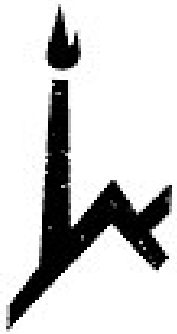
Backtrack Search Algorithms for Propositional Logic Satisfiability : Review and Innovations

Alexander Nadel

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Supervised by Prof. Eliezer Lozinski

November, 2002



1 Introduction

1.1 General

Propositional satisfiability problem (SAT) is a problem of determining for a propositional calculus formula whether it contains a contradiction and if it doesn't, finding a satisfying assignment for it.

SAT holds a central place in large family of computationally intractable NP-complete problems ([1],[2]). Therefore, it's unlikely that there is an algorithm solving it in reasonable time in all cases. Nevertheless, there are algorithms capable of solving many instances pretty fast. There is a big interest in such algorithms, since the SAT problem has extensive applications in Artificial Intelligence, Electronic Design Automation and many other fields of Computer Sciences and Engineering.

Incomplete SAT algorithms, based on local search are capable to prove satisfiability on many random and real-life SAT formulas ([3],[4],[5],[6]).

On the other hand, backtrack search complete algorithms, based on DPLL algorithm ([7],[8]) have been used for solving large number of real-world instances of SAT, a significant fraction of which may be unsatisfiable. Some of the most efficient algorithms are rel_sat[9], GRASP[10], SATO[11], Chaff[12], BerkMin[19]. There are also complete SAT algorithms based on other ideas ([13],[14], [15] for overview), but backtrack search algorithms, based on DPLL are currently much more efficient for most of benchmarks.

1.2 This Work's Contribution

In the next section of this work, we'll provide an extensive overview of existing backtrack search complete algorithms.

In section 4 we'll propose innovations in many aspects of backtrack search algorithms structure.

This includes a new algorithm generalising DPLL, called CRSAT. This also includes improvements to non-chronological backtracking, new pruning methods, new heuristics, new relevance-based learning technique, new effective data structure, new idea allowing

not repeating paths after search restarts and an idea of dynamically changing(and minimising the number of) decision variables. All(but 2 new pruning techniques) of those innovations are applicable for both DPLL and CRSAT.

In section 5 we'll introduce a new SAT solver(Jerusat) based on ideas brought in section 4 and bring empirical results, comparing its performance with performance of existing solvers.

In Appendix A, we'll bring detailed pseudo-code sections and formal proofs that will be usually omitted in the main part of the work.

In Appendix B we'll bring a detailed description of the new Jerusat solver, together with extensive analysis of its performance.

2 Basic Definitions

In this section, we'll bring definitions, which are relevant to every part of this work. Definitions and notations, which can be understood only in specific context, will be brought in relevant sections.

A conjunctive normal form (CNF) propositional formula *S* on *n* binary variables $A_1 \ldots A_n$ is a conjunction of *m* clauses $\omega_1 \ldots \omega_m$. Each of the clauses is a disjunction of one or more literals, where a *literal* is an occurrence of a variable or its negation. For each clause ω and variable A: if a positive occurrence of A is in ω, we'll say that $\omega(A)=T$; if a negative occurrence of A is in ω, we'll say that $\omega(A)=F$; otherwise we'll say that $\omega(A)=X$.
In this work a disjunction will be denoted by v, a conjunction by Λ and a logical not by $\neg$.
A CNF formula *S* denotes an *n*-variable Boolean function $f(x_1 \ldots x_n)$.
In this work, we'll suppose that S doesn't contain tautological clauses, i.e. clauses that contain an occurrence of a literal and its negation. Anyway, such clauses can be easily removed without changing *f*.
The SAT problem is concerned with finding an assignment (or interpretation) to $A_1 \ldots A_n$ that makes *f* equal to T (for simplicity, the value *TRUE* for a variable will be denoted as *T* and *FALSE* as *F*) or proving that the function have no such assignment. If such an assignment exists, it is referred to as *a model* or *a satisfying assignment.* Any assignment, which is not a model, is referred to as an *unsatisfying assignment*. If *f* has a model S is called *satisfiable*; otherwise it's called *unsatisfiable.*

An assignment may also be denoted as a function α, where:

- $\alpha(A_i)=T$ if A_i is T under α
- $\alpha(A_i)=F$ if A_i is F under α
- $\alpha(A_i)=X$ if A_i is not given value under α

If there exists i, s.t. $\alpha(A_i)=X$, than α is a *partial assignment*, otherwise it's a *complete assignment*.

We'll say that a positive literal A is satisfied by α if α(A)=T. We'll say that a negative literal ¬A is satisfied by α if α(A)=F. Unsatisfiability of A or ¬A by α is defined similarly.
We'll say that A and ¬A are undefined or not assigned a value under α, if α(A)=X. Otherwise, we'll say that (literal or variable)A is assigned a value under α.

It may also be convenient to denote an assignment for a formula *S* as set of literals. For example, if $S = (A) \Lambda (\neg A \vee B) \Lambda (\neg B \vee \neg C)$ than {¬A,B,C} is an unsatisfying assignment and {*A,B,¬C}* is a model.

We'll denote by S|α a new CNF formula, which is a result of :

1. Removing from S clauses, where at least one literal is satisfied by α
2. Removing from S clauses negations of literals which are unsatisfied by α

A literal *A* is referred to as *pure literal* if ¬*A* doesn't appear is *S.* A clause ω is referred to as *unit clause* if ω consists of a single literal.

Finally, we would like to state, that we'll use the soundness and completeness theorem for the propositional logic without mentioning it explicitly.

3 DPLL Algorithm with Enhancements Overview

In section 3.1 we'll introduce the basic DPLL algorithm. Extended overview of existing methods, which boost the performance of modern SAT solvers, based on DPLL algorithm is brought in 3.2.
In 3.3 and 3.4 we'll bring extended overviews of modern branching heuristics and data structures for CNF representation. Both are critical for performance of DPLL-based SAT solvers.
In 3.5 we'll briefly overview pre-processing techniques, which are methods to be applied on a CNF formula before running a SAT solver on it.

3.1 Davis-Putnam-Loveland-Longmann Algorithm (DPLL)

In this section we'll describe the famous DPLL algorithm.

3.1.1.1 Working Principle of DPLL Algorithm

DPLL algorithm ([7]) (it is sometimes called DP in literature) is a backtrack search algorithm, which searches the assignments space for a model.
Briefly, it works as follows:
It holds a partial assignment α and tries to extend it by assigning values to non-assigned variables. The value may be imposed by other variables' values under α or chosen.
It does so until α becomes a full assignment in which case the formula is satisfiable or there is clause that is F under α.
In the latter case it backtracks, i.e. it unassigns lately assigned variables until it finds a variable, which can be assigned to another value. If such variable isn't found, the input formula is unsatisfiable, otherwise it goes on with the backtrack search process.

In subsection 3.1.1.2 we'll bring a formal definition of DPLL algorithm

3.1.1.2 Recursive DPLL Algorithm

Let S be a CNF formula and α be an empty (partial) assignment. The DPLL algorithm below returns either {T,α}, in which case S is satisfiable and α is a model or {F,{}}, in which case S is unsatisfiable.

DPLL(S,α={}) :

2 conditions : Check the following 2 conditions

1) If S is empty, return {T,α}

2) If S contains an empty clause, return {F,{}}

3 rules : Apply one of the 3 rules below(choose any, if more than 1 can be applied)

1) Pure-literal rule – if there exists a pure literal *A* in *S* (*s.t. ¬*A* doesn't appear in *S*),* then β=αU{*A*,T); Return DPLL(S|β,β).

2) *Unit clause rule* – if S contains a unit clause {A} (*clause consisting of a single literal*) then β=AU{*A*,T};Return DPLL(S|β,β)

3) Splitting rule – Choose a literal A. Choose arbitrarily between :

(a) β=αU{*A*,T}; γ=αU{*A*,F}

(b) γ=αU{*A*,T}; β=αU{*A*,F}

If DPLL(S|β)={T,β} return {T,β}; else return DPLL(S|γ)

3.1.1.3 Exploiting Decisions Freedom in DPLL Algorithm

One can notice that there are two important choices to be made at each stage of the DPLL algorithm: choosing literals for the splitting rule and choosing the implication order of the 3 rules.

3.1.1.3.1 Choosing Literals for the Splitting Rule

In [18] it's suggested that branching heuristics is less important that efficient search pruning techniques, but this statement is disproved by results shown in [19] and [12]. According to those works SAT solvers BerkMin and zChaff respectively achieve much better results using good branching heuristics. We'll overview different branching heuristics in 3.3. We don't bring the overview here, since some heuristics use the information saved by enhancing techniques brought in 3.2.

3.1.1.3.2 Choosing the Implication Order of the 3 DPLL Rules

Generally, application of pure-literal rule as well as of the unit-clause rule should be preferred over application of the splitting rule. This because the first 2 rules restrict variable to one and only one value, while the splitting rule give to variable 2 different values. Therefore, if we use the first 2 rules, we don't need to search the assignments' space with the second variable's possible value. Of course, the pure-literal and the unit-clause rules don't hurt the completeness of the algorithm, otherwise DPLL hadn't been complete.
In practice, there is no specific engine for detecting pure literals in state of art SAT solvers, since
1) Time waste resulting from running such an engine is more than time gain resulting from using pure-literal rule.
2) Good heuristics may choose the pure literal as a literal for the splitting rule.

In contrast, the unit clause rule role is extremely important in building an efficient SAT solver. In fact all modern solvers are using the unit clause rule as many times as possible between splitting rule usage. The whole process of identifying unit clauses and applying the unit clause rule is referred to as *Boolean Constraint Propagation (BCP)[17]*.
BCP can also be seen as identification of necessary assignments for variables, in process of searching for a model for a CNF instance. For example, if

$S = (A) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg C \vee D \vee E)$, then by applying the unit clause rule, we get a set of assignments {A, B, ¬C }.
Modern SAT solvers spend most of their time in the BCP process ([12]), thus its implementation and data structures used for it are of major importance. We'll talk about it extensively in 3.4.

3.1.1.4 Definitions and Notations Induced by DPLL Algorithm

In this sections we'll bring DPLL related notations and definitions.
We'll say that a variable/literal A, that was assigned a value is a

- *1st side variable/literal,* if it was assigned a value as a result of splitting rule application and DPLL (S|β) is performed.
- *2nd side variable/literal,* if it was assigned a value as a result of splitting rule application and DPLL (S|γ) is performed.
- *decision variable/literal,* if it's a 1st side or 2nd side variable/literal
- *implied variable/literal*, if it was assigned a value as a result of unit clause rule application.
- *non-implied variable/literal*, if it's not an implied variable/literal.

Another term we introduce is *decision level*. Each variable assigned a value during the DPLL algorithm has an associated decision level, which is the number of decision variables just after assigning the variable. We'll also refer to decision level of a literal meaning the decision level of its variable.
Another notion is that *"decide A"*, means assign A the value T and "*decide ¬A*", means assign A the value F.
We'll say that clause ω is an *implication clause* of literal A, if A is an implied literal, s.t. ω is the unit-clause which made using the unit-clause rule possible for A.
Another term is *decision stack*. Throughout the run of DPLL, all the assigned variables are held on a stack. We push to the stack each newly assigned variable and pop from the stack each newly unassigned variable. Observe, that in case of recursive DPLL implementation, the stack of recursion serves as a decision stack.

We'll also introduce a *decision tree* – a tree data structure that enables us to conveniently present the system at each stage of DPLL algorithm. We define it slightly differently than it's usually defined.

Each vertex of the decision tree corresponds to a splitting-rule application. Edges that are below a vertex and are connected to it correspond to decision variable A and all the implied literals which were assigned as a result of A's assignment (the implied literals are written in brackets). The tree depth of an edge is the decision level of variables associated with it. Each branch corresponds to a conflict or to a satisfying assignment. Pre-order pass on the tree gives us all the splitting-rule applications in the order they have been made. Below is an example of a decision tree resulting from running DPLL algorithm on a simple unsatisfiable CNF formula defined on variables {A,B,C,D,E} which is a conjunction of the following clauses:

$\omega_1 = \neg A \vee D$; $\omega_2 = \neg C \vee \neg D \vee E$; $\omega_3 = \neg C \vee \neg D \vee \neg E$; $\omega_4 = C \vee E$; $\omega_5 = C \vee \neg E$; $\omega_6 = A \vee \neg C \vee D$.

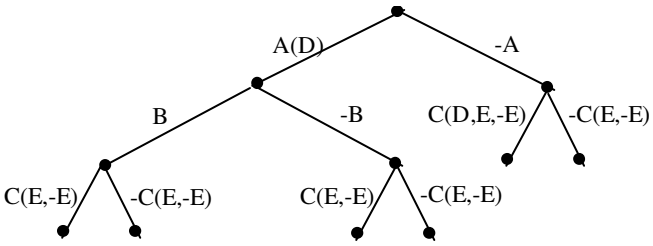


Fig 3.1.1.4

3.1.1.5 Iterative DPLL Algorithm

After the discussion of previous subsection, we can introduce an iterative implementation the DPLL algorithm. It will be useful for subsequent sections discussion.

More detailed iterative DPLL implementation may be found in Appendix A.

DPLL:

1. While (TRUE)
    1.1. If last decision variable A of last conflict was 1st side variable
        1.1.1. Assign A to the 2nd side and run BCP
    1.2. Otherwise, choose the next 1st side variable, assign it and run BCP
    1.3. If there is a contradiction
        1.3.1. While there are assigned variables (and the loop isn't broken)
            1.3.1.1. If the last decision variable A is 1st side variable
                1.3.1.1.1. Unassign it together with variables implied as a result of BCP after A's assignment.
                1.3.1.1.2. Mark that there is a variable to be assigned
                1.3.1.1.3. Break the loop 1.3.1
            1.3.1.2. If the last decision variable A is 2nd side variable
                1.3.1.2.1. Unassign it together with variables implied as a result of BCP after A's assignment.
                1.3.1.2.2. Continue the loop 1.3.1
    1.4. If all the variables are assigned
        1.4.1. Return SAT
    1.5. If there are no variables assigned and there is no variable marked to be assigned
        1.5.1. Return UNSAT

3.2 Enhancing DPLL Algorithm

In recent years, many ways of improving the basic DPLL algorithm were suggested. We'll overview it in this section.

3.2.1 Conflict Analysis-Driven Learning

The most powerful set of improvements to the basic DPLL algorithm is based on analysing the reasons bringing to each conflict and learning from it. In this section we'll

overview different possibilities of learning from conflict analyses. In subsection 3.2.1.1, we'll introduce the concept of implication graph and related concepts, which will be very helpful in explaining different learning schemes in subsequent subsections.

3.2.1.1 Implication Graph

Implication graph is a directed acyclic graph. Each vertex contains a literal corresponding to an assignment of a value to appropriate variable. Each literal has a decision level associated with it.
Each assigned variable A has a vertex associated with one of its literals corresponding to its assignment sign. In what follows "vertex A" means "vertex corresponding to literal A in the implication graph". We'll sometimes say "A" meaning "vertex A".
There is an edge from a vertex A to vertex B iff

- ω is an implication clause of B
- $\omega(\neg A) = T$ ($\neg A$ appears in ω)

Observe, that during the run of DPLL algorithm each variable A is either:

- Not assigned
- Decision variable. In this case it has no implication clause.
- Implied variable. In this case implication clause of A is a clause where A was implied.

By saving the implication clauses at the moment of assignment (in appendix A pseudo-code it's saved in ImplicationClause) we implicitly keep an implication graph at each stage of the algorithm.

The reason for each contradiction is the fact, that there is a variable A (referred to as *conflicting variable*) that has to be assigned both T and F as a result of previous assignments. Therefore A will be the only variable, which will have a vertex for both of its literals in the implication graph after a contradiction.

In what follows, **when referring to the implication graph, we refer only to the parts of implication graph, which are connected to the conflicting variable vertices (unless explicitly stated otherwise).**

Below on the left is an example of an implication graph for a particular set of assignments for a conjunction of clauses given. A branch of decision tree corresponding to it is presented on the right.

$\omega_1 = \neg H \vee D \vee G \vee \neg B$; $\omega_2 = A \vee B$; $\omega_3 = \neg B \vee C \vee D \vee \neg H$;
$\omega_4 = \neg B \vee \neg E \vee K$; $\omega_5 = D \vee \neg C \vee \neg G \vee E$; $\omega_6 = \neg E \vee \neg K$

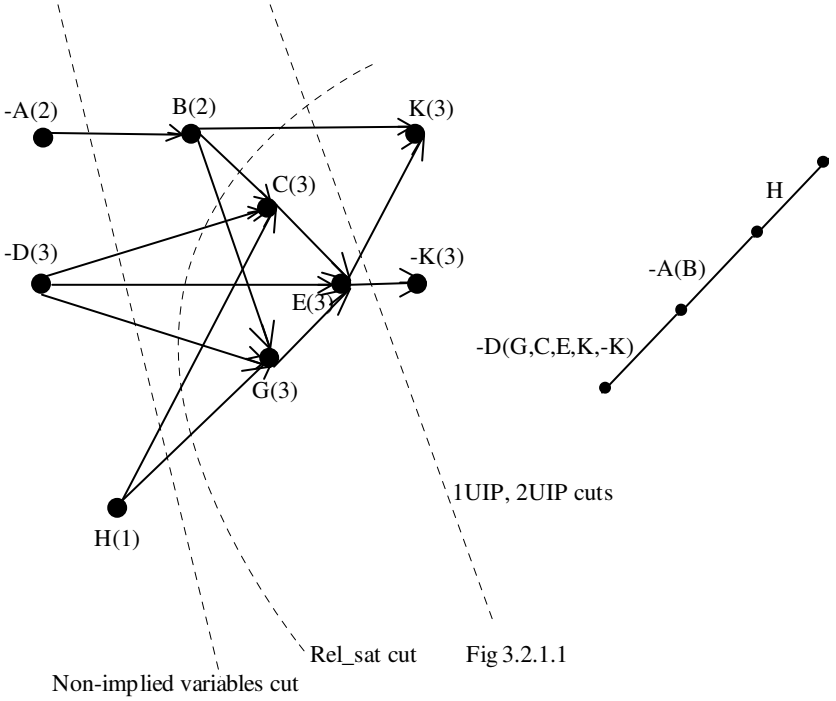


Fig 3.2.1.1

In an implication graph, vertex A is said to dominate vertex B iff any path from the vertex of decision variable of the decision level of A to B needs to go through A.
A *Unique Implication Point(UIP)* ([10]) is a vertex of a literal of last decision level that dominates both vertices of the conflicting variable. A decision variable vertex is always a UIP. There may be more than one UIP for a conflict. UIP's can be ordered starting from the conflict. In our example there are 2 UIP's: E is the first UIP and D is the second UIP.

By analysing the implication graph after a contradiction, we can deduce sets of assignments that are enough to produce a conflict.

3.2.1.2 Clause Recording Schemes

Clause recording is a pruning technique that the main idea of it is after each contradiction to identify a set of assignments (which is also a set of literals) that is enough to produce the contradiction. Such set of literals is referred to as *conflict's reason*. Then a clause, which is referred to as a *conflict clause* that prevents such set of assignments to occur again is added to the initial formula. By adding such clauses we prune the search space. This idea with respect to the SAT problem was firstly introduced in [9] and then in [10]. It was further developed and generalised in [31].
It was known as no-good learning in the literature on Truth Maintenance Systems ([32],[33]) and Constraint Satisfaction([3].
A *conflict clause* is generated by a bipartition of the implication graph ([31]). All the decision variables are on one side of partition (called *reason side*). 2 vertices of the conflicting variable has to be on the other side (called *conflict side*). All vertices on the reason side that have at least one edge to the vertices of the conflict side comprise the reason of the conflict. The conflict clause is received by taking the disjunction of negations of literals corresponding to such vertices.

Different cuts correspond to different learning schemes.

We are ready now to describe different cuts and corresponding conflict recording schemes.

3.2.1.2.1 Rel_sat cut and conflict recording scheme [9]

Rel_sat is one of the first SAT solvers to incorporate learning. Rel_sat puts all variables assigned at last decision level, but the decision variable on the conflict side and all others variables on reason side. $\neg H \vee D \vee \neg B$ is a conflict that would be recorded according to this scheme in our example.

3.2.1.2.2 GRASP conflict recording scheme [10]

GRASP(a SAT solver name) tries to learn as much as possible from a conflict. It records a few new clauses after a contradiction.

Before describing it, it's worthy to note that GRASP algorithm is a little bit different from one brought in 3.1.1.5.

GRASP considers what we called the 1st side variables to be "real decision variables". If there is a conflict straight after assigning 1st side variable GRASP may try to check the second side of an implied variable, which is the first UIP. Such a decision is called "fake decision". The implementation of this idea is conflict-recording based. GRASP adds a clause corresponding to a partition where all variables assigned after the first UIP are on the conflict side. In example 3.2.1.1 this corresponds to ¬B∨¬E. Observe that after backtracking ¬E has to be implied, as the new clause becomes a unit clause.

If there is a conflict straight after assigning a real decision variable, GRASP adds another clause, which is a *UIP reconvergence* at the current level. Such clause is produced by taking all the literals, but the UIP literal from the clause where the UIP literal was implied and replacing each such literal B by literals from clause where B was implied. In our example it means adding $\neg B \vee D \vee \neg H \vee \neg E$ for a UIP literal E. This kind of learning is used only if there exists some reconvergence in the part of graph where the learning takes place. Otherwise, no new information can be added by such learning. Observe also, that this kind of learning may take place even if there is no contradiction right now.

In addition, if "fake decision" has been made and there is a conflict straight after it, GRASP adds a clause corresponding to a cut, where all the vertices at the current decision level (including the fake decision variable) are on the conflict side. Observe that the fake decision variable is always associated with a clause where it was implied, therefore it cannot be considered a real decision variable, thus such cut doesn't contradict the definition of valid bipartition brought above.

3.2.1.2.3 Non-implied variables cut and conflict recording scheme([31])

This is a simple scheme, which adds only the non-implied variables connected to the conflicting variable vertices, to the reason side ($A \vee D \vee \neg H$ in example). The original name of the cut is Decision cut. In fact, in the DPLL algorithm, the set of decision variables is identical to the set of non-implied variables, therefore those 2 names have the same meaning. But as we'll see there are other non-implied variables in the new algorithm CRSAT and it's more convenient for us to call this cut "non-implied variables cut".
In Appendix A.2 we bring pseudo-code for identifying non-implied variables connected to the conflicting variable vertices after a conflict.

3.2.1.2.4 1UIP, 2UIP,…,AllUIP cuts and conflict recording schemes([31])

The 1UIP scheme adds one clause after each conflict. It requires that the variables assigned after the first UIP will be on the conflict side and all others will be on the reason side. Observe that this clause corresponds to one of the clauses added by GRASP (described first in section 3.2.1.2.2).

To describe the 2UIP,3UIP…AllUIP schemes we need to introduce the idea of UIP generalisation([31]). The concept of UIP may be generalised for decision levels other than the decision level at the moment of detecting a contradiction. Vertex A at decision level d is a UIP iff any path from the decision variable of d to the conflicting variable needs to go through A or a literal on the reason side of a decision level higher than A. Note that decision variables are always UIP's. In what follows UIP's of each level will be ordered starting from the conflict.

The *i*UIP scheme requires that the first UIP's of the last *i* decision levels will be on the reason side just before partition. The AllUIP scheme (example on fig. 4.7.2.b) requires that the first UIP's of all decision levels will be on the reason side just before partition.

The algorithm for finding *i*UIP cut is as follows: the solver needs to find the first UIP of the current decision level, then all variables of current decision level assigned after it that have a path to the conflicting variable will be on the conflict side; the rest will be on the reason side. Then the solver will proceed to the previous decision level variables and so on.

3.2.1.2.5 MinCut cut and conflict recording scheme([31])

This learning scheme tries to make the conflicts as small as possible. This corresponds to a problem that given an implication graph, remove the smallest number of variables so that there would existed no path from the decision variable to the conflicting variable. This is a typical min-cut problem.

3.2.1.2.6 Conclusions

In previous sections we described existing clause recording schemes. In [31] there is an extended review of it as well as some practical results comparing it. 1UIP seems to perform best on different benchmarks.
Other schemes as well as combinations of existing ones may also be considered as a possibility when designing a new SAT solver.
Another thing worth to mention is that clause recording schemes are different not only in clauses produced, but also in time it requires to produce the recorded clauses according to each scheme.
All the described schemes, but the MinCut schemes are of O(V+E) complexity, but the time required to run each one of it in practice may differ.
This also should be considered when choosing an appropriate scheme for a SAT solver.

3.2.1.3 Relevance-Based Learning

It's not practical to keep all the learned clauses both because of space restrictions and the increasing complexity of the BCP process, therefore some of them must be deleted. Different relevance-based learning techniques allow us to delete not relevant clauses. The difference between them is what clauses should be deleted and when should they be deleted.

The first relevance-based learning technique has been proposed in [9]. According to it the clauses are deleted as soon as the number of unassigned literals becomes greater than some threshold $k \geq 1$.

A similar idea, so called *k-bounded learning,* is proposed in [10]. According to it, clauses with a size less than a threshold k are kept during the subsequent search, whereas larger clauses are discarded as soon as the number of unassigned literals is greater than one.

A generalisation of the last idea is proposed in [34]. According to it, the search algorithm is organised so that all recorded clauses of size no greater than k are kept and larger clauses are deleted only after m literals have become unassigned.

Finally, it's proposed in [19] to choose clauses to remove not only according to their size, but also according to their "activity", i.e. the number of times its literals participated in conflicts and "age", i.e. when was it recorded.

We'll propose another idea for relevance-based learning in 4.8.

3.2.1.4 Non-Chronological Backtracking

Basic DPLL algorithm as well as many SAT solvers based on it is always backtracking to the lastly assigned 1st side decision variable.

However it's possible in some cases to backtrack to earlier decision levels, therefore skipping 2nd side checking of some decision variables. This idea was first proposed in [33]. It is incorporated in many modern SAT solvers([9],[10],[12] and others).
The existing implementation of this idea for SAT solvers is conflict analysis based.
We'll explain it on an example.
Suppose that there is a conflict just after the BCP procedure after assignment of 1st side decision variable D of 3d decision level (see implication graph on fig. 3.2.1.4.a). Observe that the clause corresponding to the 1st UIP cut is $\omega_1 = \neg B \vee \neg E$. It does not include any variables of decision level 2.
After unassigning E, ω_1 becomes unit, therefore ¬E should be implied immediately. This corresponds to the GRASP conflict recording scheme described in 3.2.1.2.2 and isn't implemented in basic DPLL algorithm brought in 3.1.1.5.
Suppose, that after deciding ¬E, there is a conflict and the situation is one displayed on 3.2.1.4.b. The clause corresponding to 1st UIP cut is $\omega_2 = \neg B \vee E$. Observe that ω_2 as well as ω_1 does not include variables of 2nd decision level.
Observe, that $\omega_1 V \omega_2$ is a contradiction if B is assigned to F. The values of the variables of 2nd decision level don't influence this fact. To unassign B we have to non-chronologically backtrack to decision level 1 and we are skipping the 2nd decision level!

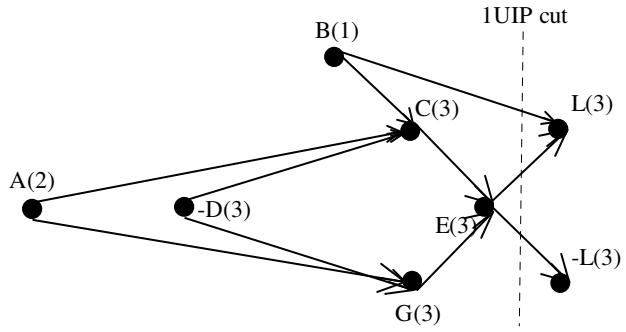


Figure 3.2.1.4.a

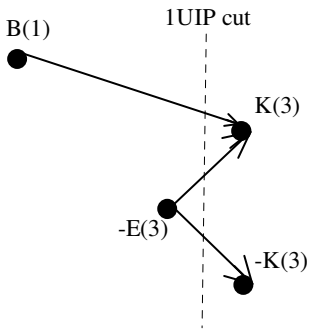


Figure 3.2.1.4.b

3.2.2 Search Restarts

By restarting the search we mean stopping the backtrack search process at some point, unassigning all the variables and starting the search again, using different assignment order. Search restarts have been proposed and shown effective for real-world SAT instances in [21]. In the above work, restarting is applied after constant number of steps. It may help avoiding staying a long time in dead-end branch. However, it harms the completeness of the algorithm. To achieve completeness using restarts the following ideas were introduced:

1. Recording conflict clauses and keeping all the recorded clauses in the database([22])
2. After each restart increase the number of steps after which next restart is applied([23])
3. Recording conflict clauses, keeping only part of it in the database, but increasing the number of clauses recorded after each restart([12])
4. Keeping the search signature([24]) (explained below).

The first solution is not practical. The 2^{nd} and 3^{d} solutions have the drawback, that paths in the search tree may be visited more than once after each restart.

The 4^{th} solution tries to cope with this problem. It induces a relevance-based learning technique, optimised for restarts. Suppose, that the decision variables on the stack are A_1(lowest decision level)…A_m and we are just after backtracking. Suppose we want to restart at this point. Suppose that $A_{i(1)}...A_{i(k)}$ are the variables from $A_1…A_m$ that participated in some conflicts after last restart. Observe, that the conflicts which contain one of $A_{i(1)}...A_{i(k)}$ are enough to define the search tree that was explored. Therefore, we can delete the other conflict clauses and still ensure that no path will be repeated after the restart.

The drawback of this idea is that the length, the age and the activity of the clauses deleted are not taken into consideration. Therefore all the relevance-based techniques described in 3.2.1.3 cannot be used. In particular, even very short clauses, which exclude from further search big parts of search space, may be deleted.

In 4.6.2 we propose a solution, which enables us to prevent paths repeat and allows us to use all the existing relevance-based learning techniques.

3.2.3 Exploiting the Structure of SAT Instances

These techniques exploit the structure of CNF formula to simplify the search. We'll briefly overview them basing on the overview in [34].

3.2.3.1 Clause Subsumption

Consider the clauses $\omega = A \vee B \vee C \vee \neg D$; $\gamma = A \vee B$. Observe that if γ is T under some interpretation than ω is also T. We say, that γ structurally subsumes ω. During the search some clauses may become dynamically subsumed by others. For example, if $\omega = G \vee A \vee B \vee C \vee \neg D$; $\gamma = K \vee A \vee B$ and G and K are assigned F during the search, than γ subsumes ω.

Observe that if some clause ω is subsumed by some clause γ, than ω may be excluded from the database, since it doesn't influence the value of inspected function. If the subsumption is dynamic ω should be returned to the database after unassignment of variables responsible for appearance of subsumption.

There exist a few ideas, when can the subsumption take place during the run of backtrack search algorithm:

1. *Apply structural subsumption between each pair of clauses prior to the search.*
   This approach produces very little number of subsumed clauses for most of the benchmarks([35]).
2. *After each BCP after decision assignment exclude the new dynamically subsumed clauses and include it again after appropriate unassignments.*
   This approach is impractical since it requires significant overhead ([34]).
3. *Each time a new clause is recorded apply structural subsumption between it and existing clauses.*
   This approach also requires significant overhead and is hardly justifiable ([35]).

3.2.3.2 Formula Partitioning

Let $X=\{x_1 \ldots x_n\}$ and $Y=\{y_1 \ldots y_m\}$ be disjoint sets of variables. Let ω be a CNF formula defined on X and γ be a CNF formula defined on Y.

Let α be a CNF formula, s.t. $\alpha = \omega \Lambda \gamma$. If we want to prove that there is no model for α, it's enough to prove that there is no model for ω or γ, i.e. run DPLL algorithm on ω and γ separately. If we found a model for ω and a model for γ, it means that α has a model, which is a union of 2 models, found.

Therefore if some formula α may be partitioned to non-trivial set of 2 or more CNF formulas defined on disjoint sets of variables, we may run DPLL algorithm on smaller formulas. This situation may occur dynamically during the backtrack search.

Obviously, for identifying such situations we need to run some procedure. There exist a few ideas, when can such procedure take place during the run of backtrack search algorithm:

1. *Prior to the search*

   This is not expected to be relevant on practical benchmarks([34]).
2. *After each BCP following decision assignment*

   This approach is impractical since it requires significant overhead([34]).
3. *The procedure is implicitly run when decision assignments are restricted to variables in a partition.*

   This approach can simplify the search as suggested in [35].

3.3 Branching Heuristics

In this section, we'll describe existing heuristics for choosing the literal for 1st side assignment.

A "good" branching heuristics is one which enables in a branch containing an assignment to quickly find it and in a branch not containing an assignment to quickly find contradictions. Another important demand to a branching heuristics is that it could be calculated with as less overhead as possible.

In 3.3.1 we'll overview branching heuristics, which do not use the information gathered during new conflicts analysis. In 3.3.2 we'll overview more recent branching heuristics, based on conflict analysis. Modern SAT solvers use such heuristics (Chaff[12] was the first one to use it), since there is very little overhead using it and it helps to achieve significant gains in performance ([12],[19]).

In 4.4 we'll propose a new conflict-analysis based heuristics.

3.3.1 Conflict Analyses Independent Branching Heuristics

Before observing the heuristics itself, it's worth to mention an interesting idea proposed in [25], although it is not (yet?) implemented in modern SAT solvers. The idea is not to apply one branching rule during the whole search process, but to give each branching rule the possibility to make a decision assignment from time to time. For details of different ways to select a rule at every point please refer to [25].

It's also worth to mention that in [18] there is a review of conflict analyses independent heuristics together with experimental results comparing it.

3.3.1.1 Bohm([26])

The Bohm branching rule selects the variable A with maximal vector $(H_1(A),H_2(A),\ldots,H_n(A))$ in lexicographical order, where $H_i(A)=\alpha \cdot \max(h_i(A),h_i(\neg A))+\beta \cdot \min(h_i(A),h_i(\neg A))$, where $h_i(A)$ is the number of unresolved clauses with i literals. Selected literal gives preference to satisfying small clauses (when assigned true) or further reducing the size of small clauses (when assigned false). α and β are chosen heuristically. Values $\alpha=1$ and $\beta=2$ are suggested in [26].

3.3.1.2 MOM – Maximum Occurrence on Clauses of Minimal Size([27])

The MOM branching rule selects a variable A that maximises $(f^*(A)+f^*(\neg A))\cdot 2^k + f^*(A) \cdot f^*(\neg A)$, where $f^*(l)$ is the number of occurrences of literal l in the smallest unresolved clause. As its name suggests, the rule gives preference to variables that occur frequently as positive as well as negative literal in many clauses The value of k can vary. For example it's used in Satz([28]) with k=10.

3.3.1.3 Jeroslaw-Wang Heuristics([29])

For a given literal A, compute:

$$J(A) = \sum_{\psi \text{ s.t. } \psi(A)=T} 2^{-(\text{length of } \psi)}$$

The *one-sided Jeroslow-Wang (JW-OS)* heuristics selects the literal with largest J(A).
The *two-sided Jeroslaw-Wang (JW-OS)* heuristics identifies variable with largest J(A)+J(¬A) and assigns A the value T if $J(A) \geq J(\neg A)$ and the value F otherwise.

3.3.1.4 Literal Count Heuristics([18])

Let C_p/C_n be the number of unresolved clauses in which a given variable A appears as a positive/negative literal.
DLCS(Dynamic Largest Combined Sum) heuristics selects the variable with maximal C_p+C_n. A literal of A to be selected is positive iff $C_p \geq C_n$.
DLIS(Dynamic Largest Individual Sum) heuristics considers literals, not variables. It selects the literal with maximal C_p(for positive literals) or C_n(for negative literals).
RDLIS(Random Dynamic Largest Individual Sum) heuristics selects the variable with maximal C_p+C_n. A literal of A to be selected is chosen randomly.

3.3.2 Conflict Analyses Based Branching Heuristics

In this section we'll overview conflict analyses based branching heuristics.

3.3.2.1 VSIDS (Variable State Independent Decaying Sum Decision) Heuristics([12])

This was the first heuristics based on conflict analyses that was proposed. It's defined as follows:

1) Each literal has a counter, initialised to 0.
2) When a clause is added to the database , the counter associated with each literal in the clause is incremented

3) The literal with highest counter value is chosen
4) Ties are broken randomly, although it's configurable
5) Periodically all the counters are divided by a constant (2 for Chaff[12]).

This strategy can be viewed as attempting to satisfy the recent conflict clauses. It's dynamic, since it gives preference to information received recently and therefore adjusts itself quickly to the changes in database. It also has extremely low overhead, since the statistics can be updated during conflict analyses.
Chaff uses STL set(which is implemented by Red-Black Tree) to maintain the variables sorted by counter value. After each update of the scores, the vector is sorted using *stable_sort*, which implies a lower bound of $O(n \cdot \log n)$ and an upper bound of $O(n \cdot (\log n)^2)$ for n variables.

3.3.2.2 VOX (Variable Ordering Extension) Heuristics([30])

This heuristics is a variation on VSIDS. The fact utilized by the author of VOX is that after each conflict recording only the variables participating in the conflict clause should be rearranged in the ordering. Let the number of such variables be k. Using STL Multiset it's possible to remove all those variables, sorting it and entering it into back the main vector. The Deletion and Insertion operations for the STL Multiset are bound by O(log n). It's also needed to divide all the variables by a constant. Therefore the overall runtime is $O(n + k(\log k)^2 + k \cdot \log(n))$. Unfortunately, this heuristics doesn't give advantages over VSIDS in experiments([30]).

3.3.2.3 BerkMin Heuristics ([19])

In a very recent article [19] a new heuristics is proposed. It's complex, but is justified in experiments ([19]).
BerkMin (SAT Solver using the new heuristics) holds the clauses in a chronologically ordered stack (the top clause is the one deduced the last). A new branching variable is

always chosen among the free variables of the upper-most clause, which makes the choice even more "dynamic" that VSIDS's. The variable selected is chosen as follows: BerkMin holds a counter for each literal. This counter role and management is similar to one of VSIDS. The important difference is that BerkMin increases the counter not only for the literals in a new conflict clause, but for all literals involved in conflict making. The value of a constant, the variables are periodically divided by, is 4. The variable selected is one with largest value of the counter.
A rather complex procedure is used for determining the literal to be chosen first. We omit it here and refer the reader to [19].

3.4 Data Structures Representing SAT Formula

Data structure enabling to implement BCP and backtracking efficiently is crucial for a SAT solver, since those operations are executed millions of times.
In order to efficiently implement the BCP procedure, a data structure should allow us to find fast unsatisfied and unit clauses after each variable's assignment. In order to efficiently implement the backtracking, data structure should allow us to make as less as possible operations to maintain the data structures consistent after backtracking.

Most backtrack search SAT algorithms represent clauses as lists of literals and associate with each variable 2 lists of clauses – one for each literal.

One obvious idea is to hold for each literal A references to **all** clauses containing A. Such data structures are called *adjacency lists*. However such data structures do not reflect the fact, that most of references to each clause need not be analysed when each its variable is assigned. *Lazy* data structures keep a reduced set of clauses for each literal and are much more efficient than adjacency list data structures ([20]).
Below we review the lazy data structures. For a review of adjacency lists data structures (as well as an alternative review of the lazy ones), one can refer to [20].
In 4.5 we'll propose a new efficient lazy data structure.

3.4.1.1 Sato's Head/Tail Lists([11])

The first lazy data structure proposed was the Head/Tail (H/T) data structure, originally used in the SATO solver ([11]).

Each clause is implemented as an array. The original order of literals in the array is arbitrary. Each array cell (or clause cell) contains literal number and pointer to previously visited literal. Initially all these pointers are set to NULL.
Each clause will initially appear on 2 literals' lists (say, literals A and B). More specifically, references to 2 different clause cells containing A and B will be on the lists of A and B. We will call those referenced clause cells: head and tail. Initially, the first array cell is the head and the last array cell is the tail.
At each point of the algorithm's execution, each clause can have many references to it, but there will always be head and tail clause cells.

Each time a value is assigned to a variable or a variable becomes unassigned, the algorithm traverses the list of clauses of corresponding literal. The algorithm is built in such way, that all the clause cells appearing on such list are head/tail.

On each visit to a clause for an assignment operation(say A is assigned T), the algorithm traverses the cells of that clause one by one from the head/tail towards the tail/head. If it finds a satisfied literal, the clause is declared satisfied and the algorithm passes to the next clause. If it finds an unassigned literal B, it:

- adds the clause cell of B to the list of B and this cell becomes the new head/tail instead of ¬A
- makes the pointer to previously visited literal of the cell of B to point to the cell of ¬A

If it doesn't find not satisfied nor unassigned literal before the other reference is reached, then the clause is declared unit, unsatisfied or satisfied according to the status of the tail/head.

On each visit to a clause for an unassignment operation(say B halts to be F), the algorithm checks the pointer to the previously assigned literal of B cell. If it's NULL, it makes nothing. If it points to other clause cell(say containing literal ¬A), the algorithm removes the clause cell of B from the list of B. As a result, ¬A becomes the new head/tail and B halts to be the head/tail.

3.4.1.2 Chaff's Watched Literals(WL)([12])

In [12], another lazy data structure is proposed. The differences from H/T are that

- each clause has 2 and only 2 references to it
- there is no difference between the 2 references (each one is called "watched").

There is also no need in pointers to the previous literals in each cell.
On each visit to a clause for an assignment operation(say A is assigned T), the algorithm traverses the cells of it, one by one from the first cell towards the last cell. If it finds a satisfied literal, the clause is declared satisfied. If it finds an unassigned literal B it:

- adds the cell of B to the list of B (the cell of B becomes watched)
- removes the cell of ¬A from the list of ¬A (the cell of ¬A halts to be watched)

If it doesn't find not satisfied nor unassigned literal, then the clause is declared unit, unsatisfied or satisfied according to the status of the other watched literal.

Using this data structure there is no need in visiting clauses during the unassignment operation. This is a clear advantage of WL over H/T. Another advantage is that there are only 2 references to every clause, while H/T might require number of references comparable to number of literals in the clause.
The drawback is that on each visit to a clause all the literals may be visited.

3.4.1.3 Head/Tail Lists with Literal Sifting (htLS) ([20])

This data structure is proposed in [20]. It is based upon H/T data structure. The main idea is that the clause cells between the first cell and the head cell are kept sorted according to

non-increasing decision level. The cell between the last cell and the tail cell are kept sorted as well.
This sorting is achieved by sifting assigned literals at each visit to a clause for an assignment operation.
In the paper, were htLS is proposed, it's stated, that it's enough to hold 4 literal references to each clause. We believe that 2 literal references (to H and T) are enough.

As in the case of H/T data structure, each time a value is assigned to a variable or a variable becomes unassigned, the algorithm traverses the list of clauses of one of its literals.
On each visit to a clause for an assignment operation (say A is assigned T), the algorithm is traversing the clause cells, one by one from the head/tail towards the tail/head. If it finds a satisfied literal, the clause is declared satisfied. If it finds an unassigned literal B, it:

- adds the clause cell of B to the list of B and this cell becomes the new head/tail instead of ¬A
- sifts ¬A cell towards beginning/ end of the clause, till a cell containing literal with lesser or equal decision level than A.

In addition, if the algorithm finds an unsatisfied literal B, it sifts it, to keep appropriate part of clause sorted.
If it doesn't find not satisfied nor unassigned literal before the other reference is reached, then the clause is declared unit, unsatisfied or satisfied according to the status of the tail/head.

On each visit to a clause for an unassignment operation (say B halts to be F), the algorithm:

- removes the cell of B from the list of B (the cell of B halts to be watched)
- if the cell of B isn't the first/last (for head/tail respectively), insert the adjacent(towards the beginning/end) cell (say, containing A) to the list of A. Therefore the cell containing A becomes the new head/tail.

The major advantage of this data structure over H/T and even more over WL is that all the literals which are assigned at low decision levels are "sifted out" towards the edges of clauses and therefore are not visited till they become unassigned. Therefore, at each visit to a clause we tend to visit fewer literals.
Another advantage of this data structure over H/T is that there are only 2 (or 4 in the original article) references to each clause. WL has 2 as well.
A disadvantage of this data structure comparing to WL is the need to visit clauses during unassignment.
Another disadvantage is the fact that literal sifting takes additional time.

3.4.1.4 Watched Literals with Literal Sifting (WLS) ([20])

This data structure is proposed in [20]. As the name suggests, it combines WL data structure with literal sifting. The major difference between this data structure and htLS should be the absence of need to visit clauses during unassignment. As in the case of htLS data structure, the clause cells from the watched cells towards the edges of the clause are kept sorted. There is a need here in 2 additional references to the clause (we'll denote it by HS and TS): each such reference points to a cell, clause is sorted from.
On each visit to a clause for an assignment operation (say A is assigned T), the algorithm is traversing the clause cells, one by one between the watched cells. If it finds a satisfied literal, the clause is declared satisfied. If it finds an unassigned literal B, it:

- adds the cell of B to the list of B (the cell of B becomes watched)
- removes the cell of ¬A from the list of ¬A (the cell of ¬A halts to be watched)
- sifts ¬A cell towards beginning/ end of the clause, till a cell containing literal with lesser or equal decision level than A.

In addition, if the algorithm finds an unsatisfied literal B, it sifts it, if needed.
If it doesn't find not satisfied nor unassigned literal before the other watched cell is reached, then the clause is declared unit, unsatisfied or satisfied according to the status of the other watched cell.

Unfortunately, there is a need to visit clauses for an unassignment operation. The cells that should be visited are HS and TS, not the watched cells. On each visit, adjacent cells

(adjacent towards the edges) become the new HS/TS instead of the old HS/TS. This operation is necessary, since otherwise on a visit to a watched cell for an assignment operation we couldn't know, whether the cells from HS/TS towards the edges of the clause are sorted. Indeed, the literals in those cells could have been unassigned and reassigned with different decision levels.

An advantage of this data structure over ones described earlier is that we tend to visit fewer literals than htLS at each visit to a clause. The advantage over H/T is that we don't need to visit clauses for unassignment of 2 watched literals.
However, there are a few disadvantages to this data structure. A clear disadvantage comparing to WL is that clauses (HS and TS references) should be visited during unassignment. Another disadvantage is that literal sifting requires additional time.

3.5 Pre-processing Techniques

Pre-processing techniques are methods that can be implied on a CNF formula prior to the search. Pre-processing algorithms can dramatically improve the performance of SAT solvers ([37]). Such algorithms simplify the formula by various means including symmetry breaking, deduction of necessary assignments, addition of implied clauses, deletion of redundant clauses and identification of equivalent literals. A number of pre-processors have been developed : SymFind([38]), Compact([39]), Compactor([40]), Simplify([41]), CompressLite([42]).
We won't give an extended overview of pre-processing techniques since they are not part of the DPLL algorithm nor an extension to it, but rather a complementary method. An overview of existing pre-processing techniques together with references to other articles on the topic can be found in [36].

4 New Algorithms and Techniques

In this section, we'll introduce ideas developed in this work.
In subsection 4.2, a new complete SAT algorithm CRSAT generalising DPLL is provided together with a sketch of correctness proof. Fool correctness proof is provided in Appendix A.4.
In other subsections innovations in many aspects of backtrack search algorithms structure are proposed. This includes improvements to non-chronological backtracking, new pruning methods, new heuristics, new relevance-based learning technique, new effective data structure, a new idea of allowing not repeat paths after restarts and an idea of dynamically changing(and minimising the number of) 1^{st} side variables.
All these innovations, but those proposed in 4.3.2 and 4.3.3 are applicable for both DPLL and CRSAT algorithms. The new pruning techniques from 4.3.2 and 4.3.3 can be used only by CRSAT algorithm.

4.1 Decisions-Based Non-Chronological Backtracking

In this section we'll describe a new non-chronological backtracking technique, which is based not on implied variables analyses (like in [9],[10]), but on decision variables analyses.
The idea is very simple. For each 1^{st} side decision variable of decision level *k*, we'll keep a counter, which will count how many times the variable participated in conflict clauses recorded according to the non-implied variables scheme. Before beginning the 2^{nd} side checking of the variable, we'll check the value of its counter. If its value is 0, no 2^{nd} side checking is needed. The reason for it is that after we unassign variables of decision level *k*, the conflict will not disappear, since it's independent of their values.
This idea is incorporated into the CRSAT algorithm introduced in next section. In fact CRSAT generalises this idea. It tries to not only skip the 2^{nd} side checking when the

counter is 0, but also minimise the checking when the counter has a small value. More information about it is provided in the next section.

4.2 CRSAT Algorithm

4.2.1 Intuition Behind the Algorithm

Let us suppose that we are running the DPLL algorithm on some CNF formula. Suppose, that decision variables assigned are $A_1 \ldots A_m$(low to high decision level) and B(highest decision level) and we are just before checking the 2nd side of B(checking ¬B) . Suppose that there were *k* conflicts when we were checking the 1st side of B.
Suppose we are keeping all the conflict clauses corresponding to the non-implied variables scheme (3.2.1.2.3). We don't add it to the formula, but keep it aside (we can add any consistent clauses to the formula, but it's irrelevant here).

In what follows we'll sometimes use the term "conflict" instead of "conflict clause".

If ¬B didn't participate in any of conflicts recorded, then according to the non-chronological backtracking principle, described in 4.1, we don't need to check its 2nd side at all.

Suppose now, that ¬B did participate in *p* conflicts. Intuitively, if we decide ¬B and make the same decisions we made while checking B, then in *k-p* paths we will meet the same (or shorter) conflicts as we met while we were checking B, since those conflicts appear independently of the value of B. Only in the *p* paths corresponding to the *p* conflicts containing ¬B, we won't meet the same conflicts (and could meet a model), since they are dependent of B value.
The immediate conclusion that, **it's enough to check the paths corresponding to conflicts where ¬B participated.** It's convenient to call to such a process "*conflicts checking"*.

It's worthy to note again, that the new principle generalises non-chronological backtracking principle from 4.1. Indeed, if there are no conflicts in which B participated – there are no paths to check.
The idea of conflict checking induces a new notation. We'll say, that a variable/literal is a *conflict variable/literal,* if it's a non-implied variable assigned a value in order to check a conflict.

Below is an example of a decision tree corresponding to DPLL run(fig. 4.2.1.a) and one corresponding to CRSAT run(fig 4.2.1.b) on the same formula which is also brought. Conflict variable and variables implied as a result of its assignment appear in *italic*. The corresponding edge appears in dots style. Conflict clauses corresponding to the non-implied variables scheme are brought below each leave on fig. 4.2.1.b.

$\omega_1 = \neg A \vee D;\ \omega_2 = \neg C \vee \neg D \vee E;\ \omega_3 = \neg C \vee \neg D \vee \neg E;\ \omega_4 = C \vee E;\ \omega_5 = C \vee \neg E;\ \omega_6 = A \vee \neg C \vee D.$

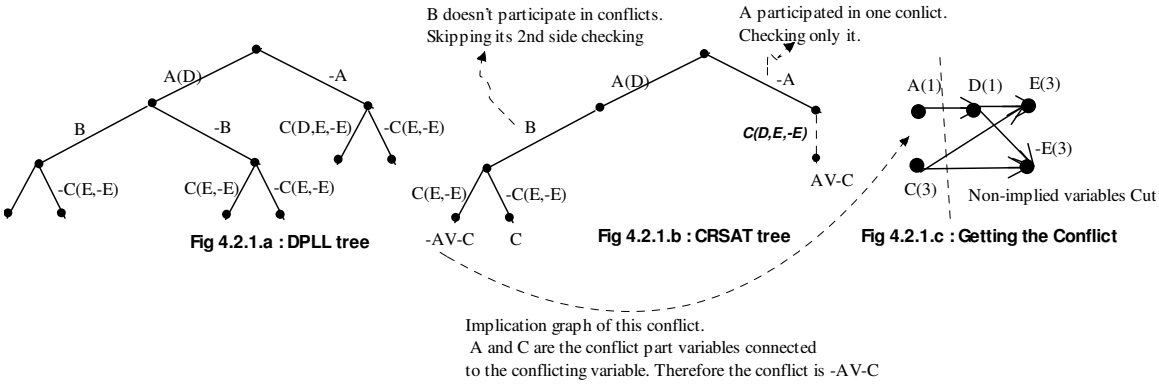


Fig 4.2.1.a : DPLL tree **Fig 4.2.1.b : CRSAT tree** **Fig 4.2.1.c : Getting the Conflict**

Fig 4.2.1.c illustrates the way conflict clause $\neg A \vee \neg C$ is received. The conflict clause variables correspond to the non-implied variables cut of implication graph (see 3.2.1.2.3 for more explanations and Appendix A.2 for detailed pseudo-code of how to receive the non-implied variables cut literals).

It's of course possible, that there will be a few conflicts to check for one decision variable. It's illustrated by an example below (the input CNF formula is irrelevant and is not brought).

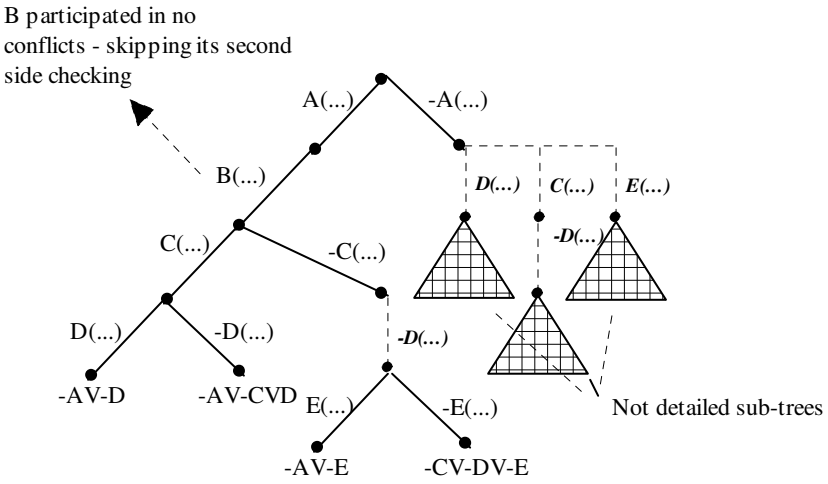


Fig. 4.2.1.d

We would like to add, that for every decision variable we can check its conflicts as it's proposed here or proceed with regular DPLL-style 2nd side checking.
In the next subsection, we'll introduce the CRSAT algorithm, which implements above principles.

4.2.2 Introducing CRSAT

The algorithm extends the DPLL algorithm brought in section 3.1.1.5.

In what follows, it will be convenient to use double-underline to stress that the double-underlined text does not appear in the DPLL algorithm and is unique to CRSAT. The text, which is wave-underlined, ensures that the non-chronological backtracking principle from 4.1 is implemented. Observe, that removing double-underlined and wave-underlined code from the algorithm below makes the code identical to the code of the DPLL algorithm brought in 3.1.1.5.

CRSAT:

1. While (TRUE)
   1.1. If last decision variable A of last conflict was 1st side variable
      1.1.1. Assign A to the 2nd side and run BCP
   1.2. If a conflict should be checked for last 2nd side variable
      1.2.1. Assign the conflict variables one by one and run BCP after each assignment till all the conflict variables are assigned or there is a contradiction.
   1.3. Otherwise, choose the next 1st side variable, assign it and run BCP
   1.4. If there is a contradiction
      1.4.1. While there are assigned variables (and the loop isn't broken)
         1.4.1.1. If the last decision variable A is 1st side variable
            1.4.1.1.1. Unassign it together with variables implied as a result of BCP after A's assignment.
            1.4.1.1.2. If there are no conflicts where it participated during current assignment
               1.4.1.1.2.1. Continue the loop 1.4.1
            1.4.1.1.3. Mark that there is a variable to be assigned
            1.4.1.1.4. If there are conflicts where it participated during current assignment
               1.4.1.1.4.1. Decide whether conflicts will be checked for this variable, associate appropriate value with the variable
            1.4.1.1.5. Break the loop 1.4.1
         1.4.1.2. If the last decision variable A is 2nd side variable
            1.4.1.2.1. If there is a conflict to check for it
               1.4.1.2.1.1. Unassign variables, assign as a result of previous conflict checking (if needed)
               1.4.1.2.1.2. Break the loop 1.4.1

            (We can be here only If we are not checking conflicts for A or we have already checked all the conflicts)

            1.4.1.2.2. Unassign it together with variables implied as a result of BCP after A's assignment.
            1.4.1.2.3. Continue the loop 1.4.1
   1.5. If all the variables are assigned
      1.5.1. Return SAT
   1.6. If there are no variables assigned and there is no variable marked to be assigned
      1.6.1. Return UNSAT

4.2.3 CRSAT as DPLL generalisation

Note that if we always choose at line 1.4.1.1.4.1 not to check conflicts, than the behaviour of CRSAT is identical to that of DPLL enhanced by non-chronological backtracking from 4.1.
To see this, observe that in this case double-underlined pseudo-code sections don't influence CRSAT behaviour. Indeed, the code executed in case that the condition of line 1.2 is true and the code executed in case that the condition of line 1.4.1.2 is true are the only code sections that perform actions unique to CRSAT. But the conditions 1.2 and 1.4.1.2 are always false in our case.
Observe also that wave-underlined code (lines 1.4.1.1.2 and 1.4.1.1.2.1), implements non-chronological backtracking principle from 4.1.

4.2.4 Correctness Proof Sketch

In this sections we'll bring a sketch of correctness proof of CRSAT. We'll bring 2 theorems with intuitive explanations. Fool proof may be found in Appendix A.4.

Theorem 4.2.4.1: Let Σ be the CNF formula being checked by CRSAT algorithm. Let B be a decision variable (we'll suppose without restriction of generality that B is assigned to T when 1st side assigned). Let Ψ_1 be the conjunction of (non-implied variables scheme) conflict clauses corresponding to all the conflicts that happened while B was 1st side assigned. Let Φ be the conjunction of all conflict clauses corresponding to all the conflicts that happened while B was 1st side and 2nd side assigned. Let A_1(lowest decision level)…A_m be assigned non-implied literals, just before B 1st side assignment. Let $\bar{A} = A_1 \Lambda \ldots \Lambda A_m$. Then :

a. If $\neg(\Sigma \wedge \bar{A} \wedge B \rightarrow \square)$ (semantically, there is a model to Σ with $A_{1\ldots}A_m$=T; B=T) then **:**

a.1. The algorithm will return SAT and won't unassign B.

b. If $(\Sigma \wedge \bar{A} \wedge B \rightarrow \square)$ (semantically, there is no model to Σ with $A_{1\ldots}A_m$=T; B=T) then :

b.1. The algorithm will unassign B after 1st side assignment.

b.2. Just after B's 1st side unassignment the following expression is true:

$\Psi_1 \wedge \bar{A} \wedge B \rightarrow \square$ (Note that $\neg\Psi_1 \vee \neg\bar{A} \vee \neg B \Leftrightarrow T$ is identical to the previous formula, and may be proofed instead)

b.3. If $\neg(\Sigma \wedge \bar{A} \wedge \neg B \rightarrow \square)$ (there is a model to Σ with $A_{1\ldots}A_m$=T; B=F) then

b.3.1. The algorithm will assign B its second value (2nd side assignment). It will return SAT and won't unassign B.

b.4. If $(\Sigma \wedge \bar{A} \wedge \neg B \rightarrow \square)$ (there is no model to Σ with $A_{1\ldots}A_m$=T; B=F)

b.4.1. $\Phi \wedge \bar{A} \rightarrow \square$ (Note that $\neg\Phi \vee \neg\bar{A} \Leftrightarrow T$ is identical to the previous formula, so it may be proofed instead)

b.4.2. If B is assigned 2nd side value, the algorithm will unassign B at some stage.

Notes of theorem 4.2.4.1:

This theorem is the heart of the proof. What it says is that for any decision variable B :

1. If there is no contradiction induced by decisions for literals $A_1 \ldots A_m$ taken before deciding B, the algorithm will return SAT
2. If there is a contradiction induced by decisions for literals $A_1 \ldots A_m$(say $A_{1..m}$=T) taken before deciding B, the algorithm will backtrack after checking B and the conflicts recorded are enough to falsify every possible interpretation containing A_i=T for each i.

These statements are proven together by reverse induction on decision level of B. It means that when we prove it for some B, we know it's true for all the variables that will be assigned after assigning B (to both sides).

The first statement is proven by analysing the structure of the algorithm.

The second statement proof is more interesting. We'll provide below the ideas behind it.

Let Ψ_2 be the conjunction of (non-implied variables scheme) conflict clauses corresponding to all the conflicts that happened while B was 2nd side assigned.
The second statement proof for a decision variable B, s.t. conflicts are not checked for, is based on the fact that if:

- Ψ_1 falsifies any interpretation with $A_{1...m}$=T;B=T
- Ψ_2 falsifies any interpretation with $A_{1...m}$=T;B=F

Than :

- $\Phi = \Psi_1 \Lambda \Psi_2$ falsifies any interpretation with $A_{1...m}$=T.

The most interesting point to prove is that the theorem is true for decision variables conflicts are checked for.

First we'll explain why the decisions-based non-chronological backtracking principle from 4.1 is working. By the induction assumption Ψ_1 falsifies the interpretations with $A_{1...m}$=T; B=T. Observe, that if there are no conflict clauses containing ¬B, when checking its first side, than Ψ_1 falsifies all the interpretations with $A_{1...m}$=T for both values of B. Therefore even if we don't check the 2nd side of B, and take $\Phi=\Psi_1$, Φ meets the requirement of falsifying any interpretation with $A_{1...m}$=T.
If we take the example of fig. 4.2.1.b, than for the decision variable B :
$\Psi_1 = (\neg A \text{ v} \neg C) \Lambda (C)$. For A=T, Ψ_1 is indeed a contradiction.

Now, we'll explain why does the conflict checking principle works. Every clause ω from Ψ_1 falsifies some set of interpretations with $A_{1..m}$=T;B=T. If ω doesn't contain ¬B, it falsifies the same set interpretations, but for every value of B. Let us suppose that ω contains ¬B. Let Π be the set of conflicts recorded when checking conflict ω. Then $\omega \Lambda \Pi$ falsifies the same set of interpretations as ω, but for every value of B.

For example, let us take the variable A from example of fig. 4.2.1.b(in this example A is the decision variable we refer to as B in this section and m=0). Let us take the first conflict $\omega = \neg A \text{ v} \neg C$. ω falsifies the interpretations containing C=T;A=T. Π =

$A \vee \neg C$ in this case. $\omega \wedge \Pi = (\neg A \vee \neg C) \wedge (A \vee \neg C)$. Therefore $\omega \wedge \Pi \rightarrow \neg C$. And indeed $\omega \wedge \Pi$ falsifies C=T for every value of A.

Let's return from the example to the general proof explanation. Let's pay attention to the fact, that Φ is a conjunction of

1. Conflict clauses from Ψ_1 that don't contain $\neg B$
2. Conflict clauses from Ψ_1 that contain $\neg B$
3. Sets of conflict clauses recorded when checking conflicts from group 2 above

Observe, that the conjunction of clauses from (1) and (2) falsify all the interpretations with $A_{1\ldots m}$ =T;B=T.

As we have shown, every clause ω from (2) has appropriate set Π from (3), s.t. the conjunction $\omega \wedge \Pi$ falsifies the same set of interpretations as ω, but <u>for every value of B</u>.

Obviously every clause from (1) falsifies a set of interpretations independently of B value.

Now we have enough information to conclude that Φ falsifies every interpretation containing $A_{1\ldots m}$=T independently of B value. For more details and a formal proof, please refer to Appendix A.4.

We'll pass to the next theorem, which proves the soundness and completeness of CRSAT.

<u>Theorem 4.2.4.2 (soundness and completeness of CRSAT):</u>

Let Σ be a CNF formula. Than :

1) If Σ is SAT, CRSAT(Σ) returns SAT
2) If Σ is UNSAT, CRSAT(Σ) returns UNSAT

<u>Notes on proof of 4.2.4.2:</u>

The proof is straightforward from the previous theorem.

Let B be the first decision variable chosen by CRSAT. Suppose, without restriction of generality, it was first assigned T.

If Σ is SAT, than according to a.1 (if there is a model containing B=T) or b.3.1 (otherwise) the algorithm will return SAT.
If Σ is UNSAT, than according to b.4.2 it will unassign B and according to the algorithm's structure it will return UNSAT (more details in Appendix A.4).

4.2.5 CRSAT Enhancements Discussion

4.2.5.1 WCRSAT Algorithm

There is a problem that one encounters when trying to implement the CRSAT algorithm. The problem is that CRSAT requires to keep all the conflict clauses corresponding to non-implied variables scheme. It's obviously impossible, since the number of such clauses is exponential.
First thing to note is that we can remove each conflict clause after conflicts have been checked for all 1^{st} side variables appearing in it. But this is still not enough.
The solution is to keep all the information only for variables corresponding to last *p* decision levels, where *p* is a parameter. Conflicts will not be checked for variables of low decision level, which not all the conflicts are remembered for. This solution may be called W(indowed)CRSAT algorithm.

4.2.5.2 CRSAT and Clause Recording

Another problem with CRSAT algorithm, is that when checking conflicts paths may be repeated.
For example, lets look on 2 conflicts from those checked for A on fig 4.2.1.d : $\omega = \neg A \vee \neg D$ and $\gamma = \neg A \vee \neg E$.
Suppose we check ω. After assigning D to T, we should choose next decision variable. Let it be E.

Suppose we completed ω checking and we are checking γ. After assigning E to T, we should choose next decision variable. If it's D, than we are on path that has already been checked when checking ω.

The solution is to use clause recording. Clause recording schemes, which can be used for DPLL algorithm, may also be used for CRSAT.

4.3 New Pruning Methods

In this section we'll bring a few techniques which enhance the pruning. The first one is applicable for decision variables, we don't check conflicts for, therefore it can be used fot DPLL algorithm, the others are applicable for decision variables, that conflicts are checked for, and can only be used for CRSAT.
We'll bring a description of each technique. The pseudo-code changes to code of Appendix A.3 needed to implement each one of it and a formal proof, that each one of it doesn't hurt the soundness and completeness of CRSAT are provided in Appendix A.5

4.3.1 Using 2nd Side Variable Non-Participation in Conflicts

Suppose that we are running the DPLL/CRSAT algorithm on some CNF formula. The decision variables on the decision stack are $A_1 \ldots A_m$(bottom to top) and ¬B(topmost) and we have just completed checking the 2nd side of B(checking ¬B) and found no model . Suppose that, conflicts weren't checked for B and that B didn't participate in no conflict while it was 2nd side checked.
The main idea is that, in this case the fact that no model containing A_i=T;B=F was found is enough to prove that there is no model containing A_i=T;B=T, since the value of B had no influence on the proof, when B was F. Therefore, we can delete all the conflicts that were recorded, when B was 1st side checked.

To see, how this could prune the search, suppose, that A_m did participate in some conflicts, while B was 1st side checked, but didn't participate in no conflicts, while B was 2nd side checked. If we proceed according to the CRSAT algorithm, without deleting

conflicts, that were recorded when B was 1st side checked, we can't apply non-chronological backtracking principle of sec. 4.1 for A_m. If we do delete those clauses, we can apply it and therefore, we don't need to check the 2nd side of A_m!

Obviously, implementing this principle could influence not only A_m, but any one of A_i's. It can reduce the number of conflicts to check for A_i's or even eliminate it at all.

4.3.2 Using Conflict Variables Non-Participation in Conflicts

Suppose that we are running the CRSAT algorithm on some CNF formula. The decision variables on the decision stack are $A_1 \ldots A_m$(bottom to top) and ¬B(topmost). Let B be 2nd side decision variable, which we are checking conflicts for. Suppose we are just after completing checking one of the conflicts of B. Let the appropriate conflict clause of conflict that have been checked be $\omega = \neg A_{i(1)} \vee \ldots \vee \neg A_{i(k)} \vee \neg B \vee \neg C_1 \vee \ldots \vee \neg C_p$. Let γ be $C_1 \wedge \ldots \wedge C_p$. Let the set of conflict clauses recorded while checking ω be Π.

Let's consider the case <u>when all C_j's don't participate in Π</u>. The conflicts recorded are independent of C_j's. Therefore, <u>recording them proves, that no model exists with A_i=T;B=F</u>. Therefore the algorithm can keep only Π conflicts, deleting all the other conflicts, that were recorded when ¬B was checked and move on with backtracking, without checking other B conflicts.
Observe, <u>that if ¬B didn't participate in Π, 4.3.1 technique could be applied after applying our new technique and all the conflicts recorded when B was 1st side checked may also be deleted</u>!

4.3.3 Deleting Some 1st Side Conflicts After Conflict Checking

Suppose that we are running the CRSAT algorithm on some CNF formula. The decision variables on the decision stack are $A_1 \ldots A_m$(bottom to top) and ¬B(topmost). Let B be 2nd side decision variable, we have checked conflicts for. Suppose we have completed the conflicts checking (without applying 4.3.1 technique). Let ω_i be a conflict clause we

checked: $\omega_i = \neg A_{i(1)} \vee \ldots \vee \neg A_{i(k)} \vee \neg B \vee \neg C_1 \vee \ldots \vee \neg C_p$. Let γ_i be $C_1 \wedge \ldots \wedge C_p$. Let the set of conflict clauses recorded while checking ω_i be Π_i.
Suppose that ¬B doesn't participate in Π_i. In this case Π_i falsifies all the interpretations, falsified by ω_i and therefore a conflict represented by ω_i may be deleted. It's true for all ω_i's, s.t. ¬B doesn't participate in Π_i. For a detailed proof, please refer to Appendix A.5.4.
This method can be used only if the conflicts checking has ended in regular way - without using 4.3.1 method.

4.4 VAP (Variable Ordering Approximation) Heuristics

In this section a new VSIDS-style heuristics together with a data structure allowing to implement all the operations (including the division of all variables' counters by a constant) in O(1) are introduced.

Like in the case of VSIDS heuristics, a counter is hold for each literal. Its value is increased for each literal, participating in derivation (i.e. not only in the clause itself) of a new conflict clause. Initially its value is set to the number of appearances of respective literal in the initial set of clauses. We'll denote by c(A) the value of appropriate counter for literal A. Another notion will be for each variable A : $s(A) = p \cdot \max(c(A),c(\neg A)) + (1 - p) \cdot \min(c(A),c(\neg A))$. p is a configurable parameter, s.t. $0.0 \leq p \leq 1.0$. p allows us to choose whether to give preference to variable, both literals of which participate in many conflicts or to variable one literal of which participates in many conflicts. If p=0.5, than s(A)=c(A)+c(¬A). If p>0.5, than preference is given to variables, s.t. a literal with maximal c(A) of 2 of their literals has big c(A). If p<0.5, than preference is given to variables, s.t. a literal with minimal c(A) of 2 of their literals has big c(A).

All free variables are contained in a priority queue. It has q (a configurable even value) queues. Each queue contains linking list of variables. The queue number where each

variable is contained is determined according to s(A). Queue number i ($0 \leq i < q$) will point to linked list of variables, s.t. $(i \cdot (q/m)) \leq s(A) < ((i + 1) \cdot (q/m))$. *m* is a (configurable) number denoting the maximal value s(A) can hold, s.t. A could appear in the priority queue.
After s(A) for some A becomes greater than m, the counters for all literals are divided by 2 and thus all the free variables can stay in the queue. A "lazy" way of doing it (enabling to implement the division in constant time) is brought below in the description of how to add a literal to the priority queue.

Each time a new branching literal is asked for, the first variable (denoted A) from the uppermost queue is returned. The first literal to assign may be the literal with greater c(L) value of the 2 literals of A. It may also be a random literal or a choice based on some other function.

Each time a variable is assigned it's removed from the priority queue. Observe, that the value of s(A) may change only if the variable is assigned and therefore is not in the queue.

Each time a variable A is unassigned, it's inserted to the priority queue. If s(A) is lesser than m, the variable is inserted into appropriate queue.
The most interesting case is when s(A) is greater or equal to m. According to the said above variable with such s(A) cannot be inserted into the priority queue. The solution, brought below enables to implement the division by a constant (2 in our case) for every variable in constant time!

First, a list hold in queue number i is replaced as follows: for $i \geq (q/2)$ an empty list is inserted; for $i < (q/2)$ a concatenation of lists that where in queues $(2 \cdot i)$ and $(2 \cdot i) + 1$ is inserted. A list from $(2 \cdot i) + 1$ should be in the head. Observe that after this operation the order of variables stays the same. Observe also, that this operation takes constant time.

Now we could traverse the lists of all queues and divide the counters of all variables by 2. But this is a non-constant time implementation. What we do, is we are leaving the counters as they are. The counter for each literal will be divided by 2 (or power of 2 if a few such lazy divisions by 2 are needed) only after its next assignment, when it's accessed as a literal participating in derivation of a conflict clause (in order to increase its value) or when it's added to the priority queue after unassignment.
To implement this idea we should always remember how many times till now, the (lazy) division by 2 took place. We'll denote this counter by div2. We should also hold for each variable, what was the value of div2 when the variable has been lastly accessed to for purposes mentioned above. We denote this counter's value by ldiv2(A). In each access to variable A, before every other action (such as increasing c(A) by 1), we'll divide c(A) and c(¬A) by $2^{div2 - ldiv2(A)}$ and we'll set ldiv2(A)=div2.
To complete the description of the operations needed, when s(A) is greater or equal to m, we should mention, that A should be inserted into appropriate queue after dividing its counters by 2. If s(A) is still not less than m, c(A) and c(¬A) should be decreased, s.t. its value would be less than m (for example, one may set $c(A)=(m-1) \cdot (c(A)/(c(A)+c(\neg A))$; $c(\neg A)=(m-1) \cdot (c(\neg A)/(c(A)+c(\neg A))$).

The pointer to maximal queue, containing non-empty list of variables could be adjusted after removing last variable from the current maximal queue (j). In most cases the maximal queue would be j-1. Anyway, it could be found in O(log(j-1))=O(1) time.

We would like to mention, that in the removal operation we may remove not the first variable, but the variable with maximal s(A) counter from all the variables from the list of the maximal queue. This adjustment hurts the constant time implementation of the removal, but may hopefully return "better" variable.

Another possible idea is to allow more randomness. Picking up the first or second best variable randomly can do it, for example.

4.5 New Efficient Data Structure Representing SAT Formula (WLCC)

4.5.1 Intuitive Description

We introduce a new data structure for efficient SAT implementation. Its name is WLCC(Watched List with Conflicts' Counter) It combines the advantages of WL data structure - only 2 references to each clause, no need to visit clause on unassignment with the advantages of data structures with literal sifting - no need to traverse most of clause's cells during assignment. In addition, WLCC enables reducing number of visits to satisfied clauses and allows visiting fewer literals in newly recorded clauses.

In order to implement the new data structure, we'll have to keep a counter of current number of conflicts (referred to as CNC) and for each assigned variable A, we'll have to save CNC at the moment of last A's assignment. We'll this value as CN(A).

Each clause cell will contain:

1) literal number
2) pointer to next cell within the same clause (its usage is described later)

Each clause will always appear on lists of 2 of its literals (say, A and B). There will be only one clause cell at a time, s.t reference to it will appear on literals lists. This clause cell will be referred to as "clause head". It will contain either A or B (say A). The pointer to next cell of clause head will point to the cell of B.
Observe that this arrangement is different from all 3.4 data structures, where each cell can appear only on list of a literal the cell contains.

Pointers from (2) above will be kept in such way, that the clause head will be the head of list and if we begin to travel from it according to the pointers, we'll visit each clause cell exactly once (the last pointer will point to NULL). This list will be kept partially sorted according to non-increasing decision level. The purpose of it is that on each visit to the

clause, we won't have to travel through all the cells, but will stop when we reach an unsatisfied literal A, the clause is known to be sorted from. This idea is used in the "literals sifting" data structures in 3.4 but with 2 partially sorted lists.
The problem with this idea is that we have to visit clauses during unassignment and by this we lose the main advantages of WL data structure.
The solution is to save for each clause the conflict counter value at the moment of assignment of variable(A), the clause is sorted from. We'll denote this counter SF(Sorted From). According to SF, we'll know at each clause visit whether the assignment of A is the same assignment it was when the clause was lastly visited. If it's the same, we don't need to travel any further, since the literals beginning from it haven't change their values. If it isn't, we will continue to check the variables, till we find a variable, which decision level is less then or equal to SF.
The need for additional literal references and visiting clauses when unassigning is eliminated!
An important fact to mention here is that clause recording engine may be built in such way that every newly recorded clause is sorted according to decision level. SF may be set to the value of the CNC at the moment of the new clause recording. We'll show now, that this can considerably reduce the number of visited literals in a newly recorded clause α. Let n be the length of α. Suppose, that m literals of it were unassigned before visiting α for assigning one of its literals. Observe that only m literals will be visited according to our method, since the other n-m literals haven't been unassigned and reassigned(we can learn it from CN of first of them). This provides a significant time gain especially for long clauses. Such clauses are often deleted after unassigning some number of their literals. In this case the literals, which haven't been unassigned, won't be visited at all according to our method!

In next subsection we'll bring detailed algorithm for the operation of assignment.

4.5.2 Algorithm for VisitClause Function

Below is the pseudo-code of possible implementation of VisitClause function, a function called at each visit to a clause when a variable watched in the clause is assigned. It may return

- **SAT** : if the clause is satisfied
- **Unit** : if after the assignment the clause becomes unit
- **Contradiction** : if there is a contradiction in the clause
- **2NotAssigned** : if there are 2 or more not assigned variables in the clause.

Let Cell be a reference to a cell in the clause. Let Cell.Literal be the literal appearing in this cell and Cell.Next be the pointer to the next cell in the same clause.

VisitClause uses auxiliary function MakeWatchedInstead(Cell, WatchedCell).
WatchedCell is a reference to the clause head or the next cell after it.
Cell is a pointer to any clause cell, but the first 2.
MakeWatchedInstead removes WatchedCell from WatchedCell.Literal's watched list. It adds Cell to Cell.Literal's watched list. It also exchanges places of Cell and WatchedCell within the list defined by the "next" pointers within the clause. We won't bring here the code of MakeWatchedInstead, since it contains only technical details of pointers moving.

{SAT,Unit,Contradiction,2NotAssigned} VisitClause(Literal L):

1. LCell = ((ClsHead.Literal == L)?(ClsHead) : (ClsHead.Next)) /*LCell is a pointer to that of two first cells containing L*/
2. SecWCell = ((ClsHead.Literal != L)?(ClsHead) : (ClsHead.Next)) /*SecWCell is a pointer to that of two first cells not containing L*/
3. If (SecWCell.Literal is satisfied by current assignment)
    3.1. Return SAT
4. For (CellPtr = ClsHead.Next.Next; CellPtr != NULL; CellPtr = CellPtrNxt)
    4.1. CellPtrNxt = CellPtr.Next
    4.2. If (CellPtr.Literal is satisfied with current assignement)
        4.2.1. MakeWatchedInstead(CellPtr, SecWCell)
        4.2.2. Return SAT
    4.3. If (CellPtr.Literal is not assigned)
        4.3.1. If (SecWCell.Literal is not assigned)
            4.3.1.1. MakeWatchedInstead (CellPtr, LCell)
            4.3.1.2. Return 2NotAssigned
        4.3.2. MakeWatchedInstead (CellPtr, SecWCell)
    4.4. If (CellPtr.Literal is assigned, but not satisfied)
        4.4.1. If (CN(CellPtr.Literal) <= SF))
            4.4.1.1. Break from loop 4
5. If (Clause contains more than 2 literals)
    5.1. Sort the clause from ClsHead.Next.Next
    5.2. SF = CN(ClsHead.Next.Next.Literal)
6. If (SecWCell.Literal is not assigned a value)
    6.1. Return Unit
7. Else
    7.1. Return Contradiction

A few important observations:

1) The sorting at line 5.1 is relatively inexpensive for short clauses. If the clause is of length n, than only n-2 literals are sorted. Since most of the clauses are relatively short, the sorting isn't taking a lot of time. The solution for long clauses may be as follows: instead of sorting at the end, testing whether the clause is sorted together

with some limited sorting possible for one pass over the sorted list may be carried out in the main loop. If the clause is found to be sorted, SF should be updated at the end, otherwise it shouldn't be updated.

2) If the algorithm exits at line 4.2.2, then if this clause is to be visited before unassignment of the satisfied literal (now appearing in one of the first two cells), the execution of VisitClause will always be terminated at line 3.1 and therefore will be of O(1) complexity. This is an important saving of running time.

4.5.3 Example

We'll bring below an example of WLCC management. Suppose we have a clause containing 5 literals A, B, C, D and E, s.t. variable A was assigned F at decision level 3 and conflict counter at the moment of assignment was 20, variable B was assigned F at decision level 1 and conflict counter at the moment of assignment was 10. C, D and E aren't assigned.

Below is a representation of all relevant information.

Rows, which first column box is grey belong to the clause. Rows, which first column box is white represent literals' information for appropriate column.

SortedFrom row represents the SF value for the clause. Rows 2, 3 and 4 hold the data of clause cells (literal, pointer to next literal and watched status). "H" in "Watched" row means clause head, "N" means watched literal, which is not clause head (but next after it).

Sorted From	-1				
Literals	A	B	C	D	E
Next. Lit. Ptr.	-	A	B	C	D
Watched	-	-	-	N	H
Dec. level	3	1	-1	-1	-1
CN	20	10	-1	-1	-1
Assign	F	F	-	-	-

Suppose we assign E a value F at decision level 5, conflict counter 30. Below is the representation of our clause after VisitClause execution till line 5.

Sorted From	-1				
Literals	A	B	C	D	E
Next. Lit. Ptr.	-	A	D	E	B
Watched	-	-	H	N	-
Dec. level	3	1	-1	-1	5
CN	20	10	-1	-1	30
Assign	F	F	-	-	F

Below is the representation of our clause after full VisitClause execution.

Sorted From	30				
Literals	A	B	C	D	E
Next. Lit. Ptr.	B	-	D	E	A
Watched	-	-	H	N	-
Dec. level	3	1	-1	-1	5
CN	20	10	-1	-1	30
Assign	F	F	-	-	F

Suppose now, we assign C a value F at decision level 7, conflict counter 40. Observe, that the “for” loop will be broken at first iteration, at line 4.4.1.1 and the clause will become unit.

Suppose, that we unassigned C, E and A (i.e. backtracked till level 3), then assigned E the value F at decision level 3, conflict counter 50 and assigned A the value T at decision level 4, conflict counter 60 and we are just before assigning D a value F :

Sorted From	30				
Literals	A	B	C	D	E
Next. Lit. Ptr.	B	-	D	E	A
Watched	-	-	H	N	-
Dec. level	4	1	-1	-1	3
CN	60	10	-1	-1	50
Assign	T	F	-	-	F

Suppose we visit the clause to assign D a value F (decision level 7, conf. co. 80). Observe, that the “for” loop won’t be broken when we visit E, since E has been assigned when conflict counter was greater than 30. Therefore, we will reach A and discover that the clause is satisfied. The data will be as follows after line 4.2.1 :

Sorted From	30				
Literals	A	B	C	D	E
Next. Lit. Ptr.	D	-	B	E	C
Watched	H	-	-	N	-
Dec. level	4	1	-1	7	3
CN	60	10	-1	80	50
Assign	T	F	-	F	F

4.6 Using Low Decision Level Variables for Clause Recording

4.6.1 Recording Clauses of Low Decision Level Variables

Suppose that we are running the DPLL(or CRSAT) algorithm on some CNF formula. Suppose we have just completed the checking the 2nd side of B(checking ¬B). Suppose that all the non implied variables, excluding B on the stack are 1st side variables A_1(smallest decision level)…A_m. Suppose, without restriction of generality A_1=T,…,A_m=T under current assignment.

This situation means that there is no model for which A_1=T,…,A_m=T, otherwise it would have been found. Therefore, we can add the clause $\neg A_1 \vee \ldots \vee \neg A_m$ to the system. Moreover, if only part of $A_1 \ldots A_m$ variables participated in conflicts, say $A_{i(1)} \ldots A_{i(n)}$, than it's more efficient to record the shorter clause $\neg A_{i(1)} \vee \ldots \vee \neg A_{i(n)}$.

The problem is that adding such clause doesn't provide us information, if we proceed with regular DPLL or CRSAT, since after adding such clause, we'll always be checking the second side of one of $A_{i(1)} \ldots A_{i(n)}$ and therefore the new clause will always be satisfied. The situation is different if we are using restarts. We'll discuss it below.

4.6.2 New Restarts Policy

Suppose we are in the situation described in previous subsection. Suppose that we have just added a new clause $\neg A_{i(1)} \vee \ldots \vee \neg A_{i(n)}$. Observe, that if we restart now, no path will be visited for the second time after restart. This leads us to a new restarts policy.

We restart if 1st side and implied variables are the only variables assigned and we add an appropriate clause each time. This approach to restarts allows us to ensure, that no path is repeated from previous restarts. It also allows us to use all the relevance-based learning techniques, (which is not the case for the idea of "the search signature" from [24] described in 3.2.2).

The drawback of this idea is that we can't restart at any time, but only when all the non-implied variables assigned are 1st side variables.

Combination of the regular restarting strategy and one proposed here can be used to overcome this restriction. We can sometimes use restarts without adding a clause, but also without requiring all non-implied variables to be 1st side variables.

4.7 Dynamically Changing 1st Side Variables Based on Conflict Analysis

Suppose, we are running DPLL or CRSAT algorithm. Suppose that we have just encountered a conflict and the following variables are on the decisions stack (from lowest to highest decision level): $B_1 \ldots B_n, A_1 \ldots A_m$, where B_i's are any variables and A_j's are 1st side or implied variables. We'll denote $A=\{A_1,A_2,\ldots,A_m\}$. Let the decision level of A_1 be *dl*.

Suppose we have found a set of literals which is a conflict's reason according to an implication graph cut (see section 3.2.1.2), s.t. no more than one variable from each decision level more than or equal to *dl* is in the conflict's reason.

If n=0 (there are no B_i's), than examples for such cuts are non-implied variables cut (3.2.1.2.3) or AllUIP cut(3.2.1.2.4). Other such cuts are possible. For example, 1UIP+NIV(see section 4.7.2). We'll denote a set of literals appearing on the reason side of such cut(for now it can be any cut) by $\bar{A}$.

The main idea is that now we can unassign all the variables corresponding to literals of A and assign the variables, corresponding to the set of literals in $\bar{A}$ as decision variables. After such a set of assignments the same conflict is produced, but now:

- there tends to be less(in worst case it might be equal number of) decision levels
- there tends to be less variables corresponding to each decision level

- variables assigned, but not relevant for this conflict tend to be unassigned
- first UIP's at each decision will tend to be closer to the decision variables (more about it in 4.7.4).

We refer to such action as *shrinking the set of assigned variables.* Different schemes of doing it can be used. We refer to it as to different *shrinking schemes*.

In the following subsections we'll explain in more details the actions required to implement the new idea for different shrinking schemes. We'll bring 3 different shrinking schemes: non-implied variables, 1UIP+NIV and AllUIP. Other schemes are also possible.

In what follows when we refer to "implication graphs cuts" / "literals" on different sides of cut, we mean cut that corresponds to "partition of A literals" / "literals of A". It's irrelevant for our matter on which side of the cuts are variables which aren't A members.

4.7.1 Non-Implied Variables Shrinking Scheme

This shrinking scheme corresponds to non-implied variables cut(3.2.1.2.3).

The actions that are taken to implement the non-implied variables shrinking scheme are as follows:

1. Identify the set of literals Π on the reason side of a cut according to non-implied variables conflict-recording scheme (3.2.1.2.3).
2. Put literals from Π on some stack S in an order imposed by corresponding variables decision level. Literal with lowest decision level will be on top of the stack.
3. Unassign all the A variables
4. Pop literal from S, assign it, run BCP.
5. Repeat step 4 till S becomes empty.

We'll illustrate the idea by an example. Suppose, that the decision stack is one on 4.7.1.a. Suppose, that A is of decision level 5. Of course, there are variable with smaller decision level than A, but they are not displayed.

A relevant implication graph is displayed on 4.7.1.b. It doesn't include variables with smaller decision level than A, even if they are connected to the conflict. It does, however, include all variables with greater decision level, even if they are not connected to the conflict, but those variables are not relevant for the non-implied variables cut.

After identifying that the non-implied variables connected to the conflicting variable are D and G, we

- unassign A,B,C,D,E,J,G,H,I
- assign D than run BCP
- assign G and run BCP

The resulting branch is on 4.7.1.c. Observe that E that was an implied variable before the shrinking is now not assigned.

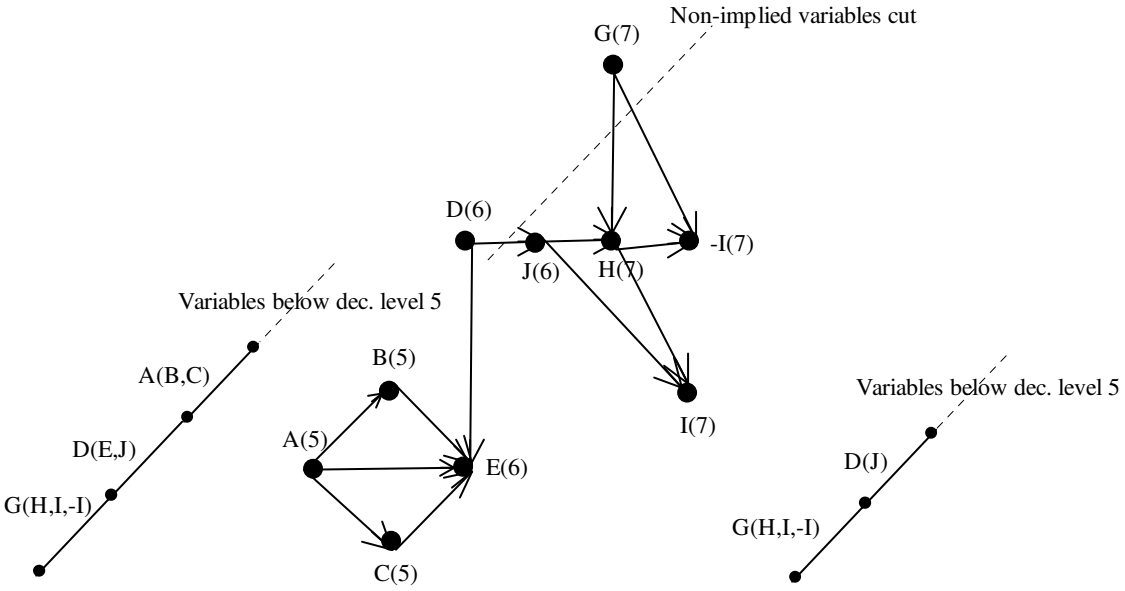


Fig. 4.7.1.a

Fig. 4.7.1.b

Fig. 4.7.1.c

4.7.2 1UIP+NIV Shrinking Scheme

This scheme doesn't correspond to a cut described in 3.2.1.2. We define the 1UIP+NIV as a cut having the first UIP of the last decision level and the last UIP's (i.e. non-implied variables) of other decision levels on its reason side.

The actions that are taken to implement the 1UIP+NIV shrinking scheme are as follows:

1. Identify the first UIP of the last decision level. We'll refer to it as C.
2. Let Γ be the set of literals which there is a path from C to it.
3. Recursively find the set of non-implied literals Π, which there is a path from it to Γ.
4. Let $\Theta=\Pi \cup \{C\}$
5. Put literals from Θ on some stack S in an order imposed by corresponding variables decision level. Literal with lowest decision level corresponding will be on top of the stack.
6. Unassign all the A variables
7. Pop literal from S, assign it, run BCP.
8. Repeat step 7 till S becomes empty.

Below is an example. 4.7.2.a is a decision tree branch before the shrinking, 4.7.2.b is the relevant implication graph. 4.7.2.c is the decision tree branch after the 1UIP+NIV shrinking.

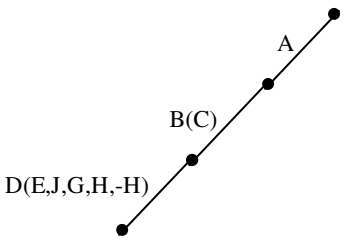


Fig. 4.7.2.a

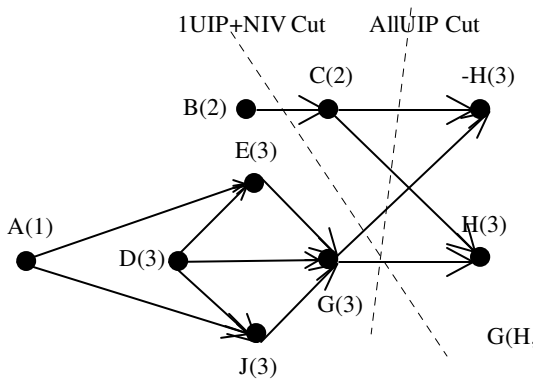


Fig. 4.7.2.b

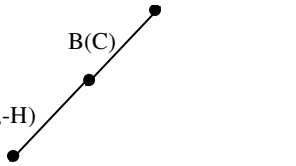


Fig. 4.7.2.c
(after 1UIP+Dec shrinking)

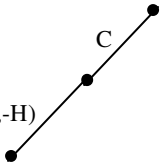


Fig. 4.7.2.d
(after AllUIP shrinking)

4.7.3 AllUIP Shrinking Scheme

This scheme corresponds to the AllUIP cut (3.2.1.2.4). Its implementation is very similar to that of 1UIP+NIV scheme. All 8 actions from previous subsection are valid here. The only difference is in action 3, where should appear "Recursively find the set Π, which is a set of first UIP's which have literals connected to Γ".

Above is an example. 4.7.2.a is a decision tree branch before the shrinking, 4.7.2.b is the relevant implication graph. 4.7.2.d is the decision tree branch after the AllUIP shrinking.

4.7.4 Conclusions

We introduced in this section 3 schemes for assigned stack shrinking. Obviously other schemes are also possible. Below we'll list two major reasons for using the shrinking.

1. By shrinking the set of assigned variables, we unassign irrelevant variables and as a result our decision tree corresponds better to conflicts identified. This can help pruning the search.

2. The CRSAT algorithm requires recording clauses corresponding to the non-implied variables clauses-recording scheme, even if we don't use non-implied variables scheme for clause recording. It would save plenty of space and time of conflict-analysis, if we used the non-implied variables scheme for clauses recording as well. However, the non-implied variables conflict-recording scheme is not really efficient comparing to schemes like 1UIP and others ([31]). But if we use shrinking and afterwards the non-implied variables scheme, than it is much more efficient. This is true, because the first UIP's at each decision level higher than *dl*, variables will tend to be closer (1UIP+NIV scheme) or identical (AllUIP scheme) to the decision variables. Therefore, clauses recorded by non-implied variables scheme after the shrinking are close to clauses recorded by UIP-based schemes both before and after the shrinking.

4.8 Extending Relevance-Based Learning by Literals' Density

In this section, we'll introduce a new relevance-based learning technique. We'll first explain the idea behind it.

Suppose that we are recording clauses using non-implied variables scheme. We would like to remind that using shrinking from previous section makes the non-implied variables scheme much more efficient by making clauses recorded by it close to different UIP-based schemes.

Suppose that the decision variables on the stack (after the shrinking if we used it) are A_1(lowest decision level)…A_m. Let $A=\{A_{1,\ldots,}A_m\}$.

Observe that if the new clause consists of negations of all A_i's, the new clause will always be satisfied, unless restart takes place. That is true, because one of 1st side A variables (not always the same one) will always be flipped and thus will satisfy the clause.

Moreover, if the new clause consists of negations of A_i's, for all $i : k < i \leq m$, where k is any number, s.t. 1<k<m, it will be also always satisfied before all its literals are finally unassigned.

We would like to give preference to clauses, which are not only short, but also tend to be satisfied less during subsequent search. For fulfilling last requirement, the clause's literals should be less "dense" with respect to decision level.

We also would like give preference to clauses, which are less dense near the last decision level. We'll explain the reason for it on an example. Suppose that A variables are 1st side decision variables. Let $\omega = \neg A_{m-3} \vee \neg A_{m-2} \vee \neg A_m$; $\gamma = \neg A_{m-3} \vee \neg A_{m-1} \vee \neg A_m$. After unassigning A_m, and assigning the 2nd side of A_{m-1}, ω becomes a unit clause, which prunes the search and γ is satisfied. Then after unassigning A_{m-1}, and assigning the 2nd side of A_{m-2}, γ becomes a 2 literals clause and ω is satisfied. Observe, that there is a difference between the influence of ω and γ on the search when those aren't satisfied. The difference is in the fact, that ω becomes a unit clause and γ becomes 2 literals clause. Therefore ω has more chances to prune the search. This is due to the fact, that ω is less dense near A_m. Observe, that after unassigning A_{m-2} both clauses have similar chances to prune the search.

Now we are ready to describe the new relevance-based learning technique. First we'll bring a few definitions.

Let ω be a recorded clause: $\omega = \neg C_1 \vee \ldots \vee \neg C_n$; $C = \{C_1 \ldots C_n\}$ and the following is true:

- C_i's are sorted according to decision level and C_1 has the lowest decision level between C_i's
- For all i : C_i is in A.

Let $B=\{B_1 \ldots B_s\}$ be set of all literals from A, s.t.

- B_i's are sorted and B_1 has the lowest decision level between B_i's
- B and C are disjoint
- The decision level of B_1 is greater than the decision level of C_1

We define a function CM : B→N :

$CM(B_i)$ = the number of C_i's with greater decision level than B_i

Let *t* be a parameter defining threshold on recorded clause length. If clause length is less or equal to *t*, than the clause is kept forever. Longer clause is deleted after *t* of its literals become unassigned. Let *d* be a parameter defining threshold on literals density in a new clause.

We'll now define *literals' density* for a newly recorded clause ω :

$$LD(\omega) = \sum_{i=1..s} (t\text{-}CM(B_i)) \cdot \begin{cases} 1: \text{if } t \geq CM(B_i) & \text{(gives preference to less dense clauses)} \\ -1: \text{otherwise} & \text{(gives preference to shorter clauses)} \end{cases}$$

Now we'll describe the actions on recording a new clause ω :

1. Calculate LD(ω) (it can be done in the process of conflict recording with very little time overhead)
2. If (LD(ω)>d)
   2.1. Do not record ω
3. Otherwise
   3.1. Record ω.
   3.2. If (|ω|>t)
      3.2.1. Delete ω after unassining t of its literals
   3.3. Otherwise
      3.3.1. Leave ω forever

5 Putting It All Together – Jerusat Solver

In this section we'll introduce a new SAT solver – Jerusat, which utilises the ideas brought in previous sections. In section 5.1 we'll provide general information about Jerusat. In section 5.2 we'll bring its performance comparing with performance of existing state of art SAT solvers zChaff([12]) and Limmat (http://www.inf.ethz.ch/personal/biere/projects/limmat/) which were very successful in recent SAT competition([43]).
More information including description of Jerusat internal structure and extensive analysis of its performance is provided in Appendix B.

5.1 General Information

Jerusat has been implemented in C language under Windows platform, but is also available under UNIX. The implementation has approximately 7680 lines of code.

Jerusat implementation may be divided into implementation of 6 following aspects :

1. **WCRSAT(4.2.5.1)** implementation(~3430 code lines), which includes also **decision-based non-chronological backtracking (4.1)** and **new pruning methods(4.3)** implementation. It's important to state, that since CRSAT(and WCRSAT) is a generalisation of DPLL it's possible (and very simple) to configure Jerusat, s.t. it would implement DPLL. In this case, it would still use the decision-based non-chronological backtracking. It will also use the pruning method of section 4.3.2, which is available for DPLL as well as for CRSAT.
2. Clause recording(~1000 code lines), including the literals' density **relevance-based learning (4.8)**.
3. Restarts policy(~120 code lines), including a possibility to use the **new restarts policy from 4.6.2**.

4. Heuristics(~510 code lines), implementing the **VAP heuristics (4.4)**.
5. Data Structures(~700 code lines), using the **WLCC data structure (4.5)** for representing clauses.
6. Shrinking schemes(~150 code lines), which makes possible using **non-implied variables shrinking scheme (4.7.1)**.

Rests of lines of code are used to implement different auxiliary operations.

5.2 Jerusat Performance Comparing with Limmat and zChaff

Below is a table comparing the performance of Jerusat, Limmat(version 1.0) and zChaff(version z2001.2.17).

Jerusat is used with its default configuration(see appendix B.2.2 for more information about configuring Jerusat). Limmat and zChaff are used with default configuration as well.

The performance was checked on a computer with 128Mb of memory, "x86 Family 6 Model 5 Stepping 2 Genuine Intel ~350 Mhz" processor and "Microsoft Windows 2000 Professional" operating system. Limmat and zChaff are originally written under UNIX, but were successfully compiled under Windows.

There is no set of benchmarks known to represent well the CNF functions We tried to choose a reasonable set of benchmark families received from different problems. More information about the benchmarks we used can be found in appendix B.2.1.

We set a cut off time of 2 hours. We added 7200 seconds for every instance that has been cut off after 2 hours. Best time on every benchmark family is highlighted.

	Jerusat		Limmat		zChaff	
Benchmarks Family	Average in seconds (7200 for a cut instance)	Cut After 2 Hours	Average in seconds (7200 for a cut instance)	Cut After 2 Hours	Average in seconds (7200 for a cut instance)	Cut After 2 Hours
aim-200	0.03125	0	0.050833	0	0.599208	0
ais	0.01	0	99.78325	0	48.39675	0
beijing	1010.768	2	959.8232	2	993.4041	2
bf	0.24275	0	0.418	0	1.044	0
blocksworld	8.475143	0	8.204571	0	13.65114	0
bmc	28.383	0	67.42838	0	2346.6	4
dubois	0.035	0	0.0125	0	0.494417	0
flat200-479	0.38391	0	2.4815	0	6.76601	0
fvp-unsat.2.0	5063.531	15	4985.998	15	3445.406	7
hanoi	74.522	0	3603.853	1	3605.123	1
ii16	0.1773	0	6.43	0	19.1849	0
ii32	0.900176	0	0.362706	0	0.931235	0
jnh	0.0202	0	0.0316	0	0.1485	0
logistics	0.3605	0	0.731	0	5.7635	0
parity16	1.9605	0	11.0173	0	20.8417	0
phole	172.7922	0	40.2456	0	23.724	0
pret150	1.1535	0	0.07	0	2.9635	0
qg	15.47486	0	80.90673	0	65.92155	0
ssa	0.05	0	0.09125	0	0.405375	0
sw100-8-lp0-c5	0.0021	0	0.0095	0	0.1103	0
uf150-645	0.35824	0	0.68928	0	2.81278	0
uuf150-645	1.22619	0	4.51746	0	12.66249	0
SUM	6380.858		9873.155		10616.95	

Table 5.2.1

As one can see, Jerusat outperforms both Limmat and zChaff in case of 15 out of 22 benchmark families. Jerusat also outperforms Limmat and zChaff in terms of sum of time. See also Appendix B.2.4 for comparison of Jerusat vs. Limmat and zChaff in terms of magnitude.

It should also be said, that Jerusat uses much less memory than Limmat and zChaff. For example, let's take an instance 2pipe.cnf of fvp-unsat.2.0, which is solved in about the same time by all solvers. Jerusat uses maximum of 1.8 Mb of memory to solve it, Limmat uses 2.6 Mb and zChaff uses 9.0 Mb.

The difference is even more significant for harder instances. Let's take instance 3bitadd_31.cnf of beijing family. After 2 minutes of running, Jerusat used 6.5 Mb, Limmat used 20.5 Mb and zChaff used 44 Mb of memory.

6 Literature

[1] S.A Cook The complexity of theorem proving procedures. In *Proceedings of the Third ACM symposium on Theory of computing,* pp. 151-158, 1971.

[2] M.R. Garry and D.S. Johnson. *Computers and Intractability: A Guide to the theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, 1979.

[3] B.Selman, H. Levesqe, and D. Mitchell. A new method for solving hard satisfiable problems. In *Proceedings of the National Conference on Artificial Intelligence,* p. 440-446, 1992.

[4] B. Selman and H. Kautz. Domain-independent extensions to GSAT : Solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence,* p. 290-295, 1993.

[5] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence,* 1996.

[6] D. Du, J. Gu, and P,M. Pardalos, editors. *Satisfiability Problem: Theory and Applications,* volume 35. American Mathematical Society, 1997.

[7] M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem proving." *Communications of the ACM,* (5):394-397, 1962.

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the association for Computing Machinery,* 7:201-215, 1960.

[9] R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence,* p. 203-208, 1997

[10] J.P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design,* pages 220-227, November 1996.

[11] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deductuion,* p. 272-275, July 1997.

[12] M. Moskewicz, C. Madigan, Y.Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference,* 2001.

[13] P. Gent and Toby Walsh: The Search for Satisfaction, 1999. Online : http://dream.dai.ed.ac.uk/group/tw/sat/index.html

[14] J. Franco: Some Interesting Research Directions in Satisfiability. In *Annals of Mathematics and Artificial Intelligence 28*(1-4): 7-15 (2000)

[15] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah: Algorithms for the Satisfiability(SAT) Problem: A Survey. In *Discrete Mathematics and Theoretical Computer Science: Satisflability (SAT) Problem*, vol. 35, pages 19-152 (1997).

[16] E. Birnbaum, E.L. Lozinski: The Good Old Davis-Putnam Procedure Helps Counting Models. In *Journal of Artificial Intelligence Research*, vol. 10, 1999, 457-477.

[17] R. Zabih and D.A. McAllester, "A Rearrangement Search Strategy for Determining Propositional Satisfiability", in *Proceedings of National Conference on Artificial Intelligence,* pp. 155-160, 1988.

[18] J.P. Marques-Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA),* September 1999.

[19] E. Goldberg and Y. Novikov. BerkMin : a Fast and Robust Sat-Solver. In *Design, Automation, and Test in Europe (DATE '02)*, March 2002, pp. 142-149.

[20] I. Lynce and J. Marques-Silva. Efficient Data Structures for Fast SAT Solvers. *International Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, May 2002

[21] C. P. Gomes, B. Selman and H. Kautz. Boosting Combinatorial Search Through Randomization, in. *Proceedings of the National Conference on Artificial Intelligence,* July 1998.

[22] E.T. Richards and B. Richards. Non-Systematic Search And No-Good Learning. *Journal of Automated Reasoning*, 24(4):483-533, 2000.

[23] L. Baptista and J. Marques-Silva. Using Randomization and Learning to Solve Real-World Instances of Satisfiability. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming,* 2000.

[24] L. Baptista, I. Lynce and J. Marques-Silva. Complete Search Restart Strategies for Satisfiability. In *the IJCAI'01 Workshop on Stochastic Search Algorithms (IJCAI-SSA)*, August 2001

[25] M. Herbstritt. Improving Propositional Satisfiability Algorithms by Dynamic Selection of Branching Rules, April 2001. Online : http://ira.informatik.uni-freiburg.de/~herbstri/publications/Her_2001b.pdf .

[26] M.Buro and H. Kleine-Büning. Report on a SAT competition. *Technical report, University of Paderborn*, November, 1992.

[27] J.W. Freeman. Improvements to Propositional Satisfiability. *PhD thesis, University of Pennsylvania, Philadelphia(PA),* May 1995

[28] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI),* pages 315-326, 1997.

[29] R.G Jeroslaw and J. Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1:167-187, 1990.

[30] M. Herbstritt. zChaff : Modifications and Extensions. September, 2001. Online : http://ira.informatik.uni-freiburg.de/~herbstri/publications/Her_2001.pdf

[31] L. Zhang, C.F. Madigan, M.H. Moskewicz, S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver . In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279-285, November 2001.

[32] J. de Kleer, "An Assumption-Based TMS", *Artificial Intelligence,* vol. 28, pp. 127-162, 1986.

[33] R.M. Stallman and G.J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis", *Artificial Intelligence,* vol. 9, pp. 135-196, October 1977.

[34] J. P. Marques Silva, "An Overview of Backtrack Search Satisfiability Algorithms", in *Fifth International Symposium on Artificial Intelligence and Mathematics*, January 1998.

[35] J.P.M. Silva and A.L. Oliveira, "Improving Satisfiability Algorithms with Dominance and Partitioning", in *International Workshop on Logic Systems,* May 1997.

[36] L. Dryke, A. Frisch, I. Lynce, J.M. Silva and T. Walsh, "Comparing SAT Pre-processing Techniques", in *Ninth Workshop on Automated Reasoning*, April 2002.

[37] I. Lynce and J. Marques-Silva. The interaction between simplification and search in propositional satisfiability. In *the CP'01 Workshop on Modeling and Problem Formulation (CP-FORMUL)*, November 2001.

[38] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, Solving Difficult SAT Instances in the Presence of Symmetry. In *Design Automation Conference* (DAC), New Orleans, Louisiana, pp. 731-736, 2002.

[39] J.M. Crawford and L.D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence,* pages 21-27, 1993.

[40] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problem. In *Proceedinbgs of the International Conference on Principles and Pratictice of Constraint Programming,* 1997.

[41] J.P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In *International Conference on Principles and Practice of Constraint Programming,* pages 537-542, September 2000.

[42] Daniel le Berre. Exploiting the real power of unit propagation lookahead. In *LICS Workshop on Theory and Applications of Satisfiability Testing,* June 2001.

[43] T Laurent Simon, Daniel Le Berre and Edward A. Hirsch. The SAT2002 competition (preliminary draft), August 2002. Online http://www.satlive.org/index.jsp.

[44] Holger H Hoos & Thomas Stützle at Darmstadt University of Technology. SATLIB - The Satisfiability Library. Online : http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/ .

Appendix A - Detailed Pseudo-Code and Formal Proofs

Appendix A.1 - DPLL Detailed Pseudo-Code

In this appendix, we'll introduce a non-recursive implementation of the DPLL algorithm. It reveals some details of the unit clauses identification process (BCP process).
This is the base for the CRSAT pseudo-code brought in Appendix A.3.

We'll suppose, that given CNF formula doesn't contain unit clauses (otherwise they can be easily removed using the unit clause rule during pre-processing before running DPLL).
The algorithm uses the following mappings(which may be implemented by an array):

1. ***IfAssigned****: Variables ->{T,F}.*
   Indicates whether the variable is assigned a value.
2. ***AssignementSign****: Variables->{T,F}.*
   Valid only for A, s.t. IfAssigned(A)==T.
   Indicates the sign of the variable under current assignment.

Note, that the 2 mappings above define a partial assignment.

3. ***AssignmentStatus****: Variables->{NULL,Side1,Side2,Implied}.*
   AssignmentStatus(A)==
   NULL: iff IfAssigned(A)==F.
   Side1: if current value of A was not imposed by other assignments and that's the first value for A we are checking.
   Side2: if current value of A was not imposed by other assignments and that's the second value for A we are checking.
   Implied: if current value of A was imposed by other assignments.
4. ***ImplicationClause****: Variables->Clauses.*
   Valid only for A, s.t. IfAssigned(A)==T.

For A, s.t. AssignmentStatus(A)=Implied, ImplicationClause(A) points to a clause, where the implication has occurred during the BCP process.
For A, s.t. AssignmentStatus(A)=Side1 or Side2, ImplicationClause(A)==0;
Saving the implication clause is not necessary for the algorithm to work, but it is crucial for implementing enhancements to DPLL and the CRSAT algorithm.

In addition, the algorithm uses:

5. ***AssignedVars*** : a stack with variables assigned a value.
6. ***ToBeImplied*** : a set holding variables, that have been assigned a value, but necessary assignments have not yet been deduced from it.

Note that, the CNF formula itself is not mentioned explicitly in the algorithm. It's considered to be a global variable and all the references to clauses refer to clauses of the CNF formula.

The algorithm is built from five functions.

1. The main function is **DPLL**. It doesn't have parameters. It returns SAT, which stands for "the formula is satisfiable" or UNSAT, which stands for "the formula is unsatisfiable".
2. The function **BCP** implements the BCP process. It doesn't have parameters. It returns either NULL, if there is no contradiction or a reference to a clause, where the contradiction has happened.
3. The function **PreAssign**. It manages the data necessary for assigning variable a value before the BCP process. It receives 4 parameters: Variable A; Implication Clause ω; Assignment Sign a; Assignment Status s. It doesn't return nothing.
4. The function **Assign.** It finally assigns a value to a variable after the BCP process for it. It receives variable A and doesn't return nothing.
5. The function **UnassignTopVar.** It unassigns top variable from AssignedVars. It receives nothing and returns the variable unassigned.

Now we'll introduce algorithm's pseudo-code:

{SAT, UNSAT} : DPLL()

1. ToBeImplied = {}
2. While (T)
   2.1. if (ToBeImplied == {})
      2.1.1. A = Any variable, s.t. IfAssigned(A)==F
      2.1.2. PreAssign(A, 0, Any value from {T,F}, Side1)
   2.2. ClauseWithContradiction = BCP()
   2.3. if ((ClauseWithContradiction == NULL) AND (For All A : IfAssigned (A)==T))
      2.3.1. Return SAT
   2.4. if (ClauseWithContradiction != NULL)
      2.4.1. while ((Not Empty(AssignedVars)) AND (AssignmentStatus(StackTop(AssignedVars)) != Side1))
         2.4.1.1. A = UnassignTopVariable()
      2.4.2. If (IsEmpty(AssignedVars))
         2.4.2.1. Return UNSAT
      2.4.3. A = StackTop(AssignedVars)
      2.4.4. If (AssignmentStatus(A) == Side1) /*That's always the case if we are here*/
         2.4.4.1. UnassignTopVariable()
         2.4.4.2. PreAssign(A; 0; ¬AssignmentSign(A); Side2)
         2.4.4.3. Continue with loop 2

void PreAssign(Variable A; Implication Clause ω; Assignment Sign a from {T,F};

Assignment Status s from {NULL, Side1, Side2,Implied})

1. ImplicationClause(A) = ω
2. AssignmentSign(A) = a
3. AssignmentStatus(A) = s
4. ToBeImplied = ToBeImplied U {A}

void Assign(Variable A)

1. IfAssigned(A) = T
2. Push(AssignedVars, A)

VariableUnassigned : UnassignTopVariable()

1. V = Pop(AssignedVars)
2. IfAssigned(A) = F
3. Return A

ClauseWithContardiction : BCP()

1. While (Not Empty(ToBeImplied))
    1.1. A = any variable in ToBeImplied
    1.2. ToBeImplied = ToBeImplied \ {A}
    1.3. Assign(A)
    1.4. For each clause ω, s.t. ω(A) == ¬AssignmentSign(A) :
        1.4.1. If (Exists B, s.t. ω(B)==AssignmentSign(B))
            1.4.1.1. Continue with the loop 1.4.
        1.4.2. If (Exist B,C, s,t, (B≠C And (ω(B)!=X) And (ω(C)!=X) And
            (AssignmentStatus(B)== AssignmentStatus(C)==NULL))
            1.4.2.1. Continue with the loop 1.4.
        1.4.3. If (Exists B s.t. ((ω(B)!=X) And (AssignmentStatus(B)==NULL)))
            1.4.3.1. PreAssign(B, ω, ω(B), Implied)
            1.4.3.2. Continue with the loop 1.4.
        1.4.4. While (Not Empty(ToBeImplied))
            1.4.4.1. C = any variable in ToBeImplied
            1.4.4.2. Assign(C)
        1.4.5. Return C (*There is a contradiction!*)
2. Return NULL

Appendix A.2 - Non-Implied Variables Scheme Conflict Recognising

Below is a procedure for identifying non-implied variables after a conflict. It can be used in the DPLL algorithm from Appendix A.1 just after a contradiction is reached (line 2.4).

The parameter to it should be the ClauseWithContradiction returned by BCP. The resulting conflict clause is received by taking the disjunction of negations of all literals of ConflictReason after running GetNonImpliedConflictClause.

GetNonImpliedConflictClause (Clause ClsWithContradiction) /*Supposes that ConflictReason is empty*/

1. For each literal A from ClsWithContradiction do
    1.1. if (A is NOT in ConflictReason)
        1.1.1. ConflictReason = ConflictReason U {A}
        1.1.2. if (ImplicationClause(A) != NULL)
            1.1.2.1. GetNonImpliedConflictClause(ImplicationClause(A)

Appendix A.3 - CRSAT Detailed Pseudo-Code

In this Appendix we'll provide a detailed pseudo-code of the CRSAT algorithm.
We'll use double underline to stress that the double-underlined text does not appear in the DPLL algorithm and is unique to CRSAT. The wave-underlined text ensures that the non-chronological backtracking principle from 4.1 is implemented.

Appendix A.3.1 - Algorithm's Organisation and Data Structures

First we'll describe the data structures used by CRSAT.

CRSAT uses generic data structure, which hold references to elements of a set. Reference to one element can't appear twice.
The following procedures are supposed to be provided:
ReferenceToElement : **FirstElemOfOrderedSet**(OrderedSet)
ReferenceToElement : **NextElemOfOrderedSet**(OrderedSet, ReferenceToElement)
void : **InsertToAnyPlaceInOrderedset**(OrderedSet; ElementOfOrderedSet)
ReferenceToElement : **LastElemInOrderedSet**(OrderedSet)

Boolean : **IfMemberOfOrderedSet**(OrderedSet; ReferenceToElement)

void : **RemoveFromOrderedSet**(OrderedSet; ReferenceToElement)

Each conflict will be represented as set of literals ordered by its decision level at the time when the conflict happened.

CRSAT algorithm uses the following data structures:

1. **IfAssigned -** defined the same as in A.1.
2. **AssignementSign -** defined the same as in A.1.
3. **ImplicationClause -** defined the same as in A.1.
4. **AssignementStatus :** *Variables->{NULL,Side1,Side2,,Implied,ConflictVar}.*

   ConflictVar means that variable is assigned a value in the process of conflicts checking. Other values have the same meaning as in DPLL algorithm

5. **ChkConflictsStatus :** ***Variables →{ NotKnown, NotChkConflicts, ChkMiddleConflicts, ChkLastConflict}***

   ChkConflictsStatus is valid only for 2nd side variables.

   ChkConflictsStatus(A)==

   NotKnown – A holds this value before deciding whether we are checking conflicts for A.

   NotChkConflicts – if we are not checking conflicts for A (proceed as in DPLL)

   ChkLastConflict - if we are checking the last conflict of A

   ChkMiddleConflicts – if we are checking any, but last conflict of A

6. ConflictsToCheck : Variables → Ordered Set of Conflicts

   For each decision variable A, ConflictsToCheck(A) will hold the set of conflicts recorded from the moment A was assigned a new value for the last time. In CRSAT algorithm below we don't make use of the order of conflicts in ConflictsToCheck(A), but it may be used for optimisation.

7. CurrentlyCheckedConf : Variables → Reference to Conflict

   For each 2nd side variable A, conflicts are checked for, CurrentlyCheckedConf(A) points to conflict currently being checked.

8. ConflictLitToAssign : Variables → Reference to Literal in Conflict

   For each 2nd side variable A, which conflicts are checked for, ConflictLitToAssign(A) points to a literal in CurrentlyCheckedConf(A), that should be assigned a value next.

9. ***AssignedVars* -** defined the same as in A.1.
10. ***ToBeImplied* -** defined the same as in A.1.

The algorithm uses the following functions.

1. The main function is CRSAT. It doesn't have parameters. It returns SAT, which stands for "the formula is satisfiable" or UNSAT, which stands for: "the formula is unsatisfiable".
2. The function **BCP** implements the BCP process. It's not brought below, since it's identical to one in A.1.
3. The function **PreAssign** – identical to A.1.
4. The function **Assign** – contains code from A.1 + it empties ConflictsToCheck for 1^{st} side variables.
5. The function **UnassignTopVar** – identical to A.1.
6. ChooseChkConflictStatus – receives a variable A, that was just 2^{nd} side decided and returns one of {ChkConflicts or NotChkConflicts}. If ChkConflicts is returned, conflicts will be checked for A, otherwise, the algorithm will proceed as in DPLL. This function will not be brought below, it can vary for different implementations. Anyway, the algorithm is correct for every possible ChooseChkConflictStatus.
7. GetFirstConflict – receives variable A, that was just 2^{nd} side decided and returns reference to a conflict, that should be checked first.
8. GetNextConflict - receives variable A, that was 2^{nd} side decided and returns reference to a conflict, that should be checked next (it supposes, that a call to GetFirstConflict was made before).
9. NotAssignedLitInCurrentConflict – receives 2^{nd} side variable A and returns reference to a literal from CurrentlyCheckedConf, that isn't assigned a value. If there is no such literal, it returns NULL.
10. GetNonImpliedConflictClause - this function returns a conflict clause corresponding to non-implied variables scheme. It's brought in A.2.
11. ManageNewConflict – this function receives a conflict clause, returned by GetNonImpliedConflictClause and keeps ConflictsToCheck valid.

12. MakeConflictSortedByDecisionLevel - this function receives a conflict clause and returns a conflict(ordered set of literals) corresponding to that clause, i.e. holding the same literals ordered by current decision level of corresponding variables. This function is called from ManageNewConflict and is not provided, since it's trivial.

Appendix A.3.2 - CRSAT Pseudo-Code

The CRSAT function extends the DPLL function from A.1. It also implements non-chronological backtracking principle from 4.1. Code unique to CRSAT will appear double-underlined. A line that ensures the non-chronological backtracking is wave-underlined. The other code is common to CRSAT and DPLL.

```
{SAT, UNSAT} : CRSAT()
1. ToBeImplied = {}
2. While (T)
   2.1. If (ToBeImplied == {})
      2.1.1.  A = Upper Variable From AssignedVars, s.t (AssignmentStatus(A)==Side1 OR AssignmentStatus(A)==Side2)
      2.1.2.  If ((A != NULL) AND (AssignmentStatus(A)==Side2) AND (ChkConflictsStatus(A) != NotChkConflcits))
         2.1.2.1. If (ChkConflictsStatus(A) == NotKnown)
            2.1.2.1.1.  Status = ChooseChkConflictStatus(A)
            2.1.2.1.2.  If (Status == NotChkConflicts)
               2.1.2.1.2.1. ChkConflictsStatus(A) = NotChkConflicts
               2.1.2.1.2.2. Go To 2.1.3
            2.1.2.1.3.  Else /*If (Status == ChkConflicts)*/
               2.1.2.1.3.1. ω = GetFirstConflict(A) /*In this function ChkConflictStatus(A) changes to
                                 ChkMiddleConflict or ChkLastConflict */
               2.1.2.1.3.2. B = NotAssignedVarInCurentConflict (A)
               2.1.2.1.3.3. If (B == NULL)
                  2.1.2.1.3.3.1.  Go To 3.1.3
               2.1.2.1.3.4. PreAssign(B, 0, ¬ω(B), ConflictVar)
               2.1.2.1.3.5. Go To 2.2
```

2.1.2.2. Else /*If (ChkConflictsStatus(A) != NotKnown)*/

2.1.2.2.1. B = NotAssignedVarInCurentConflict(A)

2.1.2.2.2. If (B != NULL)

2.1.2.2.2.1. PreAssign(B, 0, ¬ ConfChecked (B), ConflictVar)

2.1.2.2.2.2. Continue with loop 2

2.1.3. A = Any variable, s.t. IfAssigned(A)==F

2.1.4. PreAssign(A, 0, Any value from {T,F}, Side1)

2.2. ClauseWithContradiction = BCP()

2.3. if ((ClauseWithContradiction == NULL) AND (For All A : IfAssigned (A)==T))

2.3.1. Return SAT

2.4. if (ClauseWithContradiction != NULL)

2.4.1. ManageNewConflict(GetNonImpliedConflictClause(ClauseWithContradiction))

2.4.2. while ((Not Empty(AssignedVars)) AND
((AssignmentStatus(StackTop(AssignedVars)) == Side1))→(ConflictsToCheck(StackTop(AssignedVars))=={})) AND
((AssigmentStatus(StackTop(AssignedVars))==Side2)->(ChkConflictsStatus (StackTop
(AssignedVars)) != ChkMiddleConflict))

2.4.2.1. A = UnassignTopVariable()

2.4.3. If (IsEmpty(AssignedVars))

2.4.3.1. Return UNSAT

2.4.4. A = StackTop(AssignedVars)

2.4.5. If (AssignmentStatus(A) == Side1)

2.4.5.1. UnassignTopVariable()

2.4.5.2. PreAssign(A; 0; ¬AssignmentSign(A); Side2)

2.4.5.3. Continue with loop 2

2.4.6. If (AssignmentStatus(A) == Side2) /*Note : ChkConflictsStatus (A) == ChkMiddleConflict*/

2.4.6.1. ω = GetNextConflict(A) /*Here ChkConflictStatus(A) may change to ChkLastConflict*/

2.4.6.2. B = NotAssignedLitInCurrConflict(A)

2.4.6.3. If (B != NULL)

2.4.6.3.1. PreAssign(B, 0, ¬ω(B), ConflictVar)

2.4.6.4. Continue with loop 2

void Assign(Variable A)

1. IfAssigned(A) = T
2. Push(AssignedVars, A)
3. If (AssignementStatus(A)==Side1)
    3.1. ConflictsToCheck(A) = {}

void : ManageNewConflict(ConflictClause CC)

1. ω = MakeConflictSortedByDecisionLevel(CC)
2. For (A = FirstElemOfOrderedSet (ω); A != NULL; A = NextElemOfOrderedSet (ω, A))
    2.1. If (AssignmentStatus(A)==Side1)
        2.1.1. InsertToAnyPlaceInOrderedSet(ConflictsToCheck(A), ω)

ReferenceToConflict : GetFirstConflict(Variable A)

1. ω = CurrentlyCheckedConf(A) = FirstElemOfOrderedSet(ConflictsToCheck(A))
2. If (ω == LastElemOfOrderedSet(ConflictsToCheck(A))
    2.1. ChkConflictStatus(A) = ChkLastConflict
3. Else
    3.1. ChkConflictsStatus(A) = ChkMiddleConflict
4. ConflictLitToAssign(A) = NULL
5. Return ω

ReferenceToConflict : GetNextConflict(Variable A)

1. ω = CurrentlyCheckedConf (A) = NextElemOfOrderedSet(ConflictsToCheck(A), CurrentlyCheckedConf(A))
2. If (ω == LastElemOfOrderedSet(ConflictsToCheck(A))
    2.1. ChkConflictsStatus(A) = ChkLastConflict
3. ConflictLitToAssign(A) = NULL
4. Return ω

ReferenceToLiteral : NotAssignedLitInCurrentConflict(Variable A)

1. ω = CurrentlyCheckedConf(A)
2. If (ConflictLitToAssign(A) == NULL)

2.1. ConflictLitToAssign(A) = NextElemOfOrderedSet(ω, A)

3. While ((ConflictLitToAssign(A) != NULL) AND (IfAssigned(ConflictLitToAssign(A))==T))

3.1. ConflictLitToAssign(A) = NextElemOfOrderedSet(ω, ConflictLitToAssign(A))

4. Return ConflictLitToAssign(A)

Appendix A.4 - CRSAT Formal Correctness Proof

In this Appendix we'll bring a formal correctness proof of theorems 4.2.4.1 and 4.2.4.2. We remind that theorem 4.2.4.2 states that CRSAT is complete and sound. In the proof we'll address the CRSAT code from appendix A.3.
First we'll proof an auxiliary lemma.

Lemma A.4.1: Let Σ be the CNF formula being checked by CRSAT algorithm. Let $\psi_1 \ldots \psi_n$ be some conflict clauses Then $\Sigma \rightarrow \psi_1 \Lambda \ldots \Lambda \psi_n$.

Proof of A.4.1:

The process of non-implied conflict clauses recording in CRSAT is identical to that introduced in 3.2.1.2.3 and brought in Appendix A.2. Therefore for each i : $\Sigma \rightarrow \psi_i$, otherwise recording clauses according to non-implied scheme would have changed the propositional logic function, we are working with. According to basic propositional logic theory $\Sigma \rightarrow \psi_1 \Lambda \ldots \Lambda \psi_n$.

■ (lemma A.4.1)

Theorem 4.2.4.1: Let Σ be the CNF formula being checked by CRSAT algorithm. Let B be a decision variable (we'll suppose without restriction of generality that B is assigned to T when 1st side assigned). Let Ψ_1 be the conjunction of (non-implied variables scheme) conflict clauses corresponding to all the conflicts that happened while B was 1st side assigned. Let Φ be the conjunction of all conflict clauses corresponding to all the conflicts that happened while B was 1st side and 2nd side assigned. Let A_1(lowest decision level)…A_m be assigned non-implied literals, just before B 1st side assignment. Let $\bar{A} = A_1 \Lambda \ldots \Lambda A_m$. Then :

a. If $\neg(\Sigma\Lambda\bar{A}\Lambda B\rightarrow\Box)$ (semantically, there is a model to Σ with $A_{1...}A_m$=T; B=T) then **:**

a.1. The algorithm will return SAT and won't unassign B.

b. If $(\Sigma\Lambda\bar{A}\Lambda B\rightarrow\Box)$ (semantically, there is no model to Σ with $A_{1...}A_m$=T; B=T) then :

b.1. The algorithm will unassign B after 1st side assignment.

b.2. Just after B's 1st side unassignment the following expression is true: $\Psi_1\Lambda\bar{A}\Lambda B\rightarrow\Box$ (Note that $\neg\Psi_1 v\neg\bar{A} v\neg B\Leftrightarrow T$ is identical to the previous formula, and may be proofed instead)

b.3. If $\neg(\Sigma\Lambda\bar{A}\Lambda\neg B\rightarrow\Box)$ (there is a model to Σ with $A_{1...}A_m$=T; B=F) then

b.3.1. The algorithm will assign B its second value (2nd side assignment). It will return SAT and won't unassign B.

b.4. If $(\Sigma\Lambda\bar{A}\Lambda\neg B\rightarrow\Box)$ (there is no model to Σ with $A_{1...}A_m$=T; B=F)

b.4.1. $\Phi\Lambda\bar{A}\rightarrow\Box$ (Note that $\neg\Phi v\neg\bar{A}\Leftrightarrow T$ is identical to the previous formula, so it may be proofed instead)

b.4.2. If B is assigned 2nd side value, the algorithm will unassign B at some stage.

Proof of theorem 4.2.4.1:

First a few general remarks:

1) Throughout the proof, it would be convenient to denote the conjunction of conflict clauses recorded while B was 2nd side assigned by Ψ_2(it may be empty, if non-chronological backtracking is applied for B). Observe that $\Phi=\Psi_1\Lambda\Psi_2$.
2) Ψ_1 in the theorem's condition and ConflictsToCheck(B) ordered set in the algorithm from Appendix A.3 are different form of referencing the same information.
3) When we refer to condition/statement which first part of it is number(like 2.4.3), we refer to condition/statement appearing at that line in CRSAT pseudo-code. When we refer to condition or statement which first part of it is lower-case letter (like b.4.1), we refer to condition/statement appearing at that line in theorem's 4.2.4.1 condition.

The proof is by reverse induction on decision level of B (denote it *dl*).

Base: *dl*=n(n is the number of variables).

1. If **$\neg(\Sigma\wedge\bar{A}\wedge B\rightarrow\square)$(**there is a model to Σ with $A_1\ldots A_m$=T; B=T), then after B pre-assigning to T (line 2.1.4), the algorithm will execute BCP (line 2.2), the condition of 2.3 will be T since there can't be contradiction in this branch according to our condition, so ClauseWithContradiction==NULL and also all the variables are assigned, since decision level of B is equal to the number of variables, so it's the last variable assigned. Therefore the algorithm will return SAT and won't unassign B and **condition a.1 of the theorem is proven**.
2. If **$(\Sigma\wedge\bar{A}\wedge B\rightarrow\square)$ (**there is no model to Σ with $A_{1\ldots}A_m$=T; B=T), than there will be a contradiction after B assignment to T, therefore the condition 2.3 becomes F, but 2.4 becomes T. Observe, that ConflictsToCheck(B) is not empty, since B is the last non implied variable on the stack and therefore it participated in the only conflict, which happened when it was 1st side assigned. Therefore, the "while" loop of 2.4.2 condition won't be true for B and unassignment of B will happen at line 2.4.5.1 so **condition b.1 of the theorem is proven**. Now we'll prove condition B.2 ($\neg\Psi_1 v\neg\bar{A}\ v\neg B\Leftrightarrow T$). Observe that there is only one clause in Ψ_1. It's added at line 2.4.1. Therefore, according to the structure of GetNonImpliedConflictClause function : $\Psi_1\equiv\ \neg A_{i(1)}v\ldots v\neg A_{i(k)}v\neg B$, where $1\geq i(j)\leq m$. Therefore : $\neg\Psi_1 v\neg\bar{A}v\neg B\ =\ ((A_{i(1)}\wedge\ldots\wedge A_{i(k)}\wedge B)v(\neg\bar{A}v\neg B))$.
   Observe than if all of $A_1\ldots A_m$,B are T, then $(A_{i(1)}\wedge\ldots\wedge A_{i(k)}\wedge B)$ is T; otherwise $(\neg\bar{A}v\neg B)$ is T. Therefore
   $((A_{i(1)}\wedge\ldots\wedge A_{i(k)}\wedge B)v(\neg\bar{A}v\neg B))\Leftrightarrow T\ \rightarrow$
   $\neg\Psi_1 v\neg\bar{A}\ v\neg B\Leftrightarrow T$ and **b.2 is proven**.
   Now, suppose that **$\neg(\Sigma\wedge\bar{A}\wedge\neg B\rightarrow\square)$** (there is a model to Σ with $A_{1\ldots}A_m$=T; B=F). B's 2nd side assignment will happen in the next iteration of 2 during BCP called at line 2.2, then the algorithm will return SAT at line 2.3.1 without B unassigning and **b.3.1 is proven**.

Finally, suppose that **($\Sigma \Lambda \bar{A} \Lambda \neg B \rightarrow \square$)** (there is no model to Σ with $A_{1...}A_m$=T; B=F). There will be a contradiction after B 2nd side assignment. Observe that the condition of line 2.4.2 won't become true, since ChkConflictsStatus(B) == NotChkConflicts(was set at line 2.1.2.1.2.1) or ChkLastConflict(was set in GetFirstConflict, called at line 2.1.2.1.3.1). Therefore, B will be unassigned at line 2.4.5.1, so **condition b.4.2 of the theorem is proven**.

Let's now prove b.4.1($\neg \Psi_1 \vee \neg \Psi_2 \vee \neg \bar{A} \Leftrightarrow T$). Observe that there will be one clause in Ψ_2 : $\Psi_2 \equiv \neg A_{h(1)} \vee \ldots \vee \neg A_{h(1)} \vee B$, where $1 \geq h(j) \leq m$.

Therefore : $\neg \Phi \vee \neg \bar{A} = \neg \Psi_1 \vee \neg \Psi_2 \vee \neg \bar{A} =$

$(A_{i(1)} \Lambda \ldots \Lambda A_{i(k)} \Lambda B) \vee (A_{h(1)} \Lambda \ldots \Lambda A_{h(1)} \Lambda \neg B) \vee (\neg \bar{A})$.

If one of A_i is F, than the third part of the above expression is T.

If for all i : A_i=T, than

If B=T, than the first part of the above expression is T,

Otherwise the second part of the above expression is T.

Therefore $\neg \Phi \vee \neg \bar{A} \Leftrightarrow T$ and **b.4.1 is proven**.

■ (base of induction)

Step of induction: we need to prove the theorem supposing that B is of decision level *dl* and that the theorem is right for all the variables with decision level >*dl*.

1. If **$\neg(\Sigma \Lambda \bar{A} \Lambda B \rightarrow \square)$(**there is a model to Σ with $A_1 \ldots A_m$=T; B=T), then there obviously will be no contradiction after the BCP(line 2.2). It can turn out to be, that after the BCP condition 2.3 becomes true and obviously A.1 is true than. Otherwise next decision variable is chosen (line 2.1.3). We'll denote it by C and suppose without restriction of generality it's assigned T first. Obviously the decision level of C is greater than one of B. Also the model with $A_1 \ldots A_m$=T; B=T must contain C=T or C=F. Therefore according to the induction assumption, after C assignment the algorithm will return SAT either :
   - according to condition a.1, if there is a model containing C=T, or
   - according to condition b.3.1, if there is no model containing C=T, but there is one containing C=F

Therefore, the algorithm will return SAT and will not unassign B and the **condition a.1 is proven.**

2. If **$(\Sigma\Lambda\bar{A}\Lambda B\rightarrow\square)$** (there is no model to Σ with $A_{1\ldots}A_m$=T; B=T) then one of two following things can happen. We'll prove b.1 and b.2 for each :

I) *There is a conflict straight after the BCP procedure after B 1st side assignment.* In this case it's straightforward from the algorithm structure, that **b.1 is true for case I**.

Let's prove b.2. For same reasons like in the induction base proof: $\Psi_1 \equiv \neg A_{i(1)} v \ldots v \neg A_{i(k)} v \neg B$, where $1 \geq i(j) \leq m$.

$\neg\Psi_1 v \neg\bar{A} v \neg B = ((A_{i(1)} \Lambda \ldots \Lambda A_{i(k)} \Lambda B) v (\neg\bar{A} v \neg B))$.

Observe than if all of $A_1 \ldots A_m$,B are T, then $(A_{i(1)}\Lambda\ldots\Lambda A_{i(k)}\Lambda B)$ is T; otherwise $(\neg\bar{A} v \neg B)$ is T. Therefore

$(((A_{i(1)} \Lambda \ldots \Lambda A_{i(k)} \Lambda B) v (\neg\bar{A} v \neg B)) \Leftrightarrow T) \rightarrow (\neg\Psi_1 v \neg\bar{A} \; v \neg B \Leftrightarrow T)$ and **b.2 is true for case I**.

II) *There is a no conflict straight after the BCP procedure after B 1st side assignment and next decision variable is chosen.*

Let C be the variable chosen next. Decision level of C is greater than *dl*. Therefore we can use the induction assumption for C. Observe, that there is no model containing $A_{1\ldots}A_m$=T; B=T, therefore there are no models containing ($A_{1\ldots}A_m$=T; B=T; C=T) or ($A_{1\ldots}A_m$=T; B=T; C=F). So we can use the conclusions of condition b.4 of the theorem (for variable C).

According to b.4.2, C will be unassigned after its 2nd side assignment. Line 2.4.2.1 in the algorithm is the only line where 2nd side variable may be unassigned. After that line the top variable on the AssignedVars stack is be B (1st side assigned), therefore, it will be unassigned at line 2.4.2.1(if ConflictsToCheck(B)=={}) or 2.4.5.1. Therefore **b.1 is true for case II**.

Let Γ be the set of clauses recorded while C was assigned. Observe, that the set of non-implied literals being on the stack just before C 1st side assignment includes $A_1 \ldots A_m$ and B. Therefore, according to b.4.1 : $\Gamma\Lambda\bar{A}\Lambda B\rightarrow\square$. Observe now, that Ψ_1(set of clauses recorded while B is 1st side assigned) is exactly Γ. Therefore $\Psi_1\Lambda\bar{A}\Lambda B\rightarrow\square$ and **b.2 is true for case II**.

Up to this point we have proven a.1, b.1 and b.2.

Let's suppose now that b.3 is true (**$\neg(\Sigma\Lambda\bar{A}\Lambda\neg B\rightarrow\Box)$**) (there is a model to Σ with $A_{1\ldots}A_m$=T; B=F)) and prove b.3.1. We'll denote the model by M.

First we'll prove that non-chronological backtracking doesn't take place in our case. For proving it, we'll prove that ConflictsToCheck(B) cannot be empty (in terms of our proof Ψ_1 contains $\neg$B). According to lemma A.4.1, $\Sigma\rightarrow\Psi_1$. Therefore M is Ψ_1's model. Let M' be an interpretation which is different from M by <u>only</u> B value (therefore M' contains A_i=T;B=T). We have proven $\Psi_1\Lambda\bar{A}\Lambda B\rightarrow\Box$. For this formula to be true, M' must not satisfy Ψ_1. Therefore Ψ_1 must contain clauses that are satisfied by M and not satisfied by M'. Obviously such clauses contain $\neg$B. Therefore Ψ_1 contains $\neg$B and ConflictsToCheck(B) can't be empty.

It means, that the non-chronological backtracking won't happen in our case, or in another words, condition 2.4.2 won't become true for B, and B will be 2nd side assigned, therefore first part of b.3.1 statement it true.

Now we'll complete b.3.1 proof. We'll separate between 2 cases according to the return value of ChooseChkConflictStatus(line 2.1.2.1.1) for B or in another words, case (a) is the case if we don't check conflicts for B, but proceed as in DPLL algorithm, case (b) is the more interesting case, when we choose to check conflicts for B.

(a) Conflicts aren't checked for B(ChooseChkConflictStatus returns NotChkConflicts at line 2.1.2.1.1)

The proof here is very similar to a.1 proof above. After B 2nd side assignment BCP is executed(line 2.2). It may happen that after the BCP, 2.3 becomes true and obviously, b.3.1 is true in that case. Otherwise next decision variable is chosen (line 2.1.3). We'll denote it by C and without restriction of generality suppose it's assigned T first. Obviously the decision level of C is greater than one of B. Also the model with $A_1\ldots A_m$=T; B=F must contain C=T or C=F. Therefore according to the induction assumption, after C assignment the algorithm will return SAT either :

- according to condition a.1, if there is model containing C=T, or

- according to condition b.3.1, if there is no model containing C=T, but there is one containing C=F

Therefore, the algorithm will return SAT and will not unassign B and the **condition b.3.1 is proven for case (a).**

(b)Conflicts are checked for B(ChooseChkConflictStatus returns ChkConflicts at line 2.1.2.1.1)

Let us look more closely at the structure of Ψ_1. We'll denote its clauses by $\psi_1 \ldots \psi_k$ It's composed from two parts: clauses containing ¬B (we'll denote each such clause by ω_i and a conjunction of it by $\Omega=\omega_1 \Lambda \ldots \Lambda \omega_w$) and clauses not containing not B nor ¬B (we'll denote each such clause by χ_i and a conjunction of it by $X=\chi_1 \Lambda \ldots \Lambda \chi_x$). So $\Psi_1=\Omega \Lambda X$. Ψ_1 can't contain clauses with B, since all the clauses are added when B is assigned to T.

Observe, that after B 2nd side assignment, the algorithm for each ω_i assigns one by one **not yet assigned** variables from ω_i to value opposite to that of appropriate literal from ω_i(single pre-assignment happens at line 2.1.2.1.3.4), than (if there was no contradiction nor the Σ proved to be SAT) it chooses next decision variable. After checking ω_i, it backtracks and checks next conflict.

Let M be a model of Σ, s.t. it contains A_i=T;B=F. It exists since condition b.3 is true for our case. Let $C_1 \ldots C_k$ be the literals of <u>all</u> variables which are:

1) not assigned a value till the point after the BCP procedure for ¬B assignment.
2) consistent with M

Let M' be an interpretation which is different from M by <u>only</u> B value (therefore M' contains A_i=T;B=T). In order that M' would satisfy $\neg \psi_1 \vee \ldots \vee \neg \psi_k \vee \neg \bar{A} \vee \neg B$ (which is a tautology according to b.2), it must falsify one of ψ_i's. We'll denote one of such clauses by ω. Note that ω must be a member of Ω (it must contain ¬B),

because otherwise M would falsify ω, but that couldn't happen, since M is Σ's model and $\Sigma \rightarrow \omega$, since ω is a conflict clause. Hence,

$\omega = \neg A_{i(1)} \vee \ldots \vee \neg A_{i(k)} \vee \neg B \vee \neg C_{j(1)} \vee \ldots \vee \neg C_{j(h)}$, where for each f :

$1 \le j(f) \ge k;\ 1 \le i(f) \ge m.$

According to the algorithm, ω is being checked as any of the members of Ω. After all the members of ω are assigned, the following variables are on the decision stack $A_1..A_m,\neg B,C_{i(1),\ldots,}C_{i(h)}$. Observe, that there can't be contradiction after assigning one of C_i's, since the decision stack literals and C_i's are consistent with M. Now, let D be a decision variable assigned after assigning all ω members, we'll suppose without restriction of generality it's assigned T first. Obviously the decision level of D is greater than one of B. Also M contains D=T or D=F. Therefore according to the induction assumption, after D assignment the algorithm will return SAT either:

- according to condition a.1, if M contains D=T, or
- according to condition b.3.1, if M contains D=F.

Therefore, the algorithm will return SAT and will not unassign B and the **condition b.3.1 is proven for case (b).**

Up to this point we have proven a.1, b.1, b.2 and b.3.1. Let's prove b.4.1 and b.4.2.

Suppose that $(\Sigma \wedge \bar{A} \wedge \neg B \rightarrow \square)$ (there is no model to Σ with $A_{1\ldots}A_m$=T; B=F).

First we'll prove it for the case when ConflictsToCheck(B) is empty (Ψ_1 doesn't contain ¬B). Observe that Ψ_1 doesn't contain B as well according to GetNonImpliedConflictClause structure.

In this case b.4.2 is trivially true, since B isn't assigned 2nd side value according to the algorithm (2.4.2 is true when B is at the top of AssignedVars).

We'll show now that $\Psi_1 \wedge \bar{A} \rightarrow \square$. Suppose on the contrary, M is a model for $\Psi_1 \wedge \bar{A}$. If M contains B=T, it satisfies $\Psi_1 \wedge \bar{A} \wedge B$, in contradiction to b.2. If M contains B=F, let M' be an interpretation which is different from M by only B value. M' must satisfy $\Psi_1 \wedge \bar{A}$, since this expression doesn't contain B or ¬B. Therefore M' must satisfy

$\Psi_1 \Lambda \bar{A} \Lambda B$, in contradiction to b.2. So we have shown that $\Psi_1 \Lambda \bar{A} \rightarrow \square$. Observe now, that $\Phi \equiv \Psi_1$, since we don't check 2nd side of B, therefore $\Phi \Lambda \bar{A} \rightarrow \square$ and by this we have proven b.4.1 for the case when ConflictsToCheck(B)=={}.

Suppose now ConflictsToCheck(B) isn't empty (Ψ_1 contains $\neg B$). According to algorithm's structure B will be assigned 2nd side value in this case.

We'll separate between 2 cases: (a) conflicts aren't checked for B; (b) conflicts are checked for B. We'll prove b.4.1 and b.4.2 for each.

(a) Conflicts aren't checked for B(ChooseChkConflictStatus returns NotChkConflicts at line 2.1.2.1.1)

We'll proceed here as usual. There can be two cases:

[I] *There is a conflict straight after the BCP procedure after B 2nd side assignment.*

b.4.2 is true for case [I], according to the algorithm structure(this would happen at line 2.4.2.1). Let's prove b.4.1.

First, observe that there will be one clause in Ψ_2 : $\Psi_2 \equiv \neg A_{h(1)} v \ldots v \neg A_{h(1)} v B$, where $1 \geq h(j) \leq m$.

We'll show that there is no interpretation falsifying $\neg \Phi v \neg \bar{A} \equiv \neg \Psi_1 v \neg \Psi_2 v \neg \bar{A}$, by trying to build such an interpretation and seeing , that it's impossible.

Observe, that such an interpretation, if existed, would have contained $A_1 \ldots A_m$=T (to make $\neg \bar{A}$ false). Therefore, B must be T, otherwise Ψ_2 would have been F and $\neg \Psi_1 v \neg \Psi_2 v \neg \bar{A}$ would have been T.

But in this case, according to already proofed b.2 ($\neg \Psi_1 v \neg \bar{A} v \neg B \Leftrightarrow T$) $\neg \Psi_1$ must be T, but then $\neg \Psi_1 v \neg \Psi_2 v \neg \bar{A}$ is T and we have failed to build an interpretation falsifying our formula.

Therefore **b.4.1 is proven for case [I]**.

[II] *There is no conflict straight after the BCP procedure after B assignment and next decision variable is chosen.*

Let C be the variable chosen next. Decision level of C is greater than *dl*. Therefore we can use the induction assumption for C. Observe, that condition b.4 of the theorem is true for C and if C was 2nd side assigned then according to b.4.2, C will be unassigned after its 2nd side assignment. Line 2.4.2.1 in the algorithm is the only line where 2nd side variable may be unassigned. If C wasn't 2nd side assigned, it still means that it has been unassigned a value (1st side in this case) at line 2.4.2.1.

After that line the top variable on the AssignedVars stack is B(2nd side assigned), Condition 2.4.2 is true for B and therefore, it will be unassigned at line 2.4.2.1.

Therefore **b.4.2 is true for case [II]**.

Let Γ be the set of clauses recorded while C assigned a value. Observe, that the set of literals being on the stack just before C assignment includes $A_1 \ldots A_m$ and $\neg B$. Therefore, according to b.4.1 : $\neg\Gamma \vee \neg\bar{A} \vee B \Leftrightarrow T$. Observe now, that Ψ_2(set of clauses recorded while B is 2nd side assigned) is exactly Γ. Therefore $\neg\Psi_2 \vee \neg\bar{A} \vee B \Leftrightarrow T$. According to already proven b.2 : $\neg\Psi_1 \vee \neg\bar{A} \vee \neg B \Leftrightarrow T$. Observe that $\neg\Psi_1 \vee \neg\Psi_2 \vee \neg\bar{A}$ is a resolvent of left sides of two previous formulas, therefore it's also a tautology and **b.4.1 is true for case [II]** (it can be easily shown semantically as well).

By this, we have completed b.4.1 and b.4.2 proof for (a).

(b) *Conflicts are checked for B(ChooseChkConflictStatus returns ChkConflicts at line 2.1.2.1.1)*

If there is a contradiction straight after BCP for $\neg B$, the proof is identical to that of case (a). Suppose that there is no such contradiction.

We'll extend the notations of b.3.1 proof. As you may remember $\omega_i (1 \le i \ge w)$ are the clauses of Ψ_1, containing $\neg B$. Each ω_i has the following structure $\omega_i = \neg A_{i(1)} \vee \ldots \vee \neg A_{i(k)} \vee \neg B \vee \neg C_1 \vee \ldots \vee \neg C_p$, where A_i's, B and C_i's are mutually disjoint. We'll denote for each i : $\alpha_i = A_{i(1)} \wedge \ldots \wedge A_{i(k)};\ \gamma_i = C_1 \wedge \ldots \wedge C_p$.

According to this notation, for each i : $\omega_i = \neg\alpha_i \vee \neg B \vee \neg\gamma_i$.

First we'll prove that b.4.1 is true. Observe, that if we choose to check conflicts for B, the following happens: the algorithm reaches line 2.1.2.1.3.1 and chooses the first conflict to check. In another words, it chooses one of Ω's. Let it be ω_1. Then(line 2.1.2.1.3.2) it chooses a literal from γ_1, since literals from α_i and B are all assigned a value. Let this literal be C_1. It decides $\neg C_1$. Observe, that according to the algorithm's structure, it continues to choose variables from γ_1 and to assign it values till one of 2 things happen:

- There can be a conflict as a result of BCP after assigning one of γ_i's variables. In this case, the algorithm will unassign each of assigned γ_1 variables at line 2.4.2.1.

- There is no conflict while assigning γ_i's variables. In this case we must choose next decision variable D. According to the induction statement, D will be unassigned after 2^{nd} side assignment or unassigned after 1^{st} side assignment and not assigned 2^{nd} side at all, since there is no model to Σ, containing part of literals below D on the decision stack (we supposed, that no model contains $A_1 \ldots A_m$=T; B=F). Obviously there is no model, containing all the literals below D. The unassignment happens at line 2.4.2.1. Afterwards the algorithm will unassign each of assigned γ_1 variables.

Now we'll trace what happens next in both cases: if ω_1 is the only clause in Ω, the algorithm unassigns B(ChkConflictStatus(B)==ChkLastConflict in this case) and b.4.2 is met, otherwise it chooses next conflict(line 2.4.6.1) ω_2 , assigns a literal from γ_2 and goes on exactly like in ω_1 checking case. It does so for every ω_i, till it reaches the last conflict to check after which it unassigns B.

It obvious from the reasoning above that the algorithm will unassign B after 2^{nd} side assignment after checking every one of Ω members and by this **we have proven b.4.2 for case (b).**

Now we'll prove b.4.1.

Let Π_i(for each i) be the conjunction of clauses which is recorded while ω_i is being checked (while γ_i's variables are on the decision stack).

Let's prove the following statement:

Lemma A.4.2: For each i : $\Pi_i \Lambda \bar{A} \Lambda \neg B \Lambda \gamma_i \rightarrow \square$.

Proof of A.4.2:

There can be 2 cases while checking each of ω_i :

- There can be a conflict as a result of BCP after assigning one of γ_i variables. In this case Π_i consist of a single clause and has following form : $\Pi_i = (\neg A_{j(1)} v \ldots v \neg A_{j(l)})^{[1]} v (B)^{[2]} v (\neg C_{k(1)} v \ldots v \neg C_{k(m)})^{[3]}$, where [3] must have at least one member, [1] may have 0 or more(till m) members(i.e. may not appear at all) and [2] may not appear in Π_i. In any case, observe, that $\Pi_i \Lambda \bar{A} \Lambda \neg B \Lambda \gamma_i \rightarrow \square$ is a tautology. That's true, since for Π_i being true at least one of [1],[2],[3] must be true. If [1] is true, than $\bar{A}$ is false. If [2] is true, than $\neg B$ is false and if [3] is true then γ_i is false. In any case $\Pi_i \Lambda \bar{A} \Lambda \neg B \Lambda \gamma_i$ is a contradiction.
- There is no conflict while assigning γ_i's variables. In this case we must choose next decision variable D. Let Γ be the set of conflict clauses recorded while D is assigned. According to the induction assumption and condition b.4.1, $\Gamma \Lambda \bar{A} \Lambda \neg B \Lambda \gamma_i \rightarrow \square$ (observe that A_i's, $\neg B$ and C_i's are the non-implied variables below D on the decision stack while D is being assigned). But Π_i is exactly Γ, therefore $\Pi_i \Lambda \bar{A} \Lambda \neg B \Lambda \gamma_i \rightarrow \square$.

■ (lemma A.4.2)

Now we'll prove another lemma:

Lemma A.4.3: $\bar{A} \Lambda X \Lambda \Pi_i \Lambda (\neg \alpha_i v \neg B v \neg \gamma_i) \rightarrow \bar{A} \Lambda X \Lambda (\neg \alpha_i v \neg \gamma_i)$ (reminder: X is the conjunction of such Ψ_1 clauses, that don't contain not B nor $\neg B$)

Proof of lemma A.4.3:

We must prove, that if $\bar{A} \Lambda \Pi_i \Lambda X \Lambda (\neg \alpha_i v \neg B v \neg \gamma_i)^{[1]}$ is T, than $\bar{A} \Lambda X \Lambda (\neg \alpha_i v \neg \gamma_i)^{[2]}$ is T. If [1] is T than $\bar{A}$=T, Π_i=T and X=T. From the fact that $\bar{A}$=T and the structure of α_i, it follows α_i=T.

Therefore for [1] to be T it must be that B=F or γ_i=F. If γ_i=F than [2] is obviously T. Let us suppose that B=F. According to lemma A.4.2 , if Π_i=T and $\bar{A}$=T and B=F, it must be that γ_i=F, therefore [2] is T in this case as well.

■ (lemma A.4.3)

To another lemma:

Lemma A.4.4: $\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1) \wedge \ldots \wedge (\neg\alpha_w \vee \neg\gamma_w) \rightarrow \Box$

Proof of lemma A.4.4:

According to b.2 : $\Psi_1 \wedge \bar{A} \wedge B \rightarrow \Box$ holds. It can be rewritten as

$\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1 \vee \neg B) \wedge \ldots \wedge (\neg\alpha_w \vee \neg\gamma_w \vee \neg B) \wedge B \rightarrow \Box$.

Observe, that $\bar{A}$,X,α_i's and γ_i's don't contain not B nor ¬B.

Therefore, $\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1) \wedge \ldots \wedge (\neg\alpha_w \vee \neg\gamma_w)^{[*]}$ must be F under all interpretations, otherwise, if we had set B to T in the interpretation which satisfied [*], $\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1 \vee \neg B) \wedge \ldots \wedge (\neg\alpha_w \vee \neg\gamma_w \vee \neg B) \wedge B$ would have been T and this would have contradicted b.2.

So we have proven, that [*] is false under all interpretations, therefore [*]→□ therefore $\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1) \wedge \ldots \wedge (\neg\alpha_w \vee \neg\gamma_w) \rightarrow \Box$.

■ (lemma A.4.4)

Now, we'll prove b.4.1 for our case (b). By a simple regrouping we get:

$\bar{A} \wedge \Phi \Leftrightarrow \bar{A} \wedge \Psi_1 \wedge \Psi_2 \Leftrightarrow$

$\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg B \vee \neg\gamma_1) \wedge \ldots \wedge (\neg\alpha_w \vee \neg B \vee \neg\gamma_w) \wedge \Pi_1 \wedge \ldots \wedge \Pi_w \Leftrightarrow$

$(\bar{A} \wedge X \wedge \Pi_1 \neg\alpha_1 \vee \neg B \vee \neg\gamma_1) \wedge \ldots \wedge (\bar{A} \wedge X \wedge \Pi_w \neg\alpha_w \vee \neg B \vee \neg\gamma_w)$.

According to lemma A.4.3, for each i :

$(\bar{A} \wedge X \wedge \Pi_i \wedge (\neg\alpha_i \vee \neg B \vee \neg\gamma_i)) \rightarrow \bar{A} \wedge X \wedge (\neg\alpha_i \vee \neg\gamma_i)$, therefore

$\bar{A} \wedge \Psi_1 \wedge \Psi_2 \rightarrow (\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1)) \wedge \ldots \wedge (\bar{A} \wedge X \wedge (\neg\alpha_w \vee \neg\gamma_w))$.

But by regrouping, we get $(\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1)) \wedge \ldots \wedge (\bar{A} \wedge X \wedge (\neg\alpha_w \vee \neg\gamma_w)) =$

$\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1) \wedge \ldots \wedge (\neg\alpha_w \vee \neg\gamma_w)$, and according to lemma A.4.4 :

$\bar{A} \wedge X \wedge (\neg\alpha_1 \vee \neg\gamma_1) \wedge \ldots \wedge (\neg\alpha_w \vee \neg\gamma_w) \rightarrow \Box$.

Therefore $\bar{A} \wedge \Psi_1 \wedge \Psi_2 \rightarrow \Box$ (obvioulsy, if $F_1 \rightarrow F_2$ and $F_2 \rightarrow F_3$ than $F_1 \rightarrow F_3$).

So **we have proven b.4.1 for case (b)**.

But this we have completed theorem's 4.2.4.1 proof.

■ (theorem 4.2.4.1)

Next theorem proves the soundness and completeness of CRSAT :

Theorem 4.2.4.2 (soundness and completeness of CRSAT):

Let Σ be a CNF formula. Than :

1) If Σ is SAT, CRSAT(Σ) returns SAT
2) If Σ is UNSAT, CRSAT(Σ) returns UNSAT

Proof of 4.2.4.2:

1. Suppose Σ is SAT. Let B be the first decision variable chosen at line 2.1.3, suppose without restriction of generality, it was assigned T first. According to our assumption there is a model M to Σ. If B=T according to M, than according to condition a.1 of theorem 4.2.4.1, the algorithm will return SAT (observe, that in our case m=0 in condition a of 4.2.4.1). If B=F according to M, than according to condition b.3.1 of theorem 4.2.4.1, the algorithm will return SAT.
2. Suppose Σ is UNSAT. Let B be the first decision variable chosen at line 2.1.3, suppose without restriction of generality, it was assigned T first. According to our assumption, there is no model to Σ with either B=T or B=F, therefore conditions b and b.4 of theorem 4.2.4.1 are true and if B was 2^{nd} side assigned, then according to 4.2.4, the algorithm unassigns B after 2^{nd} side assignment. If B was only 1^{st} side assigned, the algorithm unassigns it according to b.1. Anyway, the unassignment happens at line 2.4.2.1. Obviously, AssignedVars stack is empty after unassigning B, since it's been the bottom-most variable in the stack. Therefore the condition 2.4.3 becomes true and CRSAT will return UNSAT at line 2.4.3.1.

■ (theorem 4.2.4.2)

Appendix A.5 - New Pruning Techniques Related Code Changes and Formal Proofs

The code changes to CRSAT code from Appendix A.3 needed to implement each one of the new pruning techniques from section 4.3 will be provided below. Formal correctness proofs for each technique will be provided as well.

The code changes will be based on code which includes changes from previous techniques.
The proofs will be based on theorem 4.2.4.1 proof brought in Appendix A.4.

In next subsection we'll introduce an auxiliary lemma, which will help us in formal proofs in further subsections.

Appendix A.5.1 - An Auxiliary Lemma

Lemma A.5.1: Let ψ and α be propositional logic formulas that are using disjoint sets of variables. Then the following is true :
If 1) $\psi \Lambda \alpha \rightarrow \square$ and 2) there is a model M to α than $\psi \rightarrow \square$.

Proof of lemma A.5.1:
Suppose that there is a model M' for ψ. Than, we could set all the variables in M' that belong to α to the values they get according to M. M' is still a model for ψ, since ψ and α are disjoint, but it's also a model for α since all the variables participating in α are set to the same values as in M. Therefore M' is a model for $\psi \Lambda \alpha$ and we get a contradiction with the assumption $\psi \Lambda \alpha \rightarrow \square$.

■ (A.5.1)

Appendix A.5.2 - Using 2nd Side Variable Non-Participation in Conflicts

A.5.2.1 - Code Changes

There is a need to record for each decision assignment of a variable not only conflict clauses recorded when the variable is 1st side assigned (currently such clauses are recorded in ConflictsToCheck), but also conflict clauses when the variable is 2nd side assigned. We'll introduce new function for this purpose:

Conf2Side : Variables → Ordered Set of Conflicts

We'll have to adjust ManageNewConflict and Assign functions to manage the new data structure :

Update to ManageNewConflict (new lines are added to the end of the code) :

```
2.2. If (AssignmentStatus(A)==Side2)
    2.2.1. InsertToAnyPlaceInOrderedSet(Conf2Side(A), ω)
```

Update to Assign (new lines are added to the end of the code) :

```
5. If (AssignementStatus(A)==Side2)
    5.1. Conf2Side(V) = {}
```

Code changes to CRSAT procedure are minimal. It requires inserting a call to a function that implements this technique before line 2.4.2.1. Therefore the code between line 2.4.2 to 2.4.3 looks as follows:

Update to CRSAT (code between lines 2.4.2 to 2.4.3 non-inclusive is shown):

```
        2.4.2.1. Impl1SConfDel(StackTop(AssignedVars))
        2.4.2.2. A = UnassignTopVariable()
```

Now to the Impl1SConfDel function. The function Impl1SConfDel uses another function DeleteConfRetainingConsistency. The latter function deletes a conflict retaining the consistency of conflict-related data structures. It means that after deleting the conflict the data structure is exactly the same as if the conflict had never been existed.

void Impl1SConfDel(Variable A)

```
1. If ((AssignmentStatus(A)==Side2) AND (ChkConflictStatus(A)==NotChkConflicts) AND (Conf2Side(A)=={}))
    1.1. For (ω = FirstElemOfOrderedSet(ConflictsToCheck(A)); ω != NULL; ω = NextElemOfOrderedSet(ω,A))
        1.1.1.DeleteConfRetainingConsistency(ω)
```

void DeleteConfRetainingConsistency(Conflict ω)

```
1. For (A = FirstElemOfOrderedSet(ω); A != NULL; A = NextElemOfOrderedSet(ω))
    1.1. If (IfMemberOfOrderedSet(ConflictsToCheck(A), ω))
```

1.1.1.RemoveFromOrderedSet(ConflictsToCheck(A), ω)

1.2. If (IfMemberOfOrderedSet(Conf2Side(A), ω))

1.2.1.RemoveFromOrderedSet(Conf2Side(A), ω)

In the next subsection we'll prove that this method doesn't hurt the soundness and completeness of CRSAT

A.5.2.2 - Formal Proof

We'll prove here theorem 4.2.4.1 for CRSAT enhanced by our method. Fortunately, we don't need to prove it from the beginning.

The induction base proof is identical to that brought in Appendix A.4, since last variable on the decision stack have to participate in conflict when it's 2^{nd} side decided.

Obviously, the induction step proof is the same for every variable which condition 1 of Impl1SConfDel is false for.

Let B be a variable which condition 1 of Impl1SConfDel is true for. We reach Impl1SConfDel after checking both sides of B and finding no model, therefore conditions a and b.3 of theorem 4.2.4.1 are false, thus a.1 and b.3.1 proofs require no changes. b.1 and b.2 are also true, since there is no change in CRSAT operations before checking B's 2^{nd} side. b.4.2 is true since the 2^{nd} side unassignment happens just after the call to Impl1SConfDel function.

Therefore, we have to prove only b.4.1. Let C be a decision variable being checked just after ¬B (C must exist, since otherwise ¬B would have participated in the only conflict just after its assignment) . Let Γ be the set of conflicts recorded while C was checked. Observe, that $A_1 \ldots A_m$ and ¬B are the variables being on the decision stack just before C's assignment. According to the induction assumption for C(its decision level is obviously greater the B's) $\Gamma \Lambda \neg B \Lambda \bar{A} \rightarrow \square$. Observe now, that Γ is identical to Φ, since we delete all the conflicts, that were recorded when B was 1^{st} side assigned. Therefore $\Phi \Lambda \neg B \Lambda \bar{A} \rightarrow \square$. According to our assumption, ¬B doesn't appear in Φ. Obviously it doesn't appear in $\bar{A}$ as well. Therefore $\Phi \Lambda \bar{A}$ and ¬B are disjoint. Observe also, that any

interpretation containing B=F is a model for ¬B. Therefore, according to lemma A.5.1, $\Phi \wedge \bar{A} \rightarrow \square$.

■ (A.5.5.2)

Appendix A.5.3 - Using Conflict Variables Non-Participation in Conflicts

A.5.3.1 - Code Changes

There is a need in a new data structure to keep conflict clauses recorded while checking a conflict. We'll introduce new function for this purpose:

ConfsWhileChkConf : Variables → Ordered Set of Conflicts

It's valid for a 2nd side variable and for a conflict variable. It keeps all the conflicts recorded:

1) while checking the current conflict(for 2nd side variable)
2) while a conflict variable is assigned a value(for a conflict variable)

We'll add a new value DelS1Confs to ChkConflictStatus's range:

ChkConflictsStatus : *Variables →{ NotKnown, NotChkConflicts, ChkMiddleConflicts, ChkLastConflict, DelS1Confs}*:

We'll use auxiliary function DeleteConfRetainingConsistency from A.5.2.1 with little changes to manage ConfsWhileChkConf :

Update to DeleteConfRetainingConsistency (new line is added to the end of the code) :

```
1.3. If (IfMemberOfOrderedSet(ConfsWhileChkConf(A), ω))
    1.3.1.RemoveFromOrderedSet(ConfsWhileChkConf(A), ω)
```

We'll have to adjust ManageNewConflict, GetFirstConflict and GetNextConflict functions to manage the new data structure :

Update to ManageNewConflict (new lines are added to the end of the code) :

2.3. If ((AssignmentStatus(A)==Side2) AND (ChkConflictStatus(A) != NotChkConflicts))

2.3.1. InsertToAnyPlaceInOrderedSet(ConfsWhileChkConf(A), ω)

2.4. If (AssignmentStatus(A)==ConflictVar)

2.4.1. InsertToAnyPlaceInOrderedSet(ConfsWhileChkConf(A), ω)

Update to GetFirstConflict (new line is added to the beginning of the code) :

1. ConfsWhileChkConf(A) = {}

Update to GetNextConflict (new line is added to the beginning of the code) :

1. ConfsWhileChkConf(A) = {}

Update to Assign (new lines are added to the end of the code) :

6. If (AssignementStatus(A)==ConflictPart)

6.1. ConfsWhileChkConf(A) = {}

The change in CRSAT is just before the call to GetNextConflict at line 2.4.6.1.

Update to CRSAT (new code just before line 3.4.6.1):

2.4.6.1. If (NoConfVarParticipatedInLastConf(A)) /*code below*/

2.4.6.1.1. ChkConflictStatus(A) = DelS1Confs

2.4.6.1.2. For (ω = FirstElemOfOrderedSet(Conf2Side(A)); ω != NULL; ω = NextElemOfOrderedSet(ω,A))

2.4.6.1.2.1. If (NOT (IfMemberOfOrderedSet(ConfsWhileChkConf, ω))

2.4.6.1.2.1.1. DeleteConfRetainingConsistency(ω)

2.4.6.1.3. Go To 2.4.2

To complete the code changes, we introduce a new function :

Boolean NoConfVarParticipatedInLastConf(Variable A)

1. For (B = NextElemOfOrderedSet(CurrentlyCheckedConf(A), A); B != NULL; B = NextElemOfOrderedSet(CurrentlyCheckedConf(A),B))

1.1. If (ConfsWhileChkConf(B) != {})

1.1.1. Return F

2. Return T

A.5.3.2 - Formal Proof

For the same reasons as in A.5.2.2, we should only prove the statement b.4.1 from theorem 4.2.4.1, while the statement b.2 is true (we'll use this fact in the proof).
Let D be a decision variable being checked just after deciding all the C_j's(D must exist, since otherwise one of C_j's would have participated in the only conflict of Π just after its assignment) . Let Θ be the set of conflicts recorded while D was checked. Observe, that $A_1 \ldots A_m, \neg B, C_1 \ldots C_p$ are the variables being on the decision stack just before D's assignment. According to the induction assumption for D(its decision level is obviously greater the B's) $\Theta \wedge \neg B \wedge \bar{A} \wedge \gamma \rightarrow \square$. Observe, that Θ is identical to Π. Therefore $\Pi \wedge \neg B \wedge \bar{A} \wedge \gamma \rightarrow \square$. According to our assumption, C_j's don't appear in Π (nor in $\bar{A}$, of course). Therefore $\Pi \wedge \bar{A} \wedge \neg B$ and γ are disjoint. Any interpretation containing C_j=T for all j is a model for γ. Therefore, according to lemma A.5.1, $\Pi \wedge \bar{A} \wedge \neg B \rightarrow \square$ or $\neg \Pi \vee \neg \bar{A} \vee B \Leftrightarrow T$.

According to statement b.2 of theorem 4.2.4.1 : $\neg \Psi_1 \vee \neg \bar{A} \vee \neg B \Leftrightarrow T$. If we use the resolution rule on two latest expressions, we get $\neg \Psi_1 \vee \neg \Pi \vee \neg \bar{A} \Leftrightarrow T$. But $\Phi = \Psi_1 \wedge \Pi$, since we deleted all other conflict clauses. Therefore $\neg \Phi \vee \neg \bar{A} \Leftrightarrow T$ and we proved b.4.1.
Observe, that if $\neg B$ doesn't participate in Π and we delete Ψ_1 clauses according to the principle from 4.3.1 (and A.5.2), b.4.1 is still true, since $\Pi \wedge \bar{A} \wedge \neg B \rightarrow \square$ means $\Pi \wedge \bar{A} \rightarrow \square$ if $\neg B$ isn't in Π(for reasons brought in A.5.2.2). $\Phi = \Pi$ in this case and therefore $\Phi \wedge \bar{A} \rightarrow \square$.
It proves that we can combine pruning methods from this and previous sections.

■ (A.5.3.2)

Appendix A.5.4 - Deleting Some 1st Side Conflicts After Conflict Checking

A.5.4.1 - Code Changes

We'll introduce a function S2VarPart :

S2VarPart : Conflicts → {T, F}

It maps checked conflicts to boolean values in the following way :
For each conflict ω, s.t.

1) There exists a decided, 2nd side variable A, s.t. ConflictsToCheck(A) includes ω.
2) ω has already been checked for A
3) A is the variable with greatest decision level of all possible variables fulfilling first 2 conditions

S2VarPart(ω)=T iff A participated in at least one of the conflicts recorded while ω was checked.

We'll introduce a new function UpdateS2VarPartStatus, to manage S2VarPart.
UpdateS2VarPartStatus should be called for each conflict that has just been checked.

void UpdateS2VarPartStatus(Variable A)

1. For (ω=ConfsWhileChkConf(A); ω != NULL; ω =NextElemOfOrderedSet(ConfsWhileChkConf(A),A))
   1.1. If (IfMemberOfOrderedSet(ω, A))
      1.1.1. S2VarPart(CurrentlyCheckedConf(A)) = T
      1.1.2. Return
2. S2VarPart(CurrentlyCheckedConf(A)) = F

Now, we'll introduce the changes to CRSAT needed to implement the new technique :
Update to CRSAT (new code just before line 2.4.6.1):

2.4.6.1.UpdateS2VarPartStatus(A)

Update to CRSAT (new code between lines 2.4.2.1 to 2.4.3 non-inclusive):

```
2.4.2.2. If ((AssignmentStatus(A)==Side2) AND (ChkConflictStatus(A) !=
        NotChkConflicts) AND (ChkConflictStatus(A) != DelS1Confs))
    2.4.2.2.1.  UpdateS2VarPartStatus(A)
    2.4.2.2.2.  For (ω=ConflictsToCheck(A); ω != NULL; ω = ω')
            /* NextElemOfOrderedSet must be applied before DeleteConfRetainingConsistency to keep
            ConflictsToCheck consistent*/
        2.4.2.2.2.1. ω' = NextElemOfOrderedSet(ConflictsToCheck(A), ω)
        2.4.2.2.2.2. If (S2VarPart(ω))
            2.4.2.2.2.2.1.  DeleteConfRetainingConsistency(ω)
```

Observe that according to the above code if we used the principle from 4.3.1 (ChkConflictStatus(A)==DelS1Confs), we don't use our new principle.

A.5.4.2 - Formal Proof

For the same reasons as in A.5.2.2, we should only prove the statement b.4.1 from theorem 4.2.4.1.

We'll use here the notation of theorem's 4.2.4.1 proof.

In theorem's 4.2.4.1 proof we used Π_i to denote the conjunction of clauses which is recorded, while ω_i is being checked. In this proof there is a difference between such Π_i's, that B doesn't participate in Π_i and therefore appropriate ω_i is deleted at the end of conflicts checking for B and such Π_i's, that B does participate in Π_i and ω_i is not deleted. We'll denote by $\Theta_j (0 \le j \le t)$ such Π_i's, which don't contain B(nor ¬B, of course) and by $\Delta_k (0 \le k \le d)$ such Π_i's, which contain B.

Observe, that w = t + d. Obviously, if t=0, the theorem is true, since we don't delete no ω_i.

Now we'll prove 2 lemmas similