Checking Satisfiability by Dependency Sequents

Eugene Goldberg, Panagiotis Manolios

Northeastern University, USA {eigold,pete}@ccs.neu.edu

Abstract. We introduce a new algorithm for checking satisfiability based on a calculus of Dependency sequents (D-sequents). Given a CNF formula $F(X)$, a D-sequent is a record stating that under a partial assignment a set of variables of X is redundant in formula $\exists X[F]$. The D-sequent calculus is based on operation *join* that forms a new D-sequent from two existing ones. The new algorithm solves the *quantified* version of SAT. That is, given a satisfiable formula F, it, in general, does not produce an assignment satisfying F. The new algorithm is called DS-QSAT where DS stands for Dependency Sequent and Q for Quantified. Importantly, a DPLL-like procedure is only a special case of DS-QSAT where a very restricted kind of D-sequents is used. We argue that this restriction a) adversely affects scalability of SAT-solvers b) is caused by looking for an *explicit* satisfying assignment rather than just proving satisfiability. We give experimental results substantiating these claims.

1 Introduction

Algorithms for solving the Boolean satisfiability problem are an important part of modern design flows. Despite great progress in the performance of such algorithms achieved recently, the scalability of SAT-solvers still remains a big issue. In this paper, we address this issue by introducing a new method of satisfiability checking that can be viewed as a descendant of the DP procedure [3].

We consider Boolean formulas represented in Conjunctive Normal Form (CNF). Given a CNF formula $F(X)$, one can formulate two different kinds of satisfiability checking problems. We will refer to the problems of the first kind as **QSAT** where Q stands for quantified. Solving QSAT means just checking if $\exists X[F]$ is true. In particular, if F is satisfiable, a QSAT-solver does not have to produce an assignment satisfying F. The problems of the second kind that we will refer to as just **SAT** are a special case of those of the first kind. If F is satisfiable, a SAT-solver has to produce an assignment satisfying F.

Intuitively, QSAT should be easier than SAT because a QSAT-solver needs to return only one bit of information. This intuition is substantiated by the fact that checking if an integer number N is prime (i.e. answering the question if non-trivial factors of N *exist*) is polynomial while finding factors of N *explicitly* is believed to be hard. However, the situation among practical algorithms defies this intuition. Currently, the field is dominated by procedures based on DPLL algorithm [2] that is by SAT-solvers. On the other hand, a classical QSAT-solver, the DP procedure [3], does not have any competitive descendants (although some

elements of the DP procedure are used in formula preprocessing performed by SAT-solvers [5]).

In this paper, we introduce a QSAT-solver called **DS-QSAT** where DS stands for Dependency Sequent. On the one hand, DS-QSAT can be viewed as a descendant of the DP procedure. On the other hand, DPLL-like procedures with clause learning is a special case of DS-QSAT. Like DP procedure, DS-QSAT is based on the idea of elimination of redundant variables. A variable $v \in X$ is redundant in $\exists X[F]$ if the latter is equivalent to $\exists X[F \setminus F^v]$ where F^v is the set of all clauses of F with v. Note that removal of clauses of F^v produces a formula that is equisatisfiable rather than functionally equivalent to F.

If F is satisfiable, all variables of X are redundant in $\exists X[F]$ because an empty set of clauses is satisfiable. If F is unsatisfiable, one can make the variables of F redundant by deriving an empty clause and adding it to F. An empty clause is unsatisfiable, hence all other clauses of F can be dropped. So, from the viewpoint of DS-QSAT, the only difference between satisfiable and unsatisfiable formulas is as follows. If F is satisfiable, its variables are *already* redundant and one just needs to prove this redundancy. If F is unsatisfiable, one has to *make* variables redundant by derivation and adding to F an empty clause.

The DP procedure makes a variable v of X redundant *in one step*, by adding to F all clauses that can be produced by resolution on v. This is extremely inefficient due to generation of prohibitively large sets of clauses even for very small formulas. DS-QSAT addresses this problem by using branching. The idea is to prove redundancy of variables in subspaces and then "merge" the obtained results. DS-QSAT records the fact that a set of variables Z is redundant in $\exists X[F]$ in subspace specified by partial assignment $\boldsymbol{q}$ as $(\exists X[F], \boldsymbol{r}) \rightarrow Z$. Here $\boldsymbol{r}$ is a subset of the assignments of $\boldsymbol{q}$ relevant to redundancy of Z. The record $(\exists X[F], \boldsymbol{r}) \rightarrow Z$ is called a dependency sequent (or **D-sequent** for short). To simplify notation, if F and X are obvious from the context, we record the D-sequent above as just $\boldsymbol{r} \rightarrow Z$.

A remarkable fact is that a resolution-like operation called *join* can be used to produce a new D-sequent from two D-sequents derived earlier [8,7]. Suppose, for example, that D-sequents $(x_1 = 0, x_2 = 0) \rightarrow \{x_9\}$ and $(x_2 = 1, x_5 = 1) \rightarrow \{x_9\}$ specify redundancy of variable x_9 in different branches of variable x_2. Then D-sequent $(x_1 = 0, x_5 = 1) \rightarrow \{x_9\}$ holds where the left part assignment of this D-sequent is obtained by taking the union of the left part assignments of the two D-sequents above but those to variable x_2. The new D-sequent is said to be obtained by joining the two D-sequents above at variable x_2. The calculus based on the join operation is complete. That is, eventually DS-QSAT derives D-sequent $\emptyset \rightarrow X$ stating *unconditional* redundancy of the variables of X in $\exists X[F]$. If by the time the D-sequent above is derived, F contains an empty clause, F is unsatisfiable. Otherwise, F is satisfiable. Importantly, if F is satisfiable, derivation of D-sequent $\emptyset \rightarrow X$ does not require finding an assignment satisfying F.

DPLL-based SAT-solvers with clause learning can be viewed as a special case of DS-QSAT where only a particular kind of D-sequents is used. This limi-

tation on D-sequents is caused by the necessity to generate a satisfying assignment as a proof of satisfiability. Importantly, this necessity deprives DPLL-based SAT-solvers of using transformations preserving equisatisfiability rather than functional equivalence. In turn, this adversely affects the performance of SAT-solvers. We illustrate this point by comparing the performance of DPLL-like SAT-solvers and a version of DS-QSAT on compositional formulas. This version of DS-QSAT use the strategy of lazy backtracking as opposed to that of eager backtracking employed by DPLL-based procedures. A compositional CNF formula has the form $F_1(X_1) \wedge \ldots \wedge F_k(X_k)$ where $X_i \cap X_j = \emptyset$,$i \neq j$. Subformulas F_i, F_j are identical modulo variable renaming/negation. We prove theoretically that performance of DS-QSAT is *linear* in k. On the other hand, one can argue that the average performance of DPLL-based SAT-solvers with conflict learning should be *quadratic* in k. In Section 8, we describe experiments confirming our theoretical results.

The contribution of this paper is fourfold. First, we use the machinery of D-sequents to explain some problems of DPLL-based SAT-solvers. Second, we describe a new QSAT-solver based on D-sequents called DS-QSAT. Third, we give a theoretical analysis of the behavior of DS-QSAT on compositional formulas. Fourth, we show the promise of DS-QSATby comparing its performance with that of well-known SAT-solvers on compositional and non-compositional formulas.

This paper is structured as follows. In Section 2 we discuss the complexity of QSAT and SAT. Section 3 gives a brief introduction into DS-QSAT. We recall D-sequent calculus in Section 4. A detailed description of DS-QSAT is given in Section 5. Section 6 gives some theoretical results on performance of DS-QSAT. Section 7 describes a modification of DS-QSAT that allows additional pruning of the search tree. Experimental results are given in Section 8. We describe some background of this research in Section 9 and give conclusions in Section 10.

2 Is QSAT Simpler Than SAT?

In this section, we make the following point. Both QSAT-solvers and SAT-solvers have exponential complexity on the set of *all* CNF formulas, unless P = NP. However, this is not true for subsets of CNF formulas. It is possible that a set K of formulas describing, say, properties of a parameterized set of designs can be solved in polynomial time by some QSAT-solver while any SAT-solver has exponential complexity on K.

To illustrate the point above, let us consider procedure *gen_sat_assgn* shown in Figure 1. It finds an assignment satisfying a CNF formula F (if any) by solving a sequence of QSAT problems. First, *gen_sat_assgn* calls a QSAT-solver *solve_qsat* to check if F is satisfiable (line 2). If it is, *gen_sat_assgn* picks a variable v of F (line 5) and calls *solve_qsat* to find assignment $v = val$ under which formula F is satisfiable (lines 6-8). Since F is satisfiable, $F_{v=0}$ and/or $F_{v=1}$ has to be satisfiable. Then *gen_sat_assgn* fixes variable v at the chosen value *val* and adds (v=*val*) to assignment $\boldsymbol{s}$ (lines 9-10) that was originally empty. The

gen_sat_assgn procedure keeps assigning variables of F in the same manner in a loop (lines 5-11) until every variable of F is assigned. At this point, $\boldsymbol{s}$ is a satisfying assignment of F.

```
gen_sat_assgn(F){
1  ans = solve_qsat(F);
2  if (ans=unsat) return(unsat);
3  s := ∅; X:=Vars(F);
4  while (X ≠ ∅) {
5     v := pick_var(X);
6     if (solve_qsat(F_{v=0}) = sat)
7        val = 0;
8     else val = 1;
9     F := F_{v=val};
10    s = s ∪ {(v = val)};
11    X := X \ {v};}
12 return(s);
```

Fig. 1. SAT-solving by QSAT

The number of QSAT checks performed by *solve_qsat* in *gen_sat_assgn* is at most $n+1$. So if there is a QSAT-solver solving all satisfiable CNF formulas in polynomial time, *gen_sat_assgn* can use this QSAT-solver in its inner loop to find a satisfying assignment for any satisfiable formula in polynomial time. However, this is not true when considering *a subset* K of all possible CNF formulas. Suppose there is a QSAT-solver solving the formulas of K in polynomial time. Let F be a formula of K. Let $F_{\boldsymbol{q}}$ denote F under partial assignment $\boldsymbol{q}$. The fact that $F \in K$ does not imply $F_{\boldsymbol{q}} \in K$. So the behavior of *gen_sat_assgn* using this QSAT-solver in the inner loop may actually be even *exponential* if this QSAT-solver does not perform well on formulas $F_{\boldsymbol{q}}$.

