
aim_200_2_0_yes2	400	25	12.9	1	100.0	10	5.0
aim_200_2_0_yes3	400	27	12.2	1	220.0	11	7.5
aim_200_2_0_yes4	400	41	14.9	1	157.5	19	6.7
aim_200_3_4_yes1	680	116	40.9	1	61.7	22	22.5
aim_200_3_4_yes2	680	44	23.9	1	65.0	46	34.1
aim_200_3_4_yes3	680	252	86.5	1	60.9	107	142.5
aim_200_3_4_yes4	680	30	21.9	1	59.2	22	25.0
aim_200_6_0_yes1	1200	2	20.7	1	100.0	3	11.7
aim_200_6_0_yes2	1200	30	37.1	1	100.0	8	16.7
aim_200_6_0_yes3	1200	10	26.6	1	85.8	6	16.7
aim_200_6_0_yes4	1200	28	35.6	1	97.5	2	5.8

Table 4: Comparative performance on AIM benchmarks

6.2.2 Performance on Parity problems

Instances in the parity class are a propositional version of parity learning problems. The instances of parity problems have been shown to be rather hard for systematic as well as local search algorithms. For the Parity-8 benchmark, there are exact 32 binary clauses in each "compressed" formula (par8-i-c.cnf) while the number of binary clauses in the larger formulae (par8-i.cnf) ranges from 314 to 330.

The results in Table 5 show that while EQSATZ always has the minimum number of branches, it is also the slowest among the three solvers. SATB improves on SATO in terms of speed in all the bigger instances (par8-i.cnf). While in the "compressed" problems (par8-i-c.cnf) SATO is sometimes better when the number of branches significantly betters that of SATB. It can be explained that the larger formulae have more binary clauses and hence SATB has more opportunities to detect inconsistencies.

6.2.3 Performance on uniform random 3SAT problems

Uniform Random-3-SAT is a family of SAT problems obtained by randomly generating 3-CNF formulae. All the instances in this benchmark use the

Benchmark			SATO		EQSATZ		SATB	
Name	n	m	Branches/Time					
par8-1-c	64	254	3	5.2	3	33.3	2	0.8
par8-1	350	1149	18	21.0	7	66.7	13	9.2
par8-2-c	68	270	6	5.9	2	33.3	4	0.8
par8-2	350	1157	16	22.0	6	70.0	13	11.7
par8-3-c	75	298	5	7.2	2	33.3	2	0.8
par8-3	350	1171	25	24.0	10	80.0	32	20.8
par8-4-c	67	266	15	7.7	1	30.8	53	14.2
par8-4	350	1155	25	22.7	2	65.9	11	8.3
par8-5-c	75	298	11	8.3	3	42.5	28	11.7
par8-5	350	1171	8	19.1	6	67.5	15	14.2

Table 5: Comparative performance on Parity-8 benchmarks

clauses/variable ratio of around 4.3 which has been proved to be hard [Mitchell et al., 1992]. We use the suites *uf-n* where n is the number of variables for $n = 20, 50, 75, 100$. Here *uf-75* suite has 100 instances and the other three suites all have 1000 instances. In Table 6 we give the average values over each suite.

Suite			SATO		EQSATZ		SATB	
Name	n	m	Average Branches/Time					
uf-20	20	91	6.3	3.7	1.4	265.1	3.3	0.8
uf-50	50	218	19.3	7.8	4.3	543.8	13.4	5.3
uf-75	75	325	47.3	15.2	8.7	107.3	41.5	28.5
uf-100	100	430	131.9	36.9	15.3	56.1	164.2	122.6

Table 6: Comparative performance on uniform random 3SAT benchmarks

Random-3-SAT problems are unstructured and contain *no* 2SAT information at the very beginning. Thus SATB is not likely to achieve good performance on this family of benchmarks. But from Table 6 we can see that SATB still beats SATO on both branches and run time for the smaller cases (*uf-20* and *uf-50*).

As the problems become bigger, it is clear that the reduced search space pays off in EQSATZ. It appears that the large number of ternary clauses in bigger problems lead to equivalence clauses being detected by the inference rules applied in EQSATZ. SATB loses in bigger instances since it requires much more (basically random) search at the top of the search tree to get to a point where there is enough 2SAT information to get useful reduction in search space.

