A decision-making procedure for resolution-based SAT-solvers

Eugene Goldberg

Cadence Research Labs, USA, 2150 Shattuck Ave.,10th floor, Berkeley, California, 94704, phone: 1-510-647-2825, fax:1-510-647-2801, egold@cadence.com

Abstract. We describe a new decision-making procedure for resolution-based SAT-solvers called Decision Making with a Reference Point (DMRP). In DMRP, a complete assignment called a reference point is maintained. DMRP is aimed at finding a change of the reference point under which the number of clauses falsified by the modified point is smaller than for the original one. DMRP makes it possible for a DPLL-like algorithm to perform a "local search strategy". We describe a SAT-algorithm with conflict clause learning that uses DMRP. Experimental results show that even a straightforward and unoptimized implementation of this algorithm is competitive with SAT-solvers like BerkMin and Minisat on practical formulas. Interestingly, DMRP is beneficial not only for satisfiable but also for unsatisfiable formulas.

1 Introduction

Resolution based SAT-solvers have gained great popularity due to their ability to solve very large practical CNF formulas. An important contributor to this success is conflict driven decision making (CDDM) introduced in [17] and further developed in BerkMin [7], Minisat [3], Siege and other SAT-solvers. CDDM takes into account the history of conflicts thus forcing the SAT-solver to explore the parts of the search space that have not been visited before.

Despite the obvious success of CDDM, still there are many directions to explore. In this paper, we introduce a resolution based SAT-solver whose decision making procedure employs a complete assignment further referred to as **a reference point**. We will refer to this procedure as Decision Making with a Reference Point **(DMRP)**. (We will refer to the SAT-solver employing DMRP that we describe in this paper as **DMRP-SAT**.)

The main idea of DMRP is as follows. Let F be the CNF formula to be solved. Let $\boldsymbol{p}$ be a reference point and $M(\boldsymbol{p})$ be the set of clauses of F that are falsified by $\boldsymbol{p}$. DMRP-SAT picks a clause C of $M(\boldsymbol{p})$ and tries to find a modification $\boldsymbol{p'}$ of $\boldsymbol{p}$ that satisfies C and does not falsify any clauses of F that are not in $M(\boldsymbol{p}) \setminus \{C\}$. In other words, $M(\boldsymbol{p'}) \subset M(\boldsymbol{p})$.

Importantly, the search of the point $\boldsymbol{p'}$ above is done by a regular DPLL-like procedure with conflict clause learning. After $\boldsymbol{p'}$ is found, it becomes a new reference point and DMRP-SAT performs a complete restart. In our previous paper [6], we described the resolution-based SAT-solver called FI that operates

on complete assignments. One can view the decision making procedure of FI as a variation of CDDM. Similarly to CDDM and decision making of FI, DMRP also gives some preference to recently derived conflict clauses. At the same time, DMRP is not just a variation of CDDM.

DMRP makes it possible for a SAT-solver to monotonically reduce the number of clauses falsified by the current reference point. So, in a sense, DMRP-SAT combines the features of algorithms based on the DPLL procedure [2] and local search SAT-algorithms pioneered in [20, 21]. The strategy of reducing the number of clauses falsified by a complete assignment has been successfully applied by local search procedures to various classes of satisfiable formulas with no (or "little") structure like random CNF formulas. For structured satisfiable formulas, DPLL based SAT-solvers are usually more successful due to conflict clause learning and Boolean Constraint Propagation. Our results imply that local search strategy can be successfully applied to structured formulas as well. Interestingly, DMRP-SAT works very well not only for satisfiable formulas (which is somewhat expected) but also for unsatisfiable ones.

Currently, the main drawback of DMRP in comparison to CDDM is that DMRP is more expensive. The reason is that DMRP has to maintain a particular set of clauses that is updated after assigning/unassigning a variable. (Satisfying all the clauses of this set means that a new reference point is found that falsifies fewer clauses than the original one). However, our experiments show that due to high quality of decision making, even a straightforward unoptimized implementation of DMRP-SAT can be competitive with SAT-solvers like BerkMin and Minisat.

This paper is structured as follows. In Section 2, we introduce the idea of DMRP and give an example. Section 3 describes DMRP-SAT in more detail. In Section 4, DMRP-SAT is compared with other SAT-solvers. Experimental results are presented in Section 5. We give some conclusions in Section 6.

2 Main idea of DMRP

In this section, we describe the basic idea of Decision Making with a Reference Point (DMRP) that is implemented in the SAT-solver DMRP-SAT.

Let F be a CNF formula and $\boldsymbol{p}$ be a complete assignment to the variables of F. (For the sake of brevity, we will also call $\boldsymbol{p}$ **a point**.) A clause C of F is said to be **falsified** (**satisfied**) by $\boldsymbol{p}$ if $C(\boldsymbol{p}) = 0$ (respectively $C(\boldsymbol{p}) = 1$). Denote by $Vars(C)$ the set of variables of clause C. Denote by $Vars(\boldsymbol{y})$ the set of variables assigned in a partial assignment $\boldsymbol{y}$.

Definition 1. *Let $M(\boldsymbol{p})$ be the set of clauses of F falsified by $\boldsymbol{p}$. We will say that $\boldsymbol{p'}$ **recursively satisfies** a clause C of $M(\boldsymbol{p})$ with respect to **the reference point $\boldsymbol{p}$** if a) $C(\boldsymbol{p'}) = 1$; b) $M(\boldsymbol{p'}) \subset M(\boldsymbol{p})$.*

The use of term "recursively" is due to the fact that, given a reference point $\boldsymbol{p}$, when looking for the point $\boldsymbol{p'}$ above, DMRP-SAT first satisfies clause C, then satisfies "descendants" of clause C that get falsified after satisfying C and so on.

Note that if F is satisfiable, an assignment $\boldsymbol{p'}$ meeting the two conditions of Definition 1 always exists. (An assignment $\boldsymbol{p'}$ satisfying F recursively satisfies any clause C of F with respect to any reference point $\boldsymbol{p}$ falsifying C.) On the other hand, even if F is unsatisfiable , one may find an assignment $\boldsymbol{p'}$ recursively satisfying C if $|M(\boldsymbol{p})| > 1$. Then $M(\boldsymbol{p'}) \subset M(\boldsymbol{p})$ and $M(\boldsymbol{p'}) \neq \emptyset$.