For example, one can form a subset K of all possible CNF formulas such that a) a formula $F \in K$ describes a check that a number N is composite and b) an assignment satisfying F (if any) specifies two numbers A,B such that $A \neq 1$,$B \neq 1$ and $A \times B = N$. The satisfiability of formulas in K can be checked by a QSAT-solver in polynomial time [14]. At the same time, finding satisfying assignments of formulas from K i.e. factorization of composite numbers is believed to be hard. For instance, *gen_sat_assgn* cannot use the QSAT-solver above to find satisfying assignments for formulas of K in polynomial time. The reason is that formula $F_{\boldsymbol{q}}$ does not specify a check if a number is composite. That is $F \in K$ does not imply that $F_{\boldsymbol{q}} \in K$.

Note that a SAT-solver is also limited in the ways of proving *unsatisfiability*. For a SAT-solver, such a proof is just a failed attempt to build a satisfying assignment *explicitly*. For example, instead of using the polynomial algorithm of [14], a SAT-solver would prove that a number N is prime by failing to find two non-trivial factors of N.

3 Brief comparison of DPLL-based SAT-solvers and DS-QSAT in Terms of D-sequents

In this section, we use the notion of D-sequents to discuss some limitations of DPLL-based SAT-solvers. We also explain how DS-QSAT (described in Section 5 in detail) overcomes those limitations.

Example 1. Let SAT_ALG be a DPLL-based SAT-solver with clause learning. We assume that the reader is familiar with the basics of such SAT-solvers [15,16].

Let F be a CNF formula of 8 clauses where $C_1 = \overline{x}_1 \vee \overline{x}_3$, $C_2 = \overline{x}_2 \vee x_3$, $C_3 = x_1 \vee x_2 \vee x_3$, $C_4 = x_2 \vee \overline{x}_3$, $C_5 = \overline{x}_1 \vee x_4 \vee x_5$, $C_6 = x_4 \vee \overline{x}_5$, $C_7 = \overline{x}_4 \vee x_5$, $C_8 = \overline{x}_1 \vee \overline{x}_4 \vee \overline{x}_5$. The set X of variables of F is equal to $\{x_1, x_2, x_3, x_4, x_5\}$.

Let SAT_ALG first make assignment $x_1 = 0$. This satisfies clauses C_1,C_5,C_8 and removes literal x_1 from C_3. Let SAT_ALG then make assignment $x_2 = 0$. Removing literal x_2 from C_3 and C_4 turn them into unit clauses x_3 and $\overline{x}_3$ respectively. This means that SAT_ALG ran into a conflict. At this point, SAT_ALG generates conflict clause $C_9 = x_1 \vee x_2$ that is obtained by resolving clauses C_3 and C_4 on x_3 and adds C_9 to F. After that, SAT_ALG erases assignment $x_2 = 0$ and the assignment made by SAT_ALG to x_3 and runs BCP that assigns $x_2 = 1$ to satisfy C_9 that is currently unit. In terms of D-sequents, one can view generation of conflict clause C_9 and adding it to F as derivation of D-sequent S equal to $(x_1 = 0, x_2 = 0) \rightarrow \{x_3, x_4, x_5\}$. D-sequent S says that making assignments falsifying clause C_9 renders all unassigned variables redundant. Note that S is inactive in the subspace $(x_1 = 0, x_2 = 1)$ that SAT_ALG enters after assigning 1 to x_2. (We will say that D-sequent $\boldsymbol{r} \rightarrow Z$ is **active** in the subspace specified by partial assignment $\boldsymbol{q}$ if the assignments of $\boldsymbol{r}$ are a subset of those of $\boldsymbol{q}$.) So the variables x_3, x_4, x_5 proved redundant in subspace $(x_1 = 0, x_2 = 0)$ become non-redundant again.

One may think that reappearance of variables x_3, x_4, x_5 in subspace $(x_1 = 0, x_2 = 1)$ is "inevitable" but this is not so. Variables x_4,x_5 have at least two reasons to be redundant in subspace $(x_1 = 0, x_2 = 0)$. First, C_9 is falsified in this subspace. Second, the only clauses of F containing variables x_4,x_5 are C_5,C_6,C_7,C_8. But C_5 and C_8 are satisfied by $x_1 = 0$ and C_6, C_7 can be satisfied by an assignment to x_4,x_5. So C_5,C_6,C_7,C_8 can be removed from F in subspace $x_1 = 0$ without affecting the satisfiability of F. Hence D-sequents S_1 and S_2 equal to $(x_1 = 0) \rightarrow \{x_4\}$ and $(x_1 = 0) \rightarrow \{x_5\}$ are true. (In Example 3, we will show how S_1 and S_2 are derived by DS-QSAT.) Suppose that one replaces the D-sequent S above with D-sequents S', S_1, S_2 where S' is equal to $(x_1 = 0, x_2 = 0) \rightarrow \{x_3\}$. Note that only D-sequent S' is inactive in subspace $(x_1 = 0, x_2 = 1)$. So *only variable* x_3 reappears after x_2 changes its value from 0 to 1 $\square$

The example above illustrates the main difference between SAT_ALG and DS-QSAT in terms of D-sequents. At every moment, SAT_ALG has *at most one* active D-sequent. This D-sequent is of the form $\boldsymbol{r} \rightarrow Z$ where r is an assignment falsifying a clause of F and Z is the set of *all* variables that are currently unassigned. DS-QSAT may have a *set* of active D-sequents $\boldsymbol{r_1} \rightarrow Z_1, \ldots, \boldsymbol{r_k} \rightarrow Z_k$ where $Z_1 \cup \ldots \cup Z_k = Z$, $Z_i \cap Z_j = \emptyset$,$i \neq j$. When SAT_ALG changes the value of variable v of $Vars(\boldsymbol{r})$, all the variables of Z reappear as non-redundant. When DS-QSAT changes the value of v, variables of Z_i reappear *only* if $v \in Vars(\boldsymbol{r_i})$. So only a subset of variables of Z reappear.

To derive D-sequents $\boldsymbol{r_i} \rightarrow Z_i$ above, DS-QSAT goes on branching *in the presence of a conflict*. Informally, the goal of such branching is to find alternative ways of proving redundancy of variables from Z. So DS-QSAT uses extra branching to minimize the number of variables reappearing in the right branch (after the left branch has been explored). This should eventually lead to the opposite result i.e. to reducing the amount of branching. Looking for alternative ways to

prove redundancy can be justified as follows. A practical formula F typically can be represented as $F_1(X_1, Y_1) \wedge \ldots \wedge F_k(X_k, Y_k)$. Here X_i are internal variables of F_i and Y_i are "communication" variables that F_i may share with some other subformulas F_j, $j \neq i$. One can view F_i as describing a "design block" with external variables Y_i. Usually, $|Y_i|$ is much smaller than $|X_i|$. Let a clause of F_i be falsified by the current assignment due to a conflict. Suppose that at the time of the conflict all variables of Y_j of subformula F_j were assigned and their values were specified by assignment $\boldsymbol{y_i}$. Suppose $\boldsymbol{y_i}$ is consistent for F_i i.e. $\boldsymbol{y_i}$ can be extended by assignments to X_i to satisfy F_i. This means that the variables of X_i are redundant in subspace $\boldsymbol{y_i}$ in $\exists V[F]$ where $V = \mathit{Vars}(F)$. Then by branching on variables of X_i one can derive D-sequent $\boldsymbol{y_i} \rightarrow X_i$. If $\boldsymbol{y_i}$ is inconsistent for F_i, then by branching on variables of X_i one can derive a clause C falsified by $\boldsymbol{y_i}$. Adding C to F makes the variables of X_i redundant in $\exists V[F]$ in subspace $\boldsymbol{y_i}$. So the existence of many ways to prove variable redundancy is essentially implied by the fact that formula F has structure.

The possibility to control the size of right branches gives an algorithm a lot of power. Suppose, for example, that an algorithm guarantees that the number of variables reappearing in the right branch is bounded by a constant d. We assume that this applies to the right branch going out of any node of the search tree, including the root node. Then the size of the search tree built by such an algorithm is $O(|X| \cdot 2^d)$. Here $|X|$ is the maximum depth of a search tree built by branching on variables of X and 2^d is the number of nodes in a full binary sub-tree over d variables. So the factor 2^d limits the size of the right branch. The complexity of an algorithm building such a search tree is *linear* in F. In Section 6, we show that bounding the size of right branches by a constant is exactly the reason why the complexity of DS-QSAT on compositional formulas is linear in the number of subformulas.

The limitation of D-sequents available to SAT_ALG is consistent with the necessity to produce a satisfying assignment. Although such limitation cripples the ability of an algorithm to compute the parts of the formula that are redundant in the current subspace, it does not matter much for SAT_ALG. The latter simply cannot use this redundancy because it is formulated with respect to formula $\exists X[F]$ rather than F. Hence, discarding the clauses containing redundant variables preserves equisatisfiability rather than functional equivalence. So, an algorithm using such transformations cannot guarantee that a satisfying assignment it found is correct.

4 D-sequent Calculus

In this section, we recall the D-sequent calculus introduced [8,7]. In Subsections 4.1 and 4.2 we give basic definitions and describe simple cases of variable redundancy. The notion of D-sequents is introduced in Subsection 4.3. Finally, the operation of joining D-sequents is presented in Subsection 4.4.

4.1 Basic definitions

Definition 1. *A **literal** of a Boolean variable v is v itself and its negation. A **clause** is a disjunction of literals. A formula F represented as a conjunction of clauses is said to be the Conjunctive Normal Form (**CNF**) of F. A CNF formula F is also viewed as **a set of clauses**. Let $\boldsymbol{q}$ be an assignment, F be a CNF formula, and C be a clause. $\boldsymbol{Vars(q)}$ denotes the variables assigned in $\boldsymbol{q}$; $\boldsymbol{Vars(F)}$ denotes the set of variables of F; $\boldsymbol{Vars(C)}$ denotes the set of variables of C.*

Definition 2. *Let $\boldsymbol{q}$ be an assignment. Clause C is **satisfied** by $\boldsymbol{q}$ if a literal of C is set to 1 by $\boldsymbol{q}$. Otherwise, C is **falsified** by $\boldsymbol{q}$. Assignment $\boldsymbol{q}$ **satisfies** F if $\boldsymbol{q}$ satisfies every clause of F.*

Definition 3. *Let F be a CNF formula and $\boldsymbol{q}$ be a partial assignment to variables of F. Denote by $F_{\boldsymbol{q}}$ that is obtained from F by a) removing all clauses of F satisfied by $\boldsymbol{q}$; b) removing the literals set to 0 by $\boldsymbol{q}$ from the clauses that are not satisfied by $\boldsymbol{q}$. Notice, that if $\boldsymbol{q}$=$\emptyset$, then $F_{\boldsymbol{q}} = F$.*

Definition 4. *Let F be a CNF formula and Z be a subset of $Vars(F)$. Denote by $\boldsymbol{F^Z}$ the set of all clauses of F containing at least one variable of Z.*

Definition 5. *The variables of Z are **redundant** in formula $\exists X[F]$ if $\exists X[F] \equiv \exists X[F \setminus F^Z]$. We note that since $F \setminus F^Z$ does not contain any Z variables, we could have written $\exists(X \setminus Z)[F \setminus F^Z]$. To simplify notation, we avoid explicitly using this optimization in the rest of the paper.*

Definition 6. *Let $\boldsymbol{q_1}$ and $\boldsymbol{q_2}$ be assignments. The expression $\boldsymbol{q_1} \leq \boldsymbol{q_2}$ denotes the fact that $Vars(\boldsymbol{q_1}) \subseteq Vars(\boldsymbol{q_2})$ and each variable of $Vars(\boldsymbol{q_1})$ has the same value in $\boldsymbol{q_1}$ and $\boldsymbol{q_2}$.*

4.2 Simple cases of variable redundancy

There at least two cases where proving that a variable of F is redundant in $\exists X[F]$ is easy. The first case concerns monotone variables of F. A variable v of F is called **monotone** if all clauses of F containing v have only positive (or only negative) literal of v. A monotone variable v is redundant in $\exists X[F]$ because removing the clauses with v from F does not change the satisfiability of F. The second case concerns the presence of an empty clause. If F contains such a clause, every variable of F is redundant.

4.3 D-sequents

Definition 7. *Let $F(X)$ be a CNF formula. Let $\boldsymbol{q}$ be an assignment to X and Z be a subset of $X \setminus Vars(\boldsymbol{q})$. A dependency sequent (**D-sequent**) has the form $(\exists X[F], \boldsymbol{q}) \rightarrow Z$. It states that the variables of Z are redundant in $\exists X[F_{\boldsymbol{q}}]$. If formula F for which a D-sequent holds is obvious from the context we will write this D-sequent in a short notation: $\boldsymbol{q} \rightarrow Z$.*

Example 2. Let F be a CNF formula of four clauses: $C_1 = x_1 \vee x_2$, $C_2 = \overline{x}_1 \vee \overline{x}_2$, $C_3 = \overline{x}_1 \vee x_3$, $C_4 = x_2 \vee \overline{x}_3$. Notice that since clause C_1 is satisfied in subspace $(x_2 = 1)$, variable x_1 is monotone in formula $F_{x_2=1}$. So D-sequent $(x_2 = 1) \rightarrow \{x_1\}$ holds. On the other hand, the assignment $\boldsymbol{r} = (x_1 = 1, x_3 = 0)$ falsifies clause C_3. So variable x_2 is redundant in $F_{\boldsymbol{r}}$ and D-sequent $\boldsymbol{r} \rightarrow \{x_2\}$ holds.

4.4 Join Operation for D-sequents

Proposition 1 ([8]). *Let $F(X)$ be a CNF formula. Let D-sequents $\boldsymbol{r'} \rightarrow Z$ and $\boldsymbol{r''} \rightarrow Z$ hold, where $Z \subseteq X$. Let $\boldsymbol{r'}$, $\boldsymbol{r''}$ have different values for exactly one variable $v \in Vars(\boldsymbol{r'}) \cap Vars(\boldsymbol{r''})$. Let $\boldsymbol{r}$ consist of all assignments of $\boldsymbol{r'}$,$\boldsymbol{r''}$ but those to v. Then, D-sequent $\boldsymbol{r} \rightarrow Z$ holds too.*

We will say that the D-sequent $\boldsymbol{r} \rightarrow Z$ of Proposition 1 is obtained by **joining D-sequents** $\boldsymbol{r'} \rightarrow Z$ and $\boldsymbol{r''} \rightarrow Z$ at variable v. The join operation is *complete* [8,7]. That is eventually, D-sequent $\emptyset \rightarrow X$ is derived proving that the variables of the current formula F are redundant. If F contains an empty clause, then F is unsatisfiable. Otherwise, it is unsatisfiable.

An obvious difference between the D-sequent calculus and resolution is that the former can handle both satisfiable and unsatisfiable formulas. This limitation of resolution is due to the fact that it operates on subspaces where formula F is unsatisfiable. One can interpret resolving clauses C', C'' to produce clause C as using the Boolean cubes K',K'' where C' and C'' are unsatisfiable to produce a new Boolean cube K where the resolvent C is unsatisfiable. On the contrary, the join operation can be performed over parts of the search space where F may be satisfiable. When D-sequents $\boldsymbol{r'} \rightarrow Z$ and $\boldsymbol{r''} \rightarrow Z$ are joined, it does not matter whether formulas $F_{\boldsymbol{r'}}$ and $F_{\boldsymbol{r''}}$ are satisfiable. The only thing that matters is that variables Z are redundant in $F_{\boldsymbol{r'}}$ and $F_{\boldsymbol{r''}}$.

4.5 Virtual redundancy

Let $F(X)$ be a CNF formula and $\boldsymbol{r}$ be an assignment to X. Let $Z \subseteq X$ and $Vars(\boldsymbol{r}) \cap Z = \emptyset$. The fact that variables of Z are redundant in F, in general, does not mean that they are redundant in $F_{\boldsymbol{r}}$. Suppose, for example, that F is satisfiable, $F_{\boldsymbol{r}}$ is unsatisfiable, F does not have a clause falsified by $\boldsymbol{r}$ and $Z = Vars(F) \setminus Vars(\boldsymbol{r})$. Then formula $F_{\boldsymbol{r}} \setminus (F_{\boldsymbol{r}})^Z$ has no clauses and so is satisfiable. Hence $\exists X[F_{\boldsymbol{r}}] \neq \exists X[F_{\boldsymbol{r}} \setminus (F_{\boldsymbol{r}})^Z]$ and so the variables of Z are not redundant in $F_{\boldsymbol{r}}$. On the other hand, since F is satisfiable, the variables of Z are redundant in $\exists X[F]$.

We will say that the variables of Z are **virtually redundant** in $F_{\boldsymbol{r}}$ where $Z \cap Vars(\boldsymbol{r}) = \emptyset$ if either a) $\exists X[F_{\boldsymbol{r}}] = \exists X[F_{\boldsymbol{r}} \setminus (F_{\boldsymbol{r}})^Z]$ or b) $\exists X[F_{\boldsymbol{r}}] \neq \exists X[F_{\boldsymbol{r}} \setminus (F_{\boldsymbol{r}})^Z]$ and F is satisfiable. In other words, if variables Z are virtually redundant in $\exists X[F_{\boldsymbol{r}}]$, removing the clauses with a variable of Z from $F_{\boldsymbol{r}}$ may be wrong but only *locally*. From the global point of view this mistake does not matter because it occurs only when F is satisfiable.

We need a new notion of redundancy because the join operation introduced above preserves virtual redundancy [8] rather than redundancy in terms of Definition 5. Suppose, for example, that the variables of Z are redundant in $F_{\boldsymbol{r_1}}$ and $F_{\boldsymbol{r_2}}$ in terms of Definition 5 and so D-sequents $\boldsymbol{r_1} \rightarrow Z$ and $\boldsymbol{r_2} \rightarrow Z$ hold. Let $\boldsymbol{r} \rightarrow Z$ be the D-sequent obtained by joining the D-sequents above. Then one can guarantee only that the variables of Z are virtually redundant in $F_{\boldsymbol{r}}$. For that reason we need to replace the notion of redundancy by Definition 5 with that of virtually redundancy. In the future explanation, we will omit the word "virtually". That is when we say that variables of Z are redundant in $F_{\boldsymbol{r}}$ we actually mean that they are *virtually* redundant in $F_{\boldsymbol{r}}$.

5 Description of DS-QSAT

In this section, we describe DS-QSAT, a QSAT-solver based on the machinery of D-sequents.

5.1 High-level view

Pseudocode of DS-QSAT is given in Figure 2. DS-QSAT accepts a CNF formula F, a partial assignment $\boldsymbol{q}$ to X where $X = \mathit{Vars}(F)$, and a set of active D-sequents Ω stating redundancy of *some* variables from $X \setminus \mathit{Vars}(\boldsymbol{q})$ in subspace $\boldsymbol{q}$. DS-QSAT returns CNF formula F that consists of the clauses of the initial formula plus some resolvent clauses and a set Ω of D-sequents stating redundancy of *every* variable of $X \setminus \mathit{Vars}(\boldsymbol{q})$ in subspace $\boldsymbol{q}$. To check satisfiability of a CNF formula, one needs to call DS-QSAT with $\boldsymbol{q} = \emptyset$, $\Omega = \emptyset$.

DS-QSAT is a branching procedure. If DS-QSAT cannot prove redundancy of some variables in the current subspace, it picks one of such variables v and branches on it. So DS-QSAT builds a binary search tree where a node corresponds to a branching variable. We will refer to the first (respectively second) assignment to v as the **left** (respectively **right**) **branch** of v. Although Boolean Constraint Propagation (BCP) is not explicitly mentioned in Figure 2, it is included into the *pick_variable* procedure as follows. Let $\boldsymbol{q}$ be the current partial assignment. Then a) preference is given to branching on variables of unit clauses of $F_{\boldsymbol{q}}$ (if any); b) if v is a variable of a unit clause of C of $F_{\boldsymbol{q}}$ and v is picked for branching, then the value satisfying C is assigned first.

As soon as a variable v is proved redundant in the current subspace $\boldsymbol{q}$, a D-sequent $\boldsymbol{r} \rightarrow \{v\}$ is recorded where $\boldsymbol{r}$ is a subset of assignments of $\boldsymbol{q}$. All the clauses of F containing variable v are marked as redundant and *ignored* until v becomes non-redundant again. This happens when a variable of $\mathit{Vars}(\boldsymbol{r})$ changes its value making the D-sequent $\boldsymbol{r} \rightarrow \{v\}$ inactive in the current subspace.

As we mentioned in Section 3, if a clause C containing a variable v is falsified after an assignment is made to v, DS-QSAT keeps making assignments to unassigned non-redundant variables. However, this happens only in the *left* branch of v. If C is falsified in the right branch of v, DS-QSAT backtracks. A unit clause C' gets falsified in the left branch only when DS-QSAT tries to satisfy another

unit clause C'' such that C' and C'' have the opposite literals of a variable v. We will refer to the node of the search tree corresponding to v as a *conflict* one. The number of conflict nodes DS-QSAT may have is not limited.

```
// F is a CNF formula
// q is an assignment to Vars(F)
// Ω is a set of active D-sequents

DS-QSAT(F,q,Ω){
1   if (empty_clause(F))
      exit(unsat);
2   if (new_falsif_clause(C, F, q))
3     if (left_branch(q))
4       Ω:=update_Dseqs(Ω, F, C);
5     else {
6       Ω:=finish_Dseqs(Ω, F, C);
7       return(F, Ω); }
8   Ω := monot_vars_Dseqs(Ω, F, q);
9   if (all_vars_assgn_or_redund(Ω,q);
10    if (no_falsif_clauses(F, q))
        exit(sat);
11    else return(F, Ω);
    - - - - - - - - - - - - - -
12  v := pick_variable(F, q, Ω);
13  q0=q ∪ {(v = 0)};
14  (F, Ω0) ←DS-QSAT(F,Ω,q0);
15  (Ω^sym, Ω^asym) = split(Ω0, v);
16  if (Ω^asym = ∅) return(F, Ω0);
17  recover_vars_clauses(F, Ω^asym);
18  q1=q ∪ {(v = 1)};
19  (F, Ω1) ←DS-QSAT(F,Ω^sym,q1);
    - - - - - - - - - - - - - -
20  (F, Ω) ← merge(F, v, q, Ω0, Ω1);
21  return(F, Ω);}
```

Fig. 2. DS-QSAT procedure

DS-QSAT consists of three parts. In Figure 2, they are separated by dashed lines. In the first part, described in Subsections 5.3 and 5.4 in more detail, DS-QSAT checks for termination conditions and builds D-sequents for variables whose redundancy is obvious. In the second part (Subsection 5.5), DS-QSAT picks an unassigned non-redundant variable v and splits the current subspace into subspaces $v = 0$ and $v = 1$. Finally, DS-QSAT merges the results of branches $v = 0$ and $v = 1$ (Subsection 5.6).

5.2 Eager and lazy backtracking (DPLL as a special case of DS-QSAT)

Let $\boldsymbol{q}$ be the current partial assignment to variables of X and variable v be the variable assigned in $\boldsymbol{q}$ most recently. Let v be assigned a first value (left branch). Let C be a clause of F falsified after v is assigned in $\boldsymbol{q}$. In this case, procedure *update_Dseqs* of DS-QSAT (line 4 of Figure 2), derives a D-sequent $\boldsymbol{r} \rightarrow Z'$. Here $\boldsymbol{r}$ is the smallest subset of assignments of $\boldsymbol{q}$ falsifying C and Z' is a subset of the current set Z of the unassigned, non-redundant variables.

The version where $Z' = \emptyset$ i.e. where no D-sequent $\boldsymbol{r} \rightarrow Z'$ is derived by *update_Dseqs* will be called DS-QSAT with *lazy backtracking*. In our theoretical and experimental evaluation of DS-QSAT given in Sections 6 and 8 we used the version with lazy backtracking. The version of DS-QSAT where Z' is always equal to Z will be referred to as DS-QSAT with *eager backtracking*. DPLL is a special case of DS-QSAT where the latter employs eager backtracking. In this case, all unassigned variables are declared redundant and DS-QSAT immediately backtracks without trying to prove redundancy of variables of Z in some other ways.

5.3 Termination conditions

```
// q0=q∪{(v = 0)}; q1=q∪{(v = 1)};
// C0 = nil, C1 = nil if no clause of F
// is falsified by q0,q1 respectively

merge(F, v, q, Ω0, Ω1){
1   for (w ∈ (Vars(F) \ (Vars(q) ∪ v)) {
2     if (symmetric_in_v(Ω1, w))
            continue;
3     S0 = extract_Dseq(Ω0, w);
4     S1 = extract_Dseq(Ω1, w);
5     S = join(S0, S1, v);
6     Ω1 = (Ω1 ∪ {S}) \ {S1} ;}
    - - - - - - - - - - - - - - -
7   C0 = pick_falsif_clause(F, q0);
8   C1 = pick_falsif_clause(F, q1);
9   if ((C0 ≠ nil) and (C1 ≠ nil)) {
10    C = resolve(C0, C1, v);
11    F = F ∪ {C};
12    Ω1 = Ω1∪{falsif_clause_Dseq(C, v)};
13  else
14    Ω1 = Ω1∪{monot_var_Dseq(F, v, q)};
15  return(F, Ω1); }
```

Fig. 3. *merge* procedure

DS-QSAT reports unsatisfiability if the current formula F contains an empty clause (line 1 of Figure 2). DS-QSAT reports satisfiability if no clause of F is falsified by the current assignment $\boldsymbol{q}$ and every variable of F is either assigned in $\boldsymbol{q}$ or proved redundant in subspace $\boldsymbol{q}$ (line 10). Note that DS-QSAT uses slight optimization here by terminating before the D-sequent $\emptyset \rightarrow X$ is derived stating unconditional redundancy of variables of X in $\exists X[F]$.

If no termination condition is met but every variable of F is assigned or proved redundant, DS-QSAT ends the current call and returns F and Ω (lines 7,11). In contrast to operator *return*, the operator *exit* used in lines 1,10 eliminates the entire stack of nested calls of DS-QSAT.

5.4 Derivation of atomic D-sequents

Henceforth, for simplicity, we will assume that DS-QSAT derives **D-sequents of the form $\boldsymbol{r} \rightarrow \{v\}$** i.e. for single variables. A D-sequent $\boldsymbol{r} \rightarrow Z$ is then represented as $|Z|$ different D-sequents $\boldsymbol{r} \rightarrow \{v\}$, $v \in Z$.

In the two cases below, variable redundancy is obvious. Then DS-QSAT derives D-sequents we will call **atomic**. The first case, is when clause of F is falsified by the current assignment $\boldsymbol{q}$. This kind of D-sequents is derived by procedures *update_Dseqs* (line 4) and *finish_Dseqs*(line 6). Let v be the variable assigned in $\boldsymbol{q}$ most recently. Let C be a clause of F falsified after the current assignment to v is made. If v is assigned a first value (left branch), then, as we mentioned in Subsection 5.2, for *some* unassigned variables $w_1, \ldots, w_m$ that are not proved redundant yet, one can build D-sequents $\boldsymbol{r} \rightarrow \{w_1\}$,...,$\boldsymbol{r} \rightarrow w_m$. Here $\boldsymbol{r}$ is the shortest assignment falsifying C. So *update_Dseqs* may leave some unassigned variables non-redundant. On the contrary, *finish_Dseqs* is called in the right branch of v. In this case, for *every* unassigned variable w_i that is not proved redundant yet, D-sequent $\boldsymbol{r} \rightarrow \{w_i\}$ is generated. So on exit from *finish_Dseqs*, every variable of F is either assigned or proved redundant.

D-sequents of monotonic variables are the second case of atomic D-sequents. They are generated by procedure *monot_vars_Dseqs* (line 8) and by procedure *monot_var_Dseq* called when DS-QSAT merges results of branches (line 14 of Figure 3). Let $\boldsymbol{q}$ be the current partial assignment and v be a monotone unassigned

variable of F. Assume for the sake of clarity, that only clauses with positive polarity of v are present in $F_{\boldsymbol{q}}$. This means that every clause of F with literal $\overline{v}$ is either satisfied by $\boldsymbol{q}$ or contains a variable w proved redundant in $F_{\boldsymbol{q}}$. Then DS-QSAT generates D-sequent $\boldsymbol{r} \rightarrow \{v\}$ where $\boldsymbol{r}$ is formed from assignments of $\boldsymbol{q}$ as follows. For every clause C of F with literal $\overline{v}$ assignment $\boldsymbol{r}$ a) contains an assignment satisfying C or b) contains all the assignments of $\boldsymbol{s}$ such that D-sequent $\boldsymbol{s} \rightarrow \{w\}$ is active and w is a variable of C. Informally, $\boldsymbol{r}$ contains a set of assignments under which variable v becomes monotone.

5.5 Branching in DS-QSAT

When DS-QSAT cannot prove redundancy of some unassigned variables in the current subspace $\boldsymbol{q}$, it picks a non-redundant variable v for branching (line 12 of Figure 2). First, DS-QSAT calls itself with assignment $\boldsymbol{q_0} = \boldsymbol{q} \cup \{(v = 0)\}$. (Figure 2 shows the case when assignment $v = 0$ is explored in the left branch but obviously the assignment $v = 1$ can be explored before $v = 0$.) Then DS-QSAT partitions the returned set of D-sequents Ω_0 into Ω^{sym} and Ω^{asym}.

The set Ω^{sym} consists of the D-sequents $\boldsymbol{r} \rightarrow \{w\}$ of Ω_0 such that $v \notin \mathit{Vars}(\boldsymbol{r})$. The D-sequents of Ω^{sym} remain active in the branch $v = 1$. The set Ω^{asym} consists of the D-sequents $\boldsymbol{r} \rightarrow \{w\}$ such that $\boldsymbol{r}$ contains assignment $(v = 0)$. The D-sequents of Ω^{asym} are inactive in the subspace $v = 1$ and the variables whose redundancy is stated by those D-sequents reappear in the right branch. If $\Omega^{asym} = \emptyset$, there is no reason to explore the right branch. So, DS-QSAT just returns the set of D-sequents Ω_0 (line 16). Otherwise, DS-QSAT recovers the variables and clauses that were marked redundant after D-sequents from Ω^{asym} were derived (line 17) and calls itself with partial assignment $\boldsymbol{q_1} = \boldsymbol{q} \cup \{(v = 1)\}$.

5.6 Merging results of branches

After both branches of variable v has been explored, DS-QSAT merges the results by calling the *merge* procedure (line 20). The pseudocode of *merge* is shown in Figure 3. DS-QSAT backtracks only when every unassigned variable is proved redundant in the current subspace. The objective of *merge* is to maintain this invariant by a) replacing the currently D-sequents that depend on the branching variable v with those that are symmetric in v; b) building a D-sequent for the branching variable v itself.

The *merge* procedure consists of two parts separated in Figure 3 by the dotted line. In the first part, *merge* builds D-sequents for the variables of $X \setminus (\mathit{Vars}(\boldsymbol{q}) \cup \{v\})$. In the second part, it builds a D-sequent for the branching variable. In the first part, *merge* iterates over variables $X \setminus (\mathit{Vars}(\boldsymbol{q}) \cup \{v\})$. Let w be a variable of $X \setminus (\mathit{Vars}(\boldsymbol{q}) \cup \{v\})$. If the current D-sequent for w (i.e. the D-sequent for w from the set Ω_1 returned in the right branch) is symmetric in v, then there is no need to build a new D-sequent (line 2). Otherwise, a new D-sequent S for w that does not depend on v is generated as follows. Let S_0 and S_1 be the D-sequents for variable w contained in Ω_0 and Ω_1 respectively (lines 3,4). That is S_0 and

S_1 were generated for variable w in branches $v = 0$ and $v = 1$. Then D-sequent S is produced by joining S_0 and S_1 at variable v (line 5).

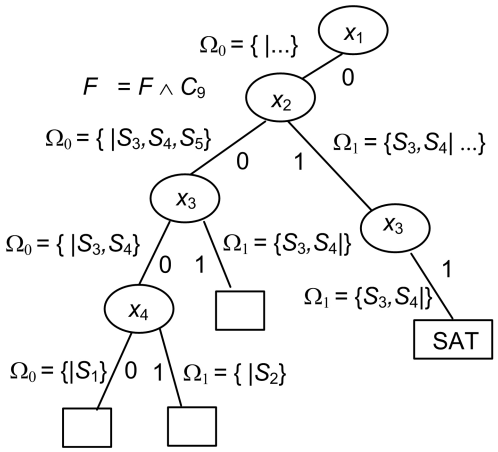

Fig. 4. Search tree built by DS-QSAT

Generation of a D-sequent for the variable v itself depends on whether node v (i.e the node of the search tree corresponding to v) is a conflict one. If so, F contains clauses C_0 and C_1 that have variable v and are falsified by $\boldsymbol{q_0}$ and $\boldsymbol{q_1}$ respectively. In this case, to make variable v redundant *merge* generates the resolvent C of C_0 and C_1 on variable v and adds C to F (lines 10,11). Then D-sequent $\boldsymbol{r} \rightarrow \{v\}$ is generated where $\boldsymbol{r}$ is the shortest assignment falsifying clause C (line 12).

If node v is not a conflict one, this means that clause C_0 and/or clause C_1 does not exist. Suppose, for example, that no clause C_0 containing variable v is falsified by $\boldsymbol{q_0}$. This means that every clause F with the positive literal of v is either satisfied by $\boldsymbol{q}$ or contains a variable redundant in subspace $\boldsymbol{q}$. In other words, v is monotone in $F_{\boldsymbol{q}}$ after removing the clauses with redundant variables. Then an atomic D-sequent is generated by *merge* (line 14) as described in Subsection 5.4.

Example 3. Here we show how DS-QSAT with *lazy backtracking* operates when solving the CNF formula F introduced in Example 1. Formula F consists of 8 clauses: $C_1 = \overline{x}_1 \vee \overline{x}_3$, $C_2 = \overline{x}_2 \vee x_3$, $C_3 = x_1 \vee x_2 \vee x_3$, $C_4 = x_2 \vee \overline{x}_3$, $C_5 = \overline{x}_1 \vee x_4 \vee x_5$, $C_6 = x_4 \vee \overline{x}_5$, $C_7 = \overline{x}_4 \vee x_5$, $C_8 = \overline{x}_1 \vee \overline{x}_4 \vee \overline{x}_5$. Figure 4 shows the search tree built by DS-QSAT. The ovals specify the branching nodes labeled by the corresponding branching variables. The label 0 or 1 on the edge connecting two nodes specifies the value made to the variable of the higher node. The rectangles specify the leaves of the search tree. The rectangle SAT specifies the leaf where DS-QSAT reported that F is satisfiable.

Every edge of the search tree labeled with value 0 (respectively 1) also shows the set of D-sequents Ω_0 (respectively Ω_1) derived when the assignment corresponding to this edge was made. The D-sequents produced by DS-QSAT are denoted in Figure 4 as $S_1, \ldots, S_5$. The values of $S_1, \ldots, S_5$ are given in Figure 5. When representing Ω_0 and Ω_1, we use the symbol '|' to separate D-sequents derived before and after a call of DS-QSAT. Consider for example, the set $\Omega_0 = \{|S_3, S_4, S_5\}$ on the path $x_1 = 0, x_2 = 0$. The set of D-sequents listed before '|' is empty in Ω_0. This means that no D-sequents had been derived when DS-QSAT was called with $\boldsymbol{q} = (x_1 = 0, x_2 = 0)$. On the exit of this invocation of DS-QSAT, D-sequents S_3, S_4, S_5 were derived. We use ellipsis after symbol

'|' for the calls of DS-QSAT that were not finished by the time F was proved satisfiable.

Below, we use Figures 4 and 5 to illustrate various aspects of the work of DS-QSAT.

Leaf nodes correspond to subspaces where every variable is either assigned or proved redundant. For example, the node on the path $(x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0)$ is a leaf because x_1, x_2, x_3, x_4 are assigned and x_5 is proved redundant.

Atomic D-sequents. D-sequents S_1, S_2, S_4, S_5 are atomic. For example, the D-sequent S_1 is derived in subspace $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0$ due to x_5 becoming monotone. S_1 is equal to $(x_1 = 0, x_4 = 0) \rightarrow \{s_5\}$ because only assignments $x_1 = 0, x_4 = 0$ are responsible for the fact that x_5 is monotone.

Branching in the presence of a conflict. On the path $x_1 = 0, x_2 = 0$, clauses C_3 and C_4 turned into unit clauses x_3 and $\overline{x}_3$ respectively. So no matter how first assignment to x_3 was made, one of these two clauses would get falsified. DS-QSAT made first assignment $x_3 = 0$ and falsified clause C_3. Since this was the *left branch* of x_3, DS-QSAT proceeded further to branch on variable x_4.

Merging results of branches. When branching on variable x_4, DS-QSAT derived sets $\Omega_0 = \{S_1\}$ and $\Omega_1 = \{S_2\}$ where S_1 is equal to $(x_1 = 0, x_4 = 0) \rightarrow \{x_5\}$ and S_2 is equal to $(x_1 = 0, x_4 = 1) \rightarrow \{x_5\}$. DS-QSAT merged the results of branching by joining S_1 and S_2 at the branching variable x_4. The resulting D-sequent S_3 equal to $(x_1 = 0) \rightarrow \{x_5\}$ does not depend on x_4.

D-sequents for branching variables. DS-QSAT generated D-sequents for branching variables x_4 and x_3. Variable x_4 was monotone in subspace $x_1 = 0, x_2 = 0, x_3 = 0$ because the clauses C_5,C_6 containing the positive literal of x_4 were not present in this subspace. C_5 was satisfied by assignment $x_1 = 0$ while C_6 contained variable x_5 whose redundancy was stated by D-sequent S_3 equal to $(x_1 = 0) \rightarrow \{x_5\}$. So the D-sequent S_4 equal to $(x_1 = 0) \rightarrow \{x_4\}$ was derived.

Variable x_3 was not monotone in subspace $\boldsymbol{q} = (x_1 = 0, x_2 = 0)$ because, in this subspace, clauses C_3 and C_4 turned into unit clauses x_3 and $\overline{x}_3$ respectively. So first, DS-QSAT *made* variable x_3 redundant by adding to F clause $C_9 = x_1 \vee x_2$ obtained by resolution of C_3 and C_4 on x_3. Note that C_9 is falsified in subspace $\boldsymbol{q}$. So the D-sequent S_5 equal to $(x_1 = 0, x_2 = 0) \rightarrow \{x_3\}$ was generated.

Reduction of the size of right branches. In the left branch of node x_2, the set of D-sequents $\Omega_0 = \{S_3, S_4, S_5\}$ was derived. D-sequent S_5 equal to $(x_1 = 0, x_2 = 0) \rightarrow \{x_3\}$ is not symmetric in x_2 (i.e. depends on x_2). On the other hand, S_3 and S_3 stating redundancy of x_4 and x_5 are symmetric in x_2. So only D-sequent S_5 was inactive in the right branch $x_2 = 1$. So only variable x_3 reappeared in this branch while x_4,x_5 remain redundant.

Termination. In subspace $\boldsymbol{q} = (x_1 = 0, x_2 = 1, x_3 = 1)$, every variable of F was assigned or redundant and no clause of F was falsified by $\boldsymbol{q}$. So DS-QSAT terminated reporting that F was satisfiable.

5.7 Correctness of DS-QSAT

$S_1 : (x_1 = 0, x_4 = 0) \to \{x_5\}$
$S_2 : (x_1 = 0, x_4 = 1) \to \{x_5\}$
$S_3 : (x_1 = 0) \to \{x_5\}$
$S_4 : (x_1 = 0) \to \{x_4\}$
$S_5 : (x_1 = 0, x_2 = 0) \to \{x_3\}$

Fig. 5. D-sequents of Figure 4

The proof of correctness of DS-QSAT can be performed by induction on the number of derived D-sequents. Since such a proof is very similar to the proof of correctness of the quantifier elimination algorithm we gave in [8], we omit it here. Below we just list the facts on which this proof of correctness is based.

- DS-QSAT derives correct atomic D-sequents.
- D-sequents obtained by the join operation are correct.
- DS-QSAT correctly reports satisfiability when every clause is either satisfied or proved redundant in the current subspace because D-sequents stating redundancy of variables are correct.
- New clauses added to the current formula are obtained by resolution and so are correct. So DS-QSAT correctly reports unsatisfiability when an empty clause is derived.

6 DS-QSAT on Compositional Formulas

In this section, we consider the performance of DS-QSAT on compositional formulas. We will say that a satisfiability checking **algorithm is compositional** if its complexity is *linear* in the number of subformulas forming a compositional formula. We prove that DS-QSAT with lazy backtracking is compositional and argue that DPLL-based SAT-solvers are not.

We say that **a formula $F(X)$ is compositional** if it can be represented as $F_1(X_1) \wedge \ldots \wedge F_k(X_k)$ where $X_i \cap X_j = \emptyset, i \neq j$. The motivation for our interest in such formulas is as follows. As we mentioned in Section 3, a practical formula F typically can be represented as $F_1(X_1, Y_1) \wedge \ldots \wedge F_k(X_k, Y_k)$ where X_i are internal variables of F_i and Y_i are communication variables. One can view compositional formulas as a degenerate case where $|Y_i| = 0, i = 1, \ldots k$ and so F_i do not talk to each other. Intuitively, an algorithm that does not scale well even when $|Y_i| = 0$ will not scale well when $|Y_i| > 0$.

From now on, we **narrow down the definition of compositional formulas** as follows. We will call formula $F_1(X_1) \wedge \ldots \wedge F_k(X_k)$ compositional if $X_i \cap X_j = \emptyset, i \neq j$ and all subformulas F_i, $i = 1, \ldots, k$ are equivalent modulo variable renaming/negation. That is F_j can be obtained from F_i by renaming some variables of F_i and then negating some variables of the result of variable renaming.

Proposition 2. *Let $F(X) = F_1(X_1) \wedge \ldots \wedge F_k(X_k)$ be a compositional formula. Let T be the search tree built by DS-QSAT with lazy backtracking when checking the satisfiability of F. The size of T is linear in k no matter how decision variables are chosen. (A variable $v \in X$ is a decision one if no clause of F that is unit in the current subspace contains v.)*

Proof. We will call a D-sequent $\boldsymbol{r} \rightarrow \{v\}$ **limited to subformula $\boldsymbol{F_i}$** if $(\mathit{Vars}(\boldsymbol{r}) \cup \{v\}) \subseteq \mathit{Vars}(F_i)$. The idea of the proof is to show that every D-sequent derived by DS-QSAT is limited to a subformula F_i. Then the size of T is limited by $|X| \cdot 2^d$ where $d = |\mathit{Vars}(F_1)| = \ldots = |\mathit{Vars}(F_k)|$. Indeed, when DS-QSAT flips the value of a variable v, only variables whose D-sequents depend on v reappear in the right branch of v. Since all D-sequents derived by DS-QSAT are limited to a subformula, the D-sequents depending on v are limited to subformula F_i such that $v \in \mathit{Vars}(F_i)$. This means that the number of variables that reappear in the right branch is limited by d. So the number of nodes of a right branch of T cannot be larger than 2^d. Hence the size of T cannot be larger than $|X| \cdot 2^d$ where $|X|$ is the maximum possible depth of T.

Now let us prove that every D-sequent derived by DS-QSAT is indeed limited to a subformula F_i. Since subformulas F_i,F_j, $i \neq j$ do not share variables, for any non-empty resolvent clause C, it is true that $\mathit{Vars}(C) \subseteq \mathit{Vars}(F_i)$ for some i. Then any atomic D-sequent built for a monotone variable v (see Subsection 5.4) is limited to the formula F_i such that $v \in \mathit{Vars}(F_i)$. DS-QSAT builds an atomic D-sequent of another type when a clause C produced by resolution on branching variable v is falsified in the current subspace. This D-sequent has the form $\boldsymbol{r} \rightarrow \{v\}$ where $\boldsymbol{r}$ is the shortest assignment falsifying C. Since $(\mathit{Vars}(C) \cup \{v\}) \subseteq \mathit{Vars}(F_i)$ where F_i is the subformula containing v, such a D-sequent is limited to F_i. Finally, a D-sequent obtained by joining D-sequents limited to F_i is limited to F_i □

Let SAT_ALG be a DPLL-based algorithm with clause learning. SAT_ALG cannot solve compositional formulas $F_1 \wedge \ldots \wedge F_k$ in the time linear in k for an arbitrary choice of decision variables. Since every resolvent clause can have only variables of one subformula F_i, the total number of clauses generated by SAT_ALG is linear in k. However, the time SAT_ALG has to spend to derive one clause is also linear k. When a conflict occurs, SAT_ALG backtracks to the decision level that is *relevant* to the conflict and is the closest to the conflict level. In the worst case, SAT_ALG has to undo assignments of all k subformulas. So in the worst case, the complexity of SAT_ALG is quadratic in k.

Notice that the DP procedure is compositional because clauses of different subformulas cannot be resolved with each other. However, as we mentioned in the introduction, this procedure is limited to one global variable order in which variables are eliminated. This limitation is the main reason why the DP procedure is outperformed by DPLL-based solvers. On the contrary, DS-QSAT is a branching algorithm that can use different variable orders in different branches (and DPLL-based SAT-solvers are a special case of DS-QSAT). So the machinery of D-sequents allows one to enjoy the flexibility of branching still preserving the compositionality of the algorithm.

7 Skipping Right Branches

```
DS-QSAT(F,q,Ω){
      .....
16    if (Ω^asym = ∅) return(F, Ω_0);
16.1  if (decision_var(q_0, v, F))
16.2     if (no_new_falsif_clause(q_0, F)){
16.3        S := branch_var_Dseq(F, v, q);
16.4        Ω:= recomp_Dseqs(Ω_0, S, q_0, F);
16.5        return(F, Ω ∪ {S}); }
17    recover_vars_clauses(F, Ω^asym);
      ....
21    return(F, Ω);}
```

Fig. 6. modified DS-QSAT procedure

In this section, we describe an optimization technique that can be used for additional pruning the search tree built by DS-QSAT. We will refer to this technique as SRB (Skipping Right Branches). The essence of SRB is that in some situations, DS-QSAT can use the D-sequents produced in the left branch of a variable v to build D-sequents that do not depend on v without exploration of the right branch of v.

This section is structured as follows. Subsection 7.1 gives pseudocode of the modification of DS-QSAT with SRB. Generation of D-sequents that do not depend on the current branching variable is explained in Subsection 7.2. Some notions introduced in [8] are recalled in Subsection 7.3. These notions are used in Subsection 7.4 to prove that the D-sequents derived by the modified part of DS-QSAT are correct.

7.1 Modified DS-QSAT

The modification of DS-QSAT due to adding the SRB technique is shown in Figure 6 (lines 16.1-16.5). SRB works as follows. Suppose that DS-QSAT has backtracked from the the left branch of v. Let $\boldsymbol{q}$ be the set of assignments made by DS-QSAT before variable v. We will follow the assumption of Figure 2 that the first value assigned to v is 0. In such a case, DS-QSAT of Figure 2 just explores the right branch $v = 1$ (line 19). The essence of DS-QSAT with SRB is that if a condition described below is satisfied, the right branch is skipped. Instead, DS-QSAT does the following. First, it builds a correct D-sequent of the branching variable v depending only on assignments to $\boldsymbol{q}$ (line 16.3). Then, every D-sequent $\boldsymbol{r} \rightarrow \{w\}$ of Ω_0 where $\boldsymbol{r}$ contains assignment $(v = 0)$ is recomputed (line 16.4).

Let $\boldsymbol{q_0}$ denote assignment $\boldsymbol{q}$ extended by $(v = 0)$. The condition under which the SRB technique is applicable is that no clause of F having literal v (i.e. the positive literal of variable v) is falsified by $\boldsymbol{q_0}$. This means that every clause with literal v is either satisfied by $\boldsymbol{q}$ or has a variable that is redundant in subspace $\boldsymbol{q_0}$. This condition is checked on line 16.2.

The SRB technique is used in the modification of DS-QSAT shown in Figure 6 only if v is a decision variable (line 16.1). The reason is as follows. Suppose that this is not the case, i.e. v is in a unit clause C of $F_{\boldsymbol{q}}$. In this case, in the left branch (respectively right branch), DS-QSAT assigns v the value that satisfies C (respectively falsifies C). But since DS-QSAT immediately backtracks if a new clause gets falsified in the right branch, pruning the left branch in this case does not save any work.

7.2 D-sequents generated by modified DS-QSAT

Let $F(X)$ be a CNF formula. Let $\boldsymbol{q}$ be a partial assignment to variables of X. Let v be a variable of $X \setminus \mathit{Vars}(\boldsymbol{q})$. Let $\boldsymbol{q_0}$ denote the assignment $\boldsymbol{q} \cup \{(v = 0)\}$. Let Ω_0 be a set of D-sequents active in the subspace specified by $\boldsymbol{q_0}$. Let every clause of F that has literal v is either satisfied by $\boldsymbol{q}$ or has a variable whose redundancy is stated by a D-sequent of Ω_0.

The procedure *branch_var_Dseq* of Figure 6 generates D-sequent $\boldsymbol{r} \rightarrow \{v\}$ such that for every clause C containing literal v

- C is satisfied by $\boldsymbol{r}$ or
- C contains a variable w whose redundancy is stated by a D-sequent $\boldsymbol{s} \rightarrow \{w\}$ of Ω_0 and $\boldsymbol{s} \leq (\boldsymbol{r} \cup \{(v = 0)\}$.

The procedure *recomp_Dseqs* of Figure 6 works as follows. For every D-sequent $\boldsymbol{e} \rightarrow \{w\}$ of Ω_0 such that $\boldsymbol{e}$ contains assignment $(v = 0)$, *recomp_Dseqs* generates a D-sequent $\boldsymbol{e'} \rightarrow \{w\}$. The assignment $\boldsymbol{e'}$ is obtained from $\boldsymbol{e}$ by replacing assignment $(v = 0)$ with the assignments of $\boldsymbol{r}$ of the D-sequent $\boldsymbol{r} \rightarrow \{v\}$ generated for the branching variable v.

7.3 Recalling some notions

In this subsection, we recall some notions introduced in [8] that are used in the proofs of Subsection 7.4. Let $F(X)$ be a CNF formula. We will refer to a complete assignment to variables of X as a **point**. A point $\boldsymbol{p}$ is called a ***Z*-boundary point** of F if

- $\boldsymbol{p}$ falsifies F
- every clause of F falsified by $\boldsymbol{p}$ contains a variable of Z
- Z is minimal i.e. no proper subset of Z satisfies the property above

A Z-boundary point $\boldsymbol{p}$ is called ***Y*-removable** in F where $Z \subseteq Y \subseteq X$ if $\boldsymbol{p}$ cannot be turned into an assignment satisfying F by changing values of variables of Y. If a Z-boundary point is Y-removable, then one can produce a clause C that is a) falsified by $\boldsymbol{p}$; b) implied by F and c) does not have any variables of Z. After adding C to F, $\boldsymbol{p}$ is not a Z-boundary point anymore, hence the name removable.

We will call a Y-removable point **just removable** if $Y = X$. It is not hard to see, that every Z-boundary point of a satisfiable (respectively unsatisfiable) formula F is unremovable (respectively removable).

Proposition 3. *Let $F(X)$ be a CNF formula and $\boldsymbol{q}$ be a partial assignment to variables of X. A set of variables Z is not redundant in $\exists X[F]$ in subspace $\boldsymbol{q}$, if and only if there is a Z-boundary point of $F_{\boldsymbol{q}}$ that is removable in F.*

The proof of this proposition is given in [8].

7.4 Correctness of D-sequents generated by modified DS-QSAT

Proposition 4. *The D-sequent generated by procedure branch_var_Dseq of Figure 6 described in Subsection 7.2 is correct.*

Proof. Assume the contrary i.e. D-sequent $\boldsymbol{r} \rightarrow \{v\}$ does not hold. From Proposition 3. it follows that there is a $\{v\}$-boundary point $\boldsymbol{p}$ such that $\boldsymbol{r} \leq \boldsymbol{p}$. This also means that F is unsatisfiable. Indeed, if F is a satisfiable, then every variable of F is already redundant and so any D-sequent holds $\boldsymbol{r} \rightarrow \{v\}$.

Let us assume that v is equal to 0 in $\boldsymbol{p}$. If v equals 1 in $\boldsymbol{p}$, one can always flip the value of v obtaining the point that is either a $\{v\}$-boundary point or a satisfying assignment (Lemma 1 of [8]). Since the assumption we made implies that F is unsatisfiable, flipping the value of v produces a $\{v\}$-boundary point.

Let G be the set clauses falsified by point $\boldsymbol{p}$. Let C be a clause of G. Since $\boldsymbol{p}$ is a $\{v\}$-boundary point, then C contains literal v. Note that under the assumption of the proposition to be proved, if a clause of F with literal v is not satisfied by $\boldsymbol{r}$, this clause has to contain a redundant variable w such that D-sequent $\boldsymbol{s} \rightarrow \{w\}$ of Ω_0 holds and $\boldsymbol{s} \leq (\boldsymbol{r} \cup \{(v = 0)\}$. Let Z be a minimal set of variables of X that are present in clauses of G and whose redundancy is stated by D-sequents of Ω_0. Then $\boldsymbol{p}$ is a Z-boundary point of F. Since F is unsatisfiable, this point is removable. Then from Proposition 3 it follows that the variables of Z are not in redundant in $F_{\boldsymbol{r'}}$ where $\boldsymbol{r'} = \boldsymbol{r} \cup \{(v = 0)\}$. Contradiction.

Proposition 5. *The D-sequents generated by the recomp_Dseqs procedure of Figure 6 described in Subsection 7.2 are correct.*

Proof. Assume the contrary i.e. the D-sequent $\boldsymbol{e'} \rightarrow \{w\}$ obtained from a D-sequent $\boldsymbol{e} \rightarrow \{w\}$ of Ω_0 does not hold. This means that there is a $\{w\}$-boundary point $\boldsymbol{p}$ such that $\boldsymbol{e'} \leq \boldsymbol{p}$. It also means that F is unsatisfiable. Let us consider the following two cases.

- Variable v is assigned 0 in $\boldsymbol{p}$. Then there is a removable $\{w\}$-boundary point in subspace $\boldsymbol{e}$ and so the D-sequent $\boldsymbol{e} \rightarrow \{w\}$ of Ω_0 does not hold. Contradiction.
- Variable v is assigned 1 in $\boldsymbol{p}$. Let $\boldsymbol{p'}$ be the point obtained from $\boldsymbol{p}$ by flipping the value of v. Let G and G' be the clauses of F falsified by $\boldsymbol{p}$ and $\boldsymbol{p'}$ respectively. Denote by G'' the set of clauses $G' \setminus G$. This set consists only of clauses having literal v because these are the only new clauses that may get falsified after flipping the value of v from 1 to 0. Then using reasoning similar to that of Proposition 4, one concludes that every clause of G'' contains a variable u such that D-sequent $\boldsymbol{s} \rightarrow \{u\}$ of Ω_0 holds and $\boldsymbol{s} \leq (\boldsymbol{r} \cup \{(v = 0)\})$ where $\boldsymbol{r}$ is the assignment of the D-sequent $\boldsymbol{r} \rightarrow \{v\}$ generated for the branching variable v. Let Z be a minimal set of such variables. Then any clause of G' either contains variable w or a variable of Z. Hence $\boldsymbol{p'}$ is a $(Z \cup \{w\})$-boundary point. Since our assumption implies unsatisfiability of F, this boundary point is removable. Let $\boldsymbol{g}$ be equal to $\boldsymbol{e'} \cup \{(v = 0)\}$. Note that since $\boldsymbol{r} \leq \boldsymbol{e'}$ and $\boldsymbol{g}$ contains assignment $(v = 0)$ every variable of Z is

redundant in $F_{\boldsymbol{g}}$. On the one hand, since $\boldsymbol{e} \leq \boldsymbol{g}$, variable w is redundant in $F_{\boldsymbol{g}}$ as well. So the variables of $Z \cup \{w\}$ are redundant in $F_{\boldsymbol{g}}$. On the other hand, $\boldsymbol{g} \leq \boldsymbol{p'}$ and so $F_{\boldsymbol{g}}$ contains a $(Z \cup \{w\})$-boundary point $\boldsymbol{p'}$ that is removable in F. From Proposition 3, it follows that variables of $(Z \cup \{w\})$ are not redundant in $F_{\boldsymbol{g}}$. Contradiction.

8 Experiments

In this section, we compare DS-QSAT with some well-known SAT-solvers on two sets of compositional and non-compositional formulas. In experiments, we used the optimization technique described in Section 7. Although, using this technique was not crucial for making our points, it allowed to improve the runtimes of DS-QSAT.

Obviously, this comparison by no way is comprehensive. Our objective here is as follows. In Subsection 5.2, we argued that DPLL-based SAT-solvers is a special case DS-QSAT when it uses eager backtracking. One may think that due to great success of modern SAT-solvers, this version of DS-QSAT is simply always the best. In this section, we show that is not the case. We give an example of meaningful formulas where the opposite strategy of lazy backtracking works much better. This result confirms the theoretical prediction of Section 6.

Table 1. *Solving compositional formulas*

#copies $\times 10^3$	#vars $\times 10^3$	#clauses $\times 10^3$	minisat (s.)	rsat (s.)	picosat (s.)	ds-qsat (s.)
5	80	170	9.1	5.1	4.2	0.7
10	160	340	110	28	20	1.6
20	320	680	917	143	80	3.3
40	640	1,360	> 1hour	621	305	7.2
80	1,280	2,720	> 1hour	2,767	1,048	15

The results of experiments with the first set of formulas are shown in Tables 1, 2 and 3 This set consists of compositional formulas $F_1(X_1) \wedge \ldots \wedge F_k(X_k)$ where $X_i \cap X_j = \emptyset$. Every subformula F_i is obtained by renaming/negating variables of the same satisfiable CNF formula describing a 2-bit multiplier. Since every subformula F_i is satisfiable, then formula $F_1 \wedge \ldots \wedge F_k$ is satisfiable too for any value of k.

Table 2. *Statistics of Picosat and DS-QSAT on compositional formulas*

#copies $\times 10^3$	picosat			ds-sat		
	#cnfl. $\times 10^3$	#dec. $\times 10^6$	#impl. $\times 10^6$	#cnfl. $\times 10^3$	#dec. $\times 10^3$	#impl. $\times 10^3$
5	0.8	3	12	0.8	25	62
10	1.5	10	39	1.5	50	122
20	3.0	40	144	3.2	100	245
40	5.5	138	489	6.5	199	493
80	9.7	443	1,533	13.0	399	985

In the DIMACS format that we used in experiments, a variable's name is a number. In the formulas of Table 1, the variables were named so that the DIMACS names of variables of different subformulas F_i interleaved. The objective of negating variables was to make sure that if an assignment $\boldsymbol{s}$ to the variables of X_i satisfies F_i, the same assignment of the corresponding variables of X_j is unlikely to satisfy F_j.

In Table 1, we compare DS-QSAT with Minisat (version 2.0), RSat (version 2.01) and Picosat (version 913) on compositional formulas. These formu-

las are different only in the value of k. The first three columns of this table show the value of k, the number of variables and clauses in thousands. The last four columns show the time taken by Minisat, RSat,Picosat and DS-QSAT to solve these formulas (in seconds). DS-QSAT significantly outperforms these three SAT-solvers. As predicted by Proposition 2, DS-QSAT shows linear complexity. On the other hand, the complexity of each of the three SAT-solvers is proportional to $m \cdot k^2$ where m is a constant.

Table 2 provides some statistics of the performance of Picosat and DS-QSAT on the formulas of Table 1. The second, third and fourth columns give the number of conflicts (in thousands), number of decision and implied assignments (in millions) for Picosat. In the following three columns, the number of conflict nodes of the search tree, number of decision and implied assignments (in thousands) are given for DS-QSAT. The results of Table 2 show that the numbers of conflicts of Picosat and those of conflict nodes of DS-QSAT are comparable. Besides, for both programs the dependence of these numbers on k is linear. However, the numbers of decision and implied assignments made by Picosat and DS-QSAT differ by three orders of magnitude. Most importantly, the number of assignments made by DS-QSAT (both decision and implied) grows linearly with k. On the other hand, the dependence of the number of assignments made by Picosat on k is closer to quadratic for both decision and implied assignments.

Table 3 provides some additional statistics characterizing the performance of DS-QSAT on the formulas of Table 1. The second column specifies the maximum number of conflict variables that appeared on a path of the search tree. A variable v is a conflict one if after making an assignment to v a new clause of F gets falsified. This column shows that DS-QSAT kept branching even after thousands of conflicts occurred on the current path.

Table 3. *More statistics of DS-QSAT for compositional formulas*

#vars $\times 10^3$	max confl vars	#assgn. vars in sol. (%)	max right branch
80	505	2	14
160	1,027	0.1	14
320	1,956	1	14
640	3,795	1	14
1,280	7,351	0.2	14

DS-QSAT reports that a formula is satisfiable when the current assignment $\boldsymbol{q}$ does not falsify a clause of F and every variable of F that is not assigned in $\boldsymbol{q}$ is proved redundant. The third column of Table 3 gives the value of $|Vars(\boldsymbol{q})|/|Vars(F)|$ (in percent) at the time DS-QSAT proved satisfiability. Informally, this value shows that DS-QSAT established satisfiability of F knowing only a very small fragment of a satisfying assignment. The last column of Table 3 shows the maximum number of non-redundant unassigned variables that appeared in a right branch of the search tree. The number of variables in subformulas F_i we used in experiments was equal to 16. As we showed in Proposition 2, in the search tree built by DS-QSAT for a compositional formula, the number of free variables that may appear in a right branch is bounded by $|V(F_i)|$ i.e by 16. Our experiments confirmed that prediction. The fact that the size of right branches is so small means that when solving a formula F of Table 1, DS-QSAT dealt only with very small fragments of F.

Table 4. *Solving non-compositional formulas*

#sub-form. $\times 10^3$	#vars $\times 10^3$	minisat (s.)	rsat (s.)	picosat (s.)	ds-qsat (s.)	ds-qsat* (s.)
5	75	5.0	3.2	3.6	5.1	0.4
10	150	34	21	15	13	1.0
20	300	548	79	57	30	2.3
40	600	>1hour	430	231	57	5.9
80	1,200	>1hour	1,869	859	127	19

Generally speaking, the problems with compositional formulas can be easily fixed by solving independent subformulas separately. Such subformulas can be found in linear time by looking for strongly connected components of a graph relating clauses that share a variable. To eliminate such a possibility we conducted the second experiment. In this experiment, we compared DS-QSAT and the three SAT-solvers above on *non-compositional formulas*. Those formulas were obtained from the same subformulas F_i obtained from a CNF formula specifying a 2-bit multiplier by renaming/negating variables. However, now, renaming was done in such a way that every pair of subformulas F_i,F_{i+1}, $i = 1, \ldots, k-1$ shared exactly one variable. So now formulas $F = F_1(X_1) \wedge \ldots \wedge F_k(X_k)$ we used in experiments did not have any independent subformulas. Table 4 shows the results of the second experiment (all formulas are still satisfiable). The first two columns of Table 4 specify the value of k (i.e. the number of subformulas F_i) and the number of variables of F in thousands. The next four columns give the runtimes of Minisat, RSat, Picosat and DS-QSAT in seconds. These runtimes show that DS-QSAT still outperforms these three SAT-solvers and scales better.

The last column of Table 4 illustrates the ability of D-sequents to take into account formula structure. In this column, we give the runtimes of DS-QSAT when it first branched on communication variables (i.e. ones shared by subformulas F_i). So in this case, DS-QSAT had information about formula structure. The results show that the knowledge of communication variables considerably improved the performance of DS-QSAT.

Table 5. *Statistics of DS-QSAT for non-compositional formulas*

#vars $\times 10^3$	max confl vars	#assgn. vars in sol. (%)	max right branch	max right branch (%)	max right branch*
75	461	4	338	0.5	11
150	903	1	475	0.3	11
300	1,765	1	571	0.2	11
600	3,512	2	773	0.1	11
1,200	7,029	1	880	0.1	11

Table 5 gives some statistics describing the performance of DS-QSAT on the formulas of Table 4. The second and third columns of Table 5 are similar to the corresponding columns of Table 3. A lot of conflicts occurred on a path of the search tree built by DS-QSAT and by the time DS-QSAT reported satisfiability, only a small fragment of a satisfying assignment was known. The fourth column shows the maximum size of a right branch of the search tree built by DS-QSAT (in terms of the number of non-redundant variables). The next column gives the ratio of the maximum size of a right branch and the total number of variables (in percent). Notice, that now the maximum size of right branches is much larger than in the case of compositional formulas. Nevertheless, again, DS-QSAT dealt only with very small fragments of the formula. The last column gives the maximum size of

right branches when DS-QSAT first branched on communication variables. Such structure-aware branching allowed DS-QSAT to dramatically reduce the size of right branches, which explains why DS-QSAT had much faster runtimes in this case (shown in the last column of Table 4).

Although DS-QSAT performed well on the formulas we used in experiments, lazy backtracking is too extreme to be successful on a more general set of benchmarks. Let F be a formula to be checked for satisfiability. Let a clause C of F be falsified by the current assignment and Z be the set of unassigned variables. At this point, any variable $v \in Z$ is redundant due to C being falsified. Lazy backtracking essentially assumes that by keeping branching in the presence of the conflict one will find a better explanation of redundancy of v. A less drastic approach is as follows. Once clause C gets falsified, a D-sequent $\boldsymbol{r} \rightarrow Z'$ is derived where $\boldsymbol{r}$ is the shortest assignment falsifying C and Z' consists of some variables of Z that are related to clause C. For example, if C is in a subformula G of F specifying a design block it may make sense to form Z' of the unassigned variables of G.

9 Background

In 1960, Davis and Putnam introduced a QSAT-solver that is now called the DP procedure [3]. Since it performed poorly even on small formulas, a new algorithm called the DPLL procedure was introduced in 1962 [2]. Two major changes were made in the DPLL procedure in comparison to the DP procedure. First, the DPLL procedure employed branching and could use different variable order in different branches. Second, it changed the semantics of variable elimination that the DP procedure was based on. Instead, the semantics of elimination of unsatisfiable assignments was introduced. The DPLL procedure backtracks as soon as it finds out that the current partial assignment cannot be extended to a satisfying assignment. Such eager backtracking is a characteristic feature of SAT-solvers i.e. algorithms proving satisfiability by producing a satisfying assignment.

The first change has been undoubtedly a great step forward. The DP procedure eliminates variables in one particular global order which makes this procedure very inefficient. The second change however has its pros and cons. On the one hand, DPLL has a very simple and natural semantics, which facilitated the great progress in SAT-solving seen in the last two decades [15,18,16,11,6,17,1]. On the other hand, as we argued before, the necessity to generate a satisfying assignment to prove satisfiability deprived DPLL-based SAT-solvers of powerful transformations preserving equisatisfiability rather than functional equivalence.

Generally speaking, transformations preserving equisatisfiability are routinely used by modern algorithms, but their usage is limited one way or another. For example such transformations are employed in preprocessing where some variables are resolved out [5] or redundant clauses are removed [13]. However, such transformations have a limited scope: they are used just to simplify the original formula that is then passed to a DPLL-based SAT-solver. Second, transforma-

tions preserving equisatisfiability are ubiquitous in algorithms on circuit formulas e.g. in ATPG algorithms [4]. Such algorithms often exploit the fact that a gate becomes unobservable under some partial assignment $\boldsymbol{r}$. In terms of variable redundancy, this means that the variable v specifying the output of this gate is redundant in subspace $\boldsymbol{r}$. Importantly, this redundancy is defined with respect to a formula where assignments to non-output variables do not matter. Such variables can be viewed as existentially quantified and the discarding of clauses containing redundant non-output variables does not preserve functional equivalence. However, such transformations are restricted only to formulas generated off circuits and do not form a complete calculus. Typically, these transformations are used in the form of heuristics.

The machinery of D-sequents was introduced in [8,9]. In turn, the notion of D-sequents and join operation were inspired by the relation between variable redundancy and boundary point elimination [10,12]. In [8,7], we formulated a method of quantifier elimination called DDS (Derivation of D-Sequents). Since QSAT is a special case of the quantifier elimination problem, DDS can be used to check satisfiability. However, since DDS employs eager backtracking, such an algorithm is a SAT-solver rather than a QSAT-solver. In particular, as we showed in [8], the complexity of DDS on compositional formulas is quadratic in the number k of subformulas, while the complexity of DS-QSAT is linear in k.

10 Conclusion

The results of this paper lead to the following three conclusions.

1) DPLL-based procedures have scalability issues. These issues can be observed even on compositional formulas i.e. on formulas with a very simple structure. Arguably, the root of the problem, is that DPLL-procedures are designed to prove satisfiability by producing a satisfying assignment. This deprives such procedures from using powerful transformations that preserve equisatisfiability rather than functional equivalence.

2) D-sequents are an effective tool for building scalable algorithms. In particular, the algorithm DS-QSAT we describe in the paper scales well on compositional formulas. The reason for such scalability is that DS-QSAT scarifies the ability to generate satisfying assignments to tap into the power of transformations preserving only equisatisfiability. The essence of transformations used by DS-QSAT is to discard large portions of the formula that are proved to be redundant in the current subspace. The results of experiments with DS-QSAT on two simple classes of compositional and non-compositional formulas show the big promise of algorithms based on D-sequents.

3) In this paper, we have only touched the tip of the iceberg. A great deal of issues needs to be resolved to make QSAT-solving by D-sequents practical.

11 Acknowledgment

This work was funded in part by NSF grant CCF-1117184.

References

1. A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
2. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
3. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
4. R. Drechsler, T. Juntilla, and I.Niemelä. Non-Clausal SAT and ATPG. In *Handbook of Satisfiability*, volume 185, chapter 21, pages 655–694. IOS Press, 2009.
5. N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.
6. N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, Santa Margherita Ligure, Italy, 2003.
7. E.Goldberg and P.Manolios. Quantifier elimiantion by dependency sequents. Accepted for publication at FMCAD-2012.
8. E.Goldberg and P.Manolios. Quantifier elimination by dependency sequents. Technical Report arXiv:1201.5653v2 [cs.LO], 2012.
9. E.Goldberg and P.Manolios. Removal of quantifiers by elimination of boundary points. Technical Report arXiv:1204.1746v2 [cs.LO], 2012.
10. E. Goldberg. Boundary points and resolution. In *SAT-09*, pages 147–160, Swansea, Wales, United Kingdom, 2009. Springer-Verlag.
11. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. *Discrete Appl. Math.*, 155(12):1549–1561, 2007.
12. Eugene Goldberg and Panagiotis Manolios. Sat-solving based on boundary point elimination. In *Haifa Verification Conference*, volume 6504 of *Lecture Notes in Computer Science*, pages 93–111, 2010.
13. Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *TACAS*, pages 129–144, 2010.
14. N. Kayal M. Agrawal and N. Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004.
15. J. Marques-Silva and K. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD-96*, pages 220–227, Washington, DC, USA, 1996.
16. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC-01*, pages 530–535, New York, NY, USA, 2001.
17. K. Pipatsrisawat and A. Darwiche. Rsat 2.0: Sat solver description. Technical Report D–153, Autom. Reas. Group, Comp. Sci. Depart., UCLA, 2007.
18. H. Zhang. Sato: An efficient propositional prover. In *CADE-97*, pages 272–275, London, UK, 1997. Springer-Verlag.