6.2.4 Performance on flat graph colouring problems

Graph Colouring is a well-known combinatorial problem from graph theory. Instances in this class contain a large fraction of binary clauses (since the constraint that two node-color combinations are inconsistent is a binary clause). We use the suites of flat (planarizable) 3-colorable graphs with 30 and 50 nodes from [SATLIB, 2001]. The *flat-30* suite contains 100 instances, while *flat-50* contains 1000 instances.

The results are shown in Table 7. For these instances there are few equivalent clauses for EQSATZ to take advantage of, and its lookahead strategy does not improve significantly over the search strategies of SATO and SATB. While SATO betters SATB in number of branches, the large amount of 2SAT information still leads to significant simplification, and SATB betters SATO in speed.

Benchmark			SATO		EQSATZ		SATB	
Name	n	m	Average Branches/Time					
flat-30	90	300	10.6	6.2	9.8	19.5	11.3	3.1
flat-50	150	545	15.3	9.9	11.2	28.5	19.0	6.1

Table 7: Comparative performance on flat graph colouring benchmarks

6.3 Look ahead variable selection

There are many ways to select the next branching variable in practice. Many are highly successful in reducing search space, for example, the look-ahead technique used in EQSATZ [Li, 2000], Jeroslow-Wang backtracking [Jeroslow and Wang, 1990] and shortest clause backtracking [Bitner and Reingold, 1975]. Deriving 2SAT information during search may improves the look-ahead style variable selection strategies since extra information is derived during the shallow look-ahead search.

We have performed some preliminary experiments to determine the effectiveness of adding look-ahead search in the style of EQSATZ to SATB. At every branch point, we branch in turn on every variable which does not yet have a permanent assignment. For each variable p we record the total number of unit propagations that occur (including those that occur as a result of binary resolution closure) when we set it to *true* (i.e. add $[p]$) and set it to $false$ (i.e. add $[\overline{p}]$). We then actually branch on the variable which results in the largest number of unit propagations to extend.

Table 8 shows the effect of the lookahead strategy in the SATB solver. The lookahead strategy usually reduces the number of branches to 1. Unfortunately sometimes at the top of the tree there is little 2SAT information for the lookahead strategy to take advantage of, the lookahead strategy makes a bad decision, which leads to very large search (aim-100-2_0-yes1-2). The overhead of lookahead is even more significant when deriving new 2SAT information, and hence the lookahead approach never competes with the simpler unsatisfiable clause strategy.

There is significantly more information that the lookahead strategy could possibly determine such as unit propagations and binary clauses that result from both adding $[p]$ and adding $[\overline{p}]$. We can safely add this information without branching on p. We intend to investigate more advanced lookahead selection strategies that derive this information.

7 Conclusion and Future Work

SATB has taken a next step in a promising research direction initiated by Alvaro. In the past, 2SAT algorithms were generally only used as a preproces-

Instance			Clause		Lookahead	
Name	n	m	Branches/Time			
aim100-1_6-yes1-1	100	160	3	0.0	1	0.0
aim100-1_6-yes1-2	100	160	1	0.8	1	17.0
aim100-1_6-yes1-3	100	160	2	0.0	1	17.0
aim100-1_6-yes1-4	100	160	1	0.0	1	17.0
aim100-2_0-yes1-1	100	200	5	3.3	5	133.0
aim100-2_0-yes1-2	100	200	8	7.5	55	2200.0
aim100-2_0-yes1-3	100	200	10	4.2	1	17.0
aim100-2_0-yes1-4	100	200	2	3.3	1	17.0
aim100-3_4-yes1-1	100	340	36	11.7	1	117.0
aim100-3_4-yes1-2	100	340	6	0.8	3	183.0
aim100-3_4-yes1-3	100	340	16	6.7	1	167.0
aim100-3_4-yes1-4	100	340	44	14.2	1	200.0
aim100-6_0-yes1-1	100	600	5	15.0	1	67.0
aim100-6_0-yes1-2	100	600	7	12.5	1	67.0
aim100-6_0-yes1-3	100	600	4	5.8	1	67.0
aim100-6_0-yes1-4	100	600	2	5.0	1	133.0

Table 8: look-ahead and clause-order branching strategies

sor to simplify general SAT formulae. SATB goes further by examining newly deduced binary clauses either from unit resolution or binary resolution closure, and using a solution to the 2SAT subproblem to direct the search. The only other work we are aware of that used newly derived 2SAT information is [Van Gelder and Tsuji, 1995]. Here we extend their approach by using an incremental version of the 2SAT satisfaction algorithm of Alvaro del Val [Alvaro del Val, 2000]. We explore the range of possibilities of using 2SAT information and conclude that using Krom subsumption resolution does not appear to be worthwhile.

There are many directions for further improving SATB. The naïve data structures used in SATB may be refined to achieve higher efficiency. We should investigate more intelligent branching strategies, and in particular the more complex lookahead strategy discussed in the previous section. It would be interesting to explore if deriving other kinds of information, such as equivalence clauses as in EQSATZ, or equivalence of literals can be used to improve the performance of SATB. As usual the requirement for efficient data structures and incremental algorithms to maintain such information is crucial for the success of such approaches.

References

[Alvaro del Val, 2000] Alvaro del Val (2000). On 2SAT and renamable horn. In *AAAI'00 Proceedings of the 7th (U.S.) National Conference on Artificial Intelligence*, pages 343–348.

[Asahiro et al., 1996] Asahiro, Y., Iwama, K., and Miyano, E. (1996). Random generation of test instances with controlled attributes. In Johnson, D. S. and Trick, M. A., editors, *Cliques, Colouring, and Satisfiability: The Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 377–394.

[Aspvall et al., 1979] Aspvall, B., Plass, M. F., and Tarjan, R. E. (1979). A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–123.

[Bitner and Reingold, 1975] Bitner, J. R. and Reingold, E. M. (1975). Backtracking programming techniques. *Communications of the Association for Computing Machinery*, 18(11):651–656.

[Brafman, 2000] Brafman, R. I. (2000). A simplifier for propositional formulas with many binary clauses. Technical report, Dept. of Computer Science, Ben-Gurion University.

[Buro and Kleini-Büning, 1992] Buro, M. and Kleini-Büning, H. (1992). Report on a SAT competition. Technical Report FB-17–, Mathematik/Informatik, Universität Paderborn.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM symposium on the Theory of Computing*, pages 151–158.

[Dalal and Etherington, 1992] Dalal, M. and Etherington, D. W. (1992). A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44:173–180.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5(7):394–397.

[Even et al., 1976] Even, S., Itai, A., and Shamir, A. (1976). On the complexity of timetable and multi-commodity flow problems. *SIAM Journal of Computing*, 5(4):691–703.

[Gu et al., 1997] Gu, J., Purdom, P. W., Franco, J., and Wah, B. W. (1997). Algorithms for Satisfiability (SAT) problem: A survey(invited paper). In *Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem*, volume 35, pages 19–152.

[Jeroslow and Wang, 1990] Jeroslow, R. E. and Wang, J. (1990). Solving propositional satisfiability problems. *Annals of mathematics and AI*, 1:167–187.

[Larrabee, 1992] Larrabee, T. (1992). Test pattern generation using Boolean satisfiability. *IEEE Trans. on Computer-Aided Design*, 11(1):4–15.

[Leeuwen, 1990] Leeuwen, J. V. (1990). Handbook of theoretical computer science. volume A, pages 539–542.

[Li, 2000] Li, C. M. (2000). Integrating equivalency reasoning into Davis-Putnam procedure. In *AAAI/IAAI 2000*, pages 291–296.

[Mitchell et al., 1992] Mitchell, D., Selman, B., and Levesque, H. (1992). Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on AAAI-92*, pages 459–465.

[SATLIB, 2001] SATLIB (2001). http://www.intellektik.informatik.tu-darmstadt.de/satlib/. Version 1.4.4, 01/05/11.

[Van Gelder and Tsuji, 1995] Van Gelder, A. and Tsuji, Y. K. (1995). Satisfiability testing with more reasoning and less guessing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 26, pages 559–586. American Mathematical Society.

[Zhang and Stickel, 2000] Zhang, H. and Stickel, M. E. (2000). Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296.