The basic idea of DMRP is to look for a complete assignment recursively satisfying a clause by regular branching as in the DPLL-procedure.

Definition 2. *Let $\boldsymbol{y}$ be a partial assignment. Denote by **modify**($\boldsymbol{p}$,$\boldsymbol{y}$) the point obtained from $\boldsymbol{p}$ by flipping the assignments that are different in $\boldsymbol{p}$ and $\boldsymbol{y}$. (So assignments to Vars($\boldsymbol{y}$) in the point modify($\boldsymbol{p}$,$\boldsymbol{y}$) are the same as in $\boldsymbol{y}$.)*

DMRP-SAT looks for a partial assignment $\boldsymbol{y}$ such that $\boldsymbol{p'} = \mathit{modify}(\boldsymbol{p},\boldsymbol{y})$ recursively satisfies C. To make this search efficient, DMRP-SAT maintains a set $D(C,\boldsymbol{p},\boldsymbol{y})$ of clauses that one needs to satisfy before finding a point recursively satisfying C (see Definition 3). So if $D(C,\boldsymbol{p},\boldsymbol{y}) = \emptyset$, the point $\boldsymbol{p'} = \mathit{modify}(\boldsymbol{p},\boldsymbol{y})$ recursively satisfies C. DMRP-SAT implements DMRP using the following **greedy heuristic** aimed at making $D(C,\boldsymbol{p},\boldsymbol{y})$ empty. The next assignment is picked by DMRP-SAT so as to satisfy the largest number of clauses of $D(C,\boldsymbol{p},\boldsymbol{y})$.

Definition 3 (of the set $D(C,\boldsymbol{p},\boldsymbol{y})$). *If partial assignment $\boldsymbol{y}$ is empty, then $D(C,\boldsymbol{p},\boldsymbol{y}) = \{C\}$. Otherwise, $D(C,\boldsymbol{p},\boldsymbol{y})$ is defined as follows. A clause C' of F is in $D(C,\boldsymbol{p},\boldsymbol{y})$ iff 1) there is a variable $x_i \in Vars(\boldsymbol{y}) \cap Vars(C')$ that is assigned differently in $\boldsymbol{p}$ and $\boldsymbol{y}$; 2) C' is not satisfied by $\boldsymbol{y}$.*

Proposition 1. *Let F be a CNF formula. Let $\boldsymbol{p}$ be a complete assignment and C be a clause of $M(\boldsymbol{p})$. Let $\boldsymbol{y}$ be a partial assignment satisfying C. If $D(C,\boldsymbol{p},\boldsymbol{y}) = \emptyset$, then the complete assignment $\boldsymbol{p'}$=modify($\boldsymbol{p}$,$\boldsymbol{y}$) recursively satisfies the clause C with respect to the reference point $\boldsymbol{p}$.*

Proof. Assume the contrary, i.e. there is a clause C' of $M(\boldsymbol{p'})$ that is not in $M(\boldsymbol{p})$ and so $\boldsymbol{p'}$ does not satisfy C recursively. Suppose the set of variables A= $Vars(C')$ $\cap$ $Vars(\boldsymbol{y})$ is empty. Then, $C'(\boldsymbol{p'})$=0 implies $C'(\boldsymbol{p})$=0 and so C' is in $M(\boldsymbol{p})$. We have a contradiction. Now suppose that $A \neq \emptyset$. Then all the assignments of $\boldsymbol{y}$ to the variables of A have to falsify corresponding literals of C'. (Otherwise, $C'(\boldsymbol{p'}) = 1$). If all the variables of A are assigned identically in $\boldsymbol{y}$ and $\boldsymbol{p}$, then $C'(\boldsymbol{p}) = 0$ and so C' is in $M(\boldsymbol{p})$. Suppose that at least one variable of A is assigned differently in $\boldsymbol{y}$ and $\boldsymbol{p}$. Then, since $D(C,\boldsymbol{p},\boldsymbol{y})$ is empty, the clause C' has to be satisfied by some assignment in $\boldsymbol{y}$. So we have a contradiction again.

Note that $D(C,\boldsymbol{p},\boldsymbol{y})$= $\emptyset$ is only a sufficient condition. For example, even if $D(C,\boldsymbol{p},\boldsymbol{y}) \neq \emptyset$ but all the clauses of $D(C,\boldsymbol{p},\boldsymbol{y})$ are satisfied by the reference point $\boldsymbol{p}$, the complete assignment $\mathit{modify}(\boldsymbol{p},\boldsymbol{y})$ may recursively satisfy C. One can give another definition of $D(C,\boldsymbol{p},\boldsymbol{y})$ so that $D(C,\boldsymbol{p},\boldsymbol{y}) = \emptyset$ is also the necessary condition for $\mathit{modify}(\boldsymbol{p},\boldsymbol{y})$ to recursively satisfy C. However, in the current version of DMRP-SAT, to simplify computation of $D(C,\boldsymbol{p},\boldsymbol{y})$ we use Definition 3.

Example 1. Let F be the CNF formula specified by the following seven clauses: $C_1 = x_1 \vee x_2 \vee x_3$, $C_2 = x_3 \vee \overline{x_4} \vee x_5$, $C_3 = \overline{x_1} \vee \overline{x_6}$, $C_4 = \overline{x_1} \vee x_7$,

$C_5 = x_2 \vee \overline{x_5} \vee \overline{x_7}$, $C_6 = \overline{x_2} \vee \overline{x_4} \vee \overline{x_7}$, $C_7 = \overline{x_2} \vee x_6 \vee x_4$. Let $\boldsymbol{p}$=(x_1=0, x_2=0, x_3=0, x_4=1, x_5=0, x_6=0, x_7=0) be a reference point. The set $M(\boldsymbol{p})$ consists of clauses C_1 and C_2. In this example, we describe a run of *DMRP-solve* (see Figures 1 and 2) called by *DMRP-SAT* when looking for a point that recursively satisfies clause C_1. In this description, we use the terminology of decision levels [22]. Decision level number k consists of the decision assignment number k and all implied assignments derived in Boolean Constraint Propagation (BCP) caused by this decision assignment.

Initially, $Vars(\boldsymbol{y})$=$\emptyset$. So $D(C_1,\boldsymbol{p},\boldsymbol{y}) = \{C_1\}$. Suppose that *DMRP-SAT* chose variable x_1 to satisfy C_1 (the function *pick_lit* of Figure 1). That is $x_1 = 1$ is the first decision assignment made by *DMRP-solve*. Then the clause C_1 is removed from $D(C_1,\boldsymbol{p},\boldsymbol{y})$ (because it is satisfied by an assignment in $\boldsymbol{y}$). Only clauses C_3 and C_4 of F have literal $\overline{x_1}$. They are added to $D(C_1,\boldsymbol{p},\boldsymbol{y})$ because $\boldsymbol{p}$ and $\boldsymbol{y}$ have different assignments to x_1 and neither C_3 nor C_4 are satisfied by an assignment in $\boldsymbol{y}$. So for $\boldsymbol{y}$= $\{x_1$=1$\}$ the set $D(C_1,\boldsymbol{p},\boldsymbol{y})$ is equal to $\{C_3, C_4\}$.

At this point C_3 and C_4 become unit. After BCP, *DMRP-solve* derives x_6=0 (from clause C_3) and x_7=1 (from clause C_4) and removes C_3,C_4 from $D(C_1,\boldsymbol{p},\boldsymbol{y})$. Since assignment x_6=0 is the same in $\boldsymbol{y}$ and $\boldsymbol{p}$, no new clauses are added to $D(C_1,\boldsymbol{p},\boldsymbol{y})$ when $\boldsymbol{y}$ becomes (x_1=1,x_6=0). On the other hand, the variable x_7 is assigned differently in $\boldsymbol{y}$ and $\boldsymbol{p}$. Since x_7 is in C_5 and C_6 and they are not satisfied by $\boldsymbol{y}$, these clauses are added to $D(C_1,\boldsymbol{p},\boldsymbol{y})$. So after completing BCP of decision level 1, we have $\boldsymbol{y}$=(x_1=1, x_6=0, x_7 = 1), $D(C_1,\boldsymbol{p},\boldsymbol{y})$= $\{C_5,C_6\}$.

Suppose that *DMRP-solve* picks second decision assignment x_2=1 to satisfy C_5. Then clauses C_6 and C_7 become unit, and *DMRP-solve* derives opposite values of x_4 from them. This leads to a conflict. *DMRP-solve* derives conflict clause $C_8 = \overline{x_1} \vee \overline{x_2}$ and backtracks to decision level 1. At this level, $\boldsymbol{y}$ =(x_1=1, x_6=0, x_7 = 1) again and $D(C_1,\boldsymbol{p},\boldsymbol{y}) = \{C_5,C_6\}$. However, now *DMRP-solve* has to update $D(C_1,\boldsymbol{p},\boldsymbol{y})$ due to appearance of conflict clause C_8 by adding it to $D(C_1,\boldsymbol{p},\boldsymbol{y})$. ($C_8$ contains variable x_1 that is assigned differently in $\boldsymbol{y}$ and $\boldsymbol{p}$ and C_8 is not satisfied by $\boldsymbol{y}$.) So, $D(C_1,\boldsymbol{p},\boldsymbol{y}) = \{C_5,C_6,C_8\}$.

At decision level 1, the conflict clause C_8 becomes unit and *DMRP-solve* derives x_2=0 from it. Since x_2=0 agrees with $\boldsymbol{p}$, no new clauses need to be added to $D(C_1,\boldsymbol{p},\boldsymbol{y})$. At the same time, C_6 and C_8 are removed from $D(C_1,\boldsymbol{p},\boldsymbol{y})$ because they are both satisfied by x_2= 0. So $D(C_1,\boldsymbol{p},\boldsymbol{y}) = \{C_5\}$. *DMRP-solve* derives x_5=0 from C_5 and the latter is removed from $D(C_1,\boldsymbol{p},\boldsymbol{y})$. Since x_5=0 agrees with $\boldsymbol{p}$, no new clauses are added to $D(C_1,\boldsymbol{p},\boldsymbol{y})$. So, for the partial assignment $\boldsymbol{y}$=(x_1=1, x_6=0, x_7=1, x_2=0, x_5=0), $D(C_1,\boldsymbol{p},\boldsymbol{y})$ is empty. This means, that the clause C_1 is recursively satisfied by the assignment $\boldsymbol{p'} = modify(\boldsymbol{p},\boldsymbol{y})$ where $\boldsymbol{p'}$ = (x_1=1, x_2=0, x_3=0, x_4=1, x_5=0, x_6=0, x_7=1). It is not hard to check that indeed $C_1(\boldsymbol{p'})$=1 and $M(\boldsymbol{p'}) = \{C_2\}$ and so $M(\boldsymbol{p'}) \subset M(\boldsymbol{p})$. Now *DMRP-solve* performs a complete restart and picks $\boldsymbol{p'}$ as the next reference point.

3 Description of DMRP-SAT

In this section, we describe DMRP-SAT in more detail.

3.1 *DMRP-SAT* (high-level view)

```
DMRP-SAT(F)
{p=gen_ref_point(F);
  while (true)
    {C = pick_clause(M(p));
    lit = pick_lit(C,M(p));
    (ans,y)=DMRP-solve(F,C,lit,p);
    if (ans == unsat) return(unsat);
    if (ans == sat) return(sat);
    if (ans == literal) continue;
    if (ans == rec_sat)
      {p = modify(p,y);
      if (M(p) == ∅) return(sat);}
    if (ans == new_point)
      p =modify(p,y);}}
```

Figure 1. Pseudocode of *DMRP-SAT*

Pseudocode of *DMRP-SAT*(F) is shown in Figure 1. First, *DMRP-SAT* generates a reference point. This is done identically to initial point generation of *FI* [6]. "Decision" assignments are made by *gen_ref_point* in the order variables of F are numbered. A decision assignment is made to variable x_i only if it has not been already assigned by BCP performed after a previous decision assignment. The polarity of assignment to x_i is chosen to satisfy the largest number of clauses of F with variable x_i. After a decision assignment is made, BCP is performed. If a clause of F is falsified during BCP, it is added to $M(\boldsymbol{p})$.

```
DMRP-solve(F,C,lit,p)
{D(C,p,y) = {C};
  while (true)
    {if (D(C,p,y) == ∅)
      {restart(F);
      return(y,rec_sat);}
    make_assgn(F,lit,D(C,p,y));
    ans = BCP(F,D(C,p,y),p);
    if (ans == sat) return(sat);
    if (ans == conflict)
      {C* = gen_cnfl_clause(F);
      if (empty(C*) return(unsat);
      if (C* == unit) p=upd_pnt(p);
      if (C* == lit)
        {restart(F) ;
        return(literal); }
      add_clause(F,C*);
      backtrack(F);}
    else continue; // no conflict
    if (num_of_cnfl++ > THRESH)
      {restart(F);
      return(y,new_point);}
    if (num_of_cnfl > thresh)
      {restart(F);
      continue;} }}
```

Figure 2. Pseudocode of *DMRP-solve*

The main work is done by *DMRP-SAT* in the 'while' loop. First, *DMRP-SAT* picks a clause C of $M(\boldsymbol{p})$ to be recursively satisfied. If $M(\boldsymbol{p})$ contains conflict clauses, then the clause derived most recently is chosen as C. Otherwise, *DMRP-SAT* picks a clause of $M(\boldsymbol{p})$ that has a literal occurring most frequently among clauses of $M(\boldsymbol{p})$. Then a literal *lit* of C is chosen by the *pick_lit* procedure. Namely, it chooses the literal of C that occurs most frequently among clauses of $M(\boldsymbol{p})$. When looking for a complete assignment recursively satisfying clause C, the function *DMRP-solve* called next examines only points for which *lit* evaluates to 1.

Being a DPLL-like procedure with learning, *DMRP-solve* returns answer *unsatisfiable* if an empty clause is derived. If all clauses of F are satisfied by the current partial assignment $\boldsymbol{y}$, then *DMRP-solve* returns satisfiable. If *DMRP-solve* derives the literal $\overline{lit}$ it returns *literal*. This means, that clause C cannot by satisfied by setting literal *lit* to 1. Then *DMRP-SAT* starts a new iteration.

If $D(C,\boldsymbol{p},\boldsymbol{y})=\emptyset$ (where $\boldsymbol{y}$ is the current partial assignment), *DMRP-solve* returns *rec_sat* (C can be recursively satisfied). A new reference point *modify*($\boldsymbol{p}$,$\boldsymbol{y}$) is computed. If $M(\boldsymbol{p}) = \emptyset$, then $\boldsymbol{p}$ is a satisfying assignment. Otherwise, a new iteration of the 'while' loop is started. If the number of conflicts that occurred in *DMRP-solve* exceeds *THRESH*, *DMRP-solve* returns *new_pnt*. In this case, *DMRP-SAT* generates a new reference point *modify*($\boldsymbol{p}$,$\boldsymbol{y}$) (where $\boldsymbol{y}$ is the partial assignment of *DMRP-solve* when it encountered the last conflict).

3.2 DMRP-solve

The pseudocode of *DMRP-solve*(F,C,*lit*,$\boldsymbol{p}$) is shown in Figure 2. On the one hand, *DMRP-solve* is a regular DPLL-like SAT-solver with conflict clause learning. In the 'while' loop, it makes a decision assignment and then runs BCP. If after BCP, all clauses of F are satisfied, then *DMRP-solve* returns *satisfiable*. If a conflict is encountered during BCP, a conflict clause C^* is generated using the 1UIP scheme ([23]). If C^* is an empty clause, *DMRP-solve* returns *unsatisfiable*. Otherwise, C^* is added to the current CNF formula, *DMRP-solve* backtracks and a new iteration starts (unless C^* is equal to $\overline{lit}$, see below). If the number of conflicts that occurred since the last restart is larger than *thresh*, *DMRP-solve* restarts [8] (i.e. backtracks to decision level 0).

On the other hand, *DMRP-solve* has a few differentiating features. If conflict clause C^* is unit and the current reference point $\boldsymbol{p}$ falsifies C^*, then $\boldsymbol{p}$ is modified by flipping the value of the variable *Vars*(C^*). Besides, if C^* is unit and equal to $\overline{lit}$, *DMRP-solve* informs *DMRP-SAT* that such a literal was derived. (Here, *lit* is the literal of clause C to be satisfied recursively that was chosen by *pick_lit* of *DMRP-SAT*). For decision-making, *DMRP-solve* maintains the set $D(C,\boldsymbol{p},\boldsymbol{y})$ (see Definition 3). Before looking for a new decision assignment, *DMRP-solve* checks if $D(C,\boldsymbol{p},\boldsymbol{y})=\emptyset$. If so, it performs a restart and informs *DMRP-SAT* that clause C is recursively satisfied.

At the first decision level, *DMRP-solve* always makes the assignment satisfying the literal *lit* of C (and so satisfying C). At a level greater than 1, *DMRP-solve* picks the next decision assignment as follows. If the set $D(C,\boldsymbol{p},\boldsymbol{y})$ contains a conflict clause, the clause C' of $D(C,\boldsymbol{p},\boldsymbol{y})$ that was derived most recently is chosen. Then, *DMRP-solve* finds the literal of C' that is shared by the largest number of clauses of $D(C,\boldsymbol{p},\boldsymbol{y})$ and picks the assignment that satisfies this literal. If $D(C,\boldsymbol{p},\boldsymbol{y})$ does not contain conflict clauses, *DMRP-solve* makes the assignment satisfying the largest number of clauses from $D(C,\boldsymbol{p},\boldsymbol{y})$.

If the number of conflicts that occurred since *DMRP-solve* has been called is larger than *THRESH*, *DMRP-solve* performs a restart. Then *DMRP-solve* informs *DMRP-SAT* to generate the new reference point $\boldsymbol{p}$=*modify*($\boldsymbol{p}$,$\boldsymbol{y}$). Here $\boldsymbol{y}$ is the partial assignment *DMRP-solve* had when the last conflict occurred. The value of *THRESH* is larger than that of *thresh* used for restarts without changing the reference point.

3.3 Computation of $D(C,\boldsymbol{p},\boldsymbol{y})$

In the current implementation of *DMRP-solve*, set $D(C,\boldsymbol{p},\boldsymbol{y})$ is computed incrementally. Initially, $D(C,\boldsymbol{p},\boldsymbol{y}) = \{C\}$. When making an assignment $x_i = b$, $b \in \{0,1\}$ (either decision one or derived by BCP), *DMRP-solve* does the following. First it checks if reference point $\boldsymbol{p}$ has the same assignment $x_i=b$. If so, no new clauses are added to $D(C,\boldsymbol{p},\boldsymbol{y})$. Otherwise, *DMRP-solve* examines all the clauses of $D(C,\boldsymbol{p},\boldsymbol{y})$ in which the assignment $x_i=b$ sets a literal of x_i to 0. If such a clause is neither satisfied nor it is already in $D(C,\boldsymbol{p},\boldsymbol{y})$, it is added to $D(C,\boldsymbol{p},\boldsymbol{y})$. Then *DMRP-solve* removes from $D(C,\boldsymbol{p},\boldsymbol{y})$ all the clauses that are satisfied by $x_i=b$.

When undoing the assignment $x_i=b$ above (when backtracking), *DMRP-solve* does similar updates. First, it removes from $D(C,\boldsymbol{p},\boldsymbol{y})$ the clauses that were added to $D(C,\boldsymbol{p},\boldsymbol{y})$ due to assignment $x_i=b$. Second, it returns to $D(C,\boldsymbol{p},\boldsymbol{y})$ all the clauses that were removed because they got satisfied by $x_i=b$.

3.4 Brief discussion of DMRP and CDDM

Similar to conflict driven decision making (CDDM) introduced by Chaff, DMRP takes into account the history of conflicts. First, when picking a clause C of $M(\boldsymbol{p})$ to be satisfied recursively, DMRP gives preference to conflict clauses derived most recently. Second, next decision assignment is made to a variable of the most recently derived conflict clause C^* of $D(C,\boldsymbol{p},\boldsymbol{y})$ (if any).

At the same time, there are obvious differences. When picking next assignment, DMRP finds the literal of $Vars(C^*)$ with the largest occurrence in clauses of $D(C,\boldsymbol{p},\boldsymbol{y})$. So no preference is given to conflict clauses. Besides, no decay scheme is used for literal activity computation. So DMRP cannot be called just a variation of CDDM.

In our current implementation, DMRP is more expensive than CDDM. As we mentioned above, every time DMRP-SAT makes/unmakes an assignment (decision or implied one) it recomputes $D(C,\boldsymbol{p},\boldsymbol{y})$. So one of the directions for future research is to cut the cost of DMRP. A potential solution to this problem is to compute $D(C,\boldsymbol{p},\boldsymbol{y})$ approximately.

4 Background

In this section, we compare DMRP-SAT with other SAT-solvers. In this comparison we take into account the following four features: BCP, learning, maintaining a complete assignment, making restarts. Each of these features is arguably beneficial. BCP allows one to find "forced" assignments. Learning (e.g. conflict clause recording [22]) helps in pruning away big chunks of the search space. Maintaining a complete assignment provides some information about how far a SAT-solver is from a satisfying assignment [20, 21]. Besides, having a complete assignment can be used for (implicit) identification of small unsatisfiable sub-formulas [6]. Restarts [8] alleviate the problem of SAT-solver's getting stuck in a part of the search space that does not contain satisfying assignments. At the same time, we

do not claim that the more of these four features a SAT-solver has, the more advanced it is. For example, SAT-solvers that do not employ conflict clause learning (e.g. Satz [14]) work much better for random CNF formulas.

SAT-algorithms like GSAT[20] and WalkSat[21] (and many other local search algorithms [11]) operate on complete assignments and make restarts (in the sense that they pick a new initial complete assignment every once in a while). These algorithms work very well for some classes of formulas like satisfiable random formulas. However, lack of learning and BCP makes local search algorithms less efficient when applied to "highly structured" formulas. On the other hand, DPLL-like SAT-solvers like Grasp [22], SATO [24], Zchaff [17], BerkMin [7], Minisat[3], Siege and many others use learning and BCP. Most of them also employ restarts. These SAT-algorithms have been very successful in solving both satisfiable and unsatisfiable structured formulas. This success can be attributed to a) efficient conflict driven learning (introduced by GRASP), b) fast BCP (introduced by SATO and improved by Chaff) and c) conflict driven decision making (introduced by Chaff and further developed by BerkMin, Minisat, Siege and others).

There have been significant effort to combine local search algorithms and SAT-solvers based on the DPLL procedure. In [15], in every node of the search tree, a local search procedure is invoked to identify the next variable to branch on. (An important observation made in [15] is that local search can be used for identifying unsatisfiable cores.) This approach is further improved in [9] by taking into account variable dependencies. In [19], random backtracking is used to improve the scalability of the DPLL procedure. In UnitWalk [10], BCP is used to correct values of a complete assignment. The values of this complete assignment are re-assigned in a random order, every variable assignment being followed by BCP. A complete local search algorithm augmented by clause generation is introduced in [4]. Clause generation is used in [4] for escaping local minima. In [6], we described the resolution-based SAT-solver called FI that operates on complete assignments. The decision making procedure of FI can be viewed as a variation of CDDM. Namely, the choice of branching variables is reduced to variables of clauses falsified by the current complete assignment.

Although only FI and DMRP-SAT have all four features mentioned above, some SAT-algorithms can be augmented with missing features (for example, one can add clause learning to UnitWalk.) However, only *DMRP-SAT* combines a DPLL-like procedure and the "genuine" local search strategy of minimizing the set of clauses falsified by a complete assignment. Experiments show that such a local search strategy can be very useful even for highly structured formulas (both satisfiable and unsatisfiable).

There is similarity between the notion of a recursively satisfied clause and that of an autarky [16, 5, 13]. When looking for a partial assignment $\boldsymbol{y}$ such that *modify*($\boldsymbol{p}$,$\boldsymbol{y}$) recursively satisfies a clause C of F, one tries to satisfy clauses of F "touched" by $\boldsymbol{y}$ (like it is done when searching for an autarky). The main difference is that a clause C' of F is considered as touched by an assignment to variable x_i only if x_i is assigned differently in $\boldsymbol{y}$ and the reference point $\boldsymbol{p}$.

5 Experimental results

In this section, we give results of some experiments with an implementation of DMRP-SAT. The experiments were run on Intel's Xeon CPU (3.06GHz) under Linux. The main objective of experiments was to show that although currently DMRP is more expensive than conflict driven decision making, it is competitive with the latter due to high quality of decision making. To keep our implementation easily modifiable we made it very simple. In particular, it lacked many techniques commonly employed to speed up a SAT-solver (see subsection 5.1).

We tried DMRP-SAT on a large set of structured CNF formulas. Here we give data on Bounded Model Checking (BMC) [1, 18, 12] and equivalence checking formulas. This data is representative of the typical behavior of DMRP-SAT. For satisfiable formulas, DMRP-SAT seems to be, in general, more robust than SAT-solvers based on conflict driven decision making. This can be attributed to that, like local search algorithms, DMRP-SAT looks for a satisfying assignment "incrementally".

It is important to note that the current version of DMRP-SAT is meant just to prove that decision-making with a reference point is viable. An optimal design of DMRP-SAT (and many details such as generation of the initial reference point, the best schedule for changing reference points and so on) will be the subject of future research.

5.1 Brief description of implementation

DMRP-SAT is written in C++ and compiled by gcc (version 3.2.2). We used the STL library for data structures like dynamic arrays. As mentioned above, our implementation of DMRP-SAT is very simple. It does not have advanced features like fast BCP, efficient formula representation, special treatment of binary clauses and so on. For example, to check if a clause is unit in BCP, DMRP-SAT just counts the number of unassigned literals (as it was done before SATO and Chaff). The only kind of optimization we used in DMRP-SAT was lazy computation of $D(C,\boldsymbol{p},\boldsymbol{y})$. Namely, during BCP, DMRP-SAT accumulated all the new assignments of $\boldsymbol{y}$ and only when BCP was over it updated $D(C,\boldsymbol{p},\boldsymbol{y})$ *if no conflict occurred*. The reason is that in case of a conflict, recomputing $D(C,\boldsymbol{p},\boldsymbol{y})$ is a waste of time because DMRP-SAT immediately backtracks eliminating all the assignments made at the conflict decision level.

For each literal $lit(x_i)$, DMRP-SAT maintains an array with indexes of clauses of the current formula containing $lit(x_i)$. So when x_i is, say, set to 0, DMRP-SAT examines the clauses of the corresponding array to see if new unit clauses appeared. To avoid examining satisfied clauses, when $lit(x_i)$ is set to 1, all the clauses with $lit(x_i)$ unsatisfied so far are marked as satisfied. The clauses satisfied at a particular decision level are recorded together so that they can be efficiently unmarked when backtracking.

To facilitate decision making and computation of the set $D(C,\boldsymbol{p},\boldsymbol{y})$, DMRP-SAT maintains an array that marks clauses that are currently in $D(C,\boldsymbol{p},\boldsymbol{y})$. For every $lit(x_i)$ it also maintains a counter containing the number of clauses of

$D(C,\boldsymbol{p},\boldsymbol{y})$ that have $lit(x_i)$. Besides, it maintains the set of variables of clauses that are currently in $D(C,\boldsymbol{p},\boldsymbol{y})$. If this set is empty, then $D(C,\boldsymbol{p},\boldsymbol{y}) = \emptyset$ and C is recursively satisfied by $modify(\boldsymbol{p},\boldsymbol{y})$. Finally, DMRP-SAT records the indexes of clauses that are added to $D(C,\boldsymbol{p},\boldsymbol{y})$ at a particular decision level. When undoing assignments of this level, DMRP-SAT removes the recorded clauses from $D(C,\boldsymbol{p},\boldsymbol{y})$.

In all the experiments, the values of *thresh* and *THRESH* were 150 and 3000 respectively (see Figure 1 and Figure 2). That is every 150 conflicts DMRP-SAT made a restart without changing the reference point and every 3000 conflicts such a restart was accompanied by changing the reference point.

5.2 BMC and equivalence checking formulas

In this subsection, we compare our implementation of DMRP-SAT with two SAT-solvers. The first SAT-solver is a version of BerkMin [7] that is very close to Forklift, the winner of the SAT-2003 contest in the industrial category (but in contrast to Forklift, it only learns conflict clauses). This version is much faster than the publicly available one (BerkMin561) on large CNF formulas. The second SAT-solver is Minisat [3], version 1.13 (a similar version of Minisat was the runner-up of the SAT-2005 contest in the industrial category). Table 1 summarizes results of BerkMin, Minisat and DMRP-SAT on a set of large BMC formulas (up to a few millions of variables). These formulas describe various properties of more than a dozen of customer designs. This set consists of 79 formulas (28 satisfiable and 51 unsatisfiable). For all three SAT-solvers, Table 1 gives the total number of conflicts (in thousands), total and median runtime for satisfiable, unsatisfiable and both types of formulas. A sample of formulas from this set are shown in Table 2 (satisfiable formulas are marked with '*').

These two tables show that, for satisfiable formulas, DMRP-SAT makes significantly fewer backtracks (conflicts). Even though BerkMin and Minisat have much faster code and DMRP is more expensive, DMRP-SAT converts the advantage in the number of conflicts into smaller run-times. For unsatisfiable BMC formulas, DMRP-SAT also has fewer conflicts, but this difference is not large enough to convert it into better performance. (However, this should change with a faster implementation.)

Table 3 gives direct evidence that DMRP-SAT indeed benefits from its decision making strategy. For a sample of satisfiable BMC formulas (from the set of 28 formulas mentioned above), this table describes the process of finding a satisfying assignment in more detail. DMRP-SAT can find a satisfying assignment in two ways (see Figures 1,2). First, it can extend the current partial assignment $\boldsymbol{y}$ so that all clauses of the CNF formula become satisfied. Second, when DMRP-SAT is successful in recursively satisfying a clause C, it may find a reference point $\boldsymbol{p'} = modify(\boldsymbol{p},\boldsymbol{y})$ such that $M(\boldsymbol{p'}) = \emptyset$. (When this happens, current partial assignment $\boldsymbol{y}$ may satisfy only a fraction of clauses of F.) Interestingly, for each of 28 satisfiable BMC formulas we used, a satisfying assignment was found after recursively satisfying a clause.

Table 1. BMC formulas

category (#formulas)	BerkMin		Minisat		DMRP-SAT	
	#cnfl. $\times 10^3$	total (median) time, sec.	#cnfl. $\times 10^3$	total (median) time, sec.	#cnfl. $\times 10^3$	total (median) time, sec.
sat (28)	2,546	44,814 (246)	3,457	58,319 (619)	**333**	**9,565 (57)**
unsat (51)	2,156	28,594 **(64)**	1,355	**14,507** (80)	**791**	15,160 (151)
total(79)	4,702	73,408 (96)	4,812	72,826 (178)	**1,124**	**24,725 (69)**

The number of backtracks made before finding a satisfying assignment is reported in the second column of Table 3. The third column shows the number of clauses $|M(\boldsymbol{p})|$ falsified by the original reference point $\boldsymbol{p}$. The number of cases when a clause of $M(\boldsymbol{p})$ was recursively satisfied is given in the fourth column. The size of the longest chain of events when a clause was recursively satisfied with fewer than *THRESH*=3000 backtracks is shown in the fifth column. (Recall that when the number of backtracks exceeds 3000, DMRP-SAT makes a restart and picks a new reference point $\boldsymbol{p'}$. Usually $|M(\boldsymbol{p})| < |M(\boldsymbol{p'})|$.) The last column gives the size of $\boldsymbol{y}$ (in percent of $|Vars(F)|$) when a satisfying assignment $\boldsymbol{p'} = modify(\boldsymbol{p},\boldsymbol{y})$ was found.

Table 3 shows that DMRP-SAT indeed successfully used the "local search strategy" of minimizing the set of falsified clauses to find satisfying assignments. For example, for the formula *byteen*, the original reference point falsified 543 clauses. Then after 255 cases of recursively satisfied clauses a satisfying assignment was found. At this point, only 3.5% of the variables were assigned in the partial assignment $\boldsymbol{y}$. So, in the case of formula *byteen*, DMRP-SAT kept monotonically reducing the size of $M(\boldsymbol{p})$ until a satisfying assignment was found. For some formulas (like *data*), the value of *THRESH* was exceeded and a new reference point was generated (possibly more than once). In such cases the size of the longest chain is smaller than the number of cases when a clause was recursively satisfied. It is worth mentioning that DMRP-SAT had a lot cases of recursively satisfying clauses of $M(\boldsymbol{p})$ for unsatisfiable formulas too.

Finally, Table 4 summarizes results of experiments with satisfiable equivalence checking CNF formulas. Each formula F of Table 4 describes equivalence checking of a combinational circuit N_1 with a circuit N_2 obtained from N_1 by optimization. If N_1 and N_2 are functionally equivalent (inequivalent), then F is unsatisfiable (respectively satisfiable). We manually introduced detectable bugs to the circuit N_2. So all equivalence checking formulas of Table 4 were satisfiable. The first column of Table 4 identifies circuit N_1 (*des* stands for *design*) and gives the number of bugs introduced in circuit N_2 (each bug corresponds to a separate satisfiable formula). Second and third columns give the size of the formula F describing equivalence checking of N_1 and N_2 without any bugs. (Introducing a bug does not affect the formula size much.)

Table 2. Sample of BMC formulas (satisfiable* and unsatisfiable)

name	#vars $\times 10^6$	#clauses $\times 10^6$	BerkMin		Minisat		DMRP-SAT	
			#cnfl. $\times 10^3$	time (sec.)	#cnfl. $\times 10^3$	time (sec.)	#cnfl. $\times 10^3$	time (sec.)
sched*	1.0	2.7	24	386	23	1,038	**0.07**	**2.6**
byteen*	0.2	0.6	21	**138**	60	1,074	**8.8**	245
stimulus*	0.1	0.4	7.9	**39**	49	370	**7.5**	82
ipt*	1.2	3.5	61	2,896	108	3,029	**4.8**	**205**
iqm*	2.3	7.0	308	11,704	732	16,568	**0.5**	**70**
prop3*	1.4	4.3	822	5,230	495	9,084	**77**	**2,479**
gmtx*	2.7	7.9	12	281	47	2,462	**0.05**	**7.5**
sdl*	0.4	1.2	183	551	149	**472**	**75**	1,659
write*	0.6	1.8	8.4	168	48	552	**1.2**	**51**
prop9*	1.0	3.0	74	898	40	429	**2.9**	**58**
unsatisfiable formulas								
always	0.2	0.8	19	45	21	213	**5.0**	**38**
page	0.2	0.8	19	**35**	19	151	**14**	425
mcbdm	0.3	0.8	17	144	6.2	84	**1.5**	**31**
lddata	0.2	0.5	20	**31**	55	666	**18**	255
cmcnt	1.2	3.6	8.5	491	**2.5**	**68**	3.0	134
iwrk	1.3	4.1	202	3,934	31	447	**6.5**	**108**
cho	0.1	0.3	**14**	**23**	15	42	31	1,308
CCC	0.3	1.1	38	199	**22**	**165**	23	1,941

Results of Table 4 show again that DMRP-SAT generated fewer conflicts than BerkMin and Minisat and this advantage was converted into better summary performance. Although DMRP-SAT did not have smaller number of backtracks for all designs, it showed more robust behavior. In particular, it relatively easily solved the equivalence checking formulas generated off the design des_7 that contained a multiplier. We also applied DMRP-SAT to unsatisfiable equivalence checking formulas (no bugs in N_2). DMRP-SAT again had very good performance in terms of the number of backtracks and run-times. For the lack of space we omit these results.

6 Conclusions

We introduce a new decision making strategy DMRP (decision making with a reference point) for resolution-based SAT-solvers. DMRP allows a DPLL-like procedure to pursue a local search strategy. Experiments show that our SAT-solver DMRP-SAT implementing DMRP works well for both satisfiable and unsatisfiable structured formulas. In the current implementation, DMRP is more expensive than conflict driven decision making introduced by Chaff. In our future research we will work on reducing the cost of DMRP. At the same time, even

Table 3. Statistics on recursively satisfied clauses

name	#confl.	size of initial $M(\boldsymbol{p})$	#cases of rec. sat. a clause	#longest chain	$\lvert\boldsymbol{y}\rvert/\lvert Vars(F)\rvert$ when $M(\boldsymbol{p})=\emptyset$ %
sched	67	1	1	1	18
byteen	8,824	543	255	255	3.5
stimulus	7,518	276	29	29	1.8
data	15,521	1,034	212	114	77
ifreeq	3,426	615	438	438	1.9
ipt	4,750	775	601	601	0.8
prop3	77,127	44	29	6	76
muls	556	104	69	69	1.4
T1	64	2	1	1	67
TX	77,934	8	7	3	96
HP-4850	17,932	62	8	7	1.0
HP-974	2,092	1	1	1	44
write	1,175	149	87	87	0.9
prop9	2,892	1	1	1	31
SUN-442	17	1	1	1	95
SUN-443	2,010	3,999	2,000	2,000	1.6

a straightforward and unoptimized implementation of DMRP-SAT shows very good performance due to high quality of decision-making.

References

1. A.Biere, A.Cimatti, E.Clarke, O.Strichman, Y.Zhu. *Bounded Model Checking*, (a book chapter), Advances in computers vol. 58, ed. by M. Zelkovitz, Elsevier, 2003.
2. M.Davis, G.Longemann, D.Loveland. *A Machine program for theorem proving.* Communications of the ACM, 1962, vol. 5, pp.394-397.
3. Een N., Sorensson N. *An extensible SAT-solver.* Proceedings of SAT-2003 in LNCS 2919, pp.503-518.
4. H.Fang, W.Ruml. *Complete Local Search for Propositional Satisfiability.* Proc. of 19th National Conference on Artificial Intelligence, 2004, pp.161-166.
5. A.V.Gelder *Autarky pruning in propositional model elimination reduces failure redundancy.* J. of Autom. Reasoning,1999, vol 23(2),pp.137-193
6. E.Goldberg. *Determinization of resolution by an algorithm operating on complete assignments.* SAT-2006, LNCS 4121, pp.90-95.
7. E.Goldberg, Y.Novikov. *BerkMin: a Fast and Robust SAT-Solver.* DATE-2002, Paris, pp.142-149.
8. C. P. Gomes, B. Selman, H. Kautz. *Boosting Combinatorial Search Through Randomization.* Proc. AAAI-98.
9. D.Habet, C.M.Li, L.Devendeville, M.Vasquez. *A hybrid approach for SAT.* Int. Conf. on Principles and Practice of Constraint Programming, 2002, pp. 172-184.
10. E. A. Hirsch, A. Kojevnikov. *UnitWalk: A new SAT solver that uses local search guided by unit clause elimination.* Annals of Math. and Artif. Intell. 43(1-4), pp.91-111, 2005

Table 4. Equivalence checking (satisfiable formulas)

name (#bugs)	#vars $\times 10^3$	#clauses $\times 10^3$	BerkMin		Minisat		DMRP-SAT	
			#cnfl. $\times 10^3$	total time (sec.)	#cnfl. $\times 10^3$	total time (sec.)	#cnfl. $\times 10^3$	total time (sec.)
des_1 (7)	4.7	53	271	120	110	143	**11**	**56**
des_2(8)	2.4	24	229	40	**95**	**10**	162	197
des_3(7)	2.7	29	169	39	**46**	**3**	83	114
des_4(5)	1.0	9.8	54	5	14	**0.4**	**13**	6
des_5 (5)	1.9	20	80	8	55	**4**	**37**	25
des_6(7)	9.5	106	2,484	2,624	1,327	1,783	**88**	**389**
des_7 (4)	1.6	16	75	11	39	**2**	**15**	8
des_8(4)	3.5	39	69	**17**	111	53	**52**	116
Total			3,431	2,864	1,797	1,998	**461**	**911**

11. H.Hoos, T.Stutzle. *Stochastic Local Search: Foundations and Applications.* Morgan Kaufmann, San Francisco (CA), USA, 2004.
12. J.Katz, Z.Hanna, N.Dershowitz. *Space-efficient Bounded Model Checking.* DATE-2005, pp. 686-687.
13. O.Kullmann. *Investigations on autark assignments*, Discrete Applied Mathematics 2000, vol 107, pp.99-137.
14. C.M.Li. *A constrained-based approach to narrow search trees for satisfiability.* Information processing letters,1999, 71, pp.75-80.
15. B.Mazure, L.Sais, and R.Gregoire. *Boosting complete techniques thanks to local search methods.* Annals of Math. and Artif. Intell, vol. 22 (1998), pp. 319-331.
16. B.Monien, E.Speckenmeyer. *Solving satisfiability in less than 2^n steps.* Discrete Applied Mathematics, 1985, vol. 10, pp.287-295.
17. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. *Chaff: Engineering an Efficient SAT Solver.* DAC-2001.
18. M.Prasad, A.Biere, A.Gupta, *A survey of recent advances in SAT-based formal verification*, STTT vol. 7(2), pp.16-173, 2005.
19. S.Prestwich. *Local search and backtracking vs. non-systematic backtracking.* AAAI Fall Symposium on Using Uncertainty Within Computation. Nov. 2-4, 2001, North Falmouth, Cape Cod, MA,pp.109-115.
20. B. Selman H. Levesque, D. Mitchell. 1992. *A New Method for Solving Hard Satisfiability Problems.* AAAI-92, pp. 440-446.
21. B.Selman, H.A.Kautz. and B.Cohen. *Noise strategies for improving local search.* AAAI-94, Seattle, pp. 337-343, 1994.
22. J.P.M.Silva, K.A.Sakallah. *GRASP: A Search Algorithm for Propositional Satisfiability.* IEEE Transactions of Computers, 1999, vol. 48, pp. 506-521.
23. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. *Efficient Conflict Driven Learning in a Boolean Satisfiability Solver.* ICCAD-2001.
24. H.Zhang. *SATO: An efficient propositional prover.* International Conference on Automated Deduction, July 1997, pp.272-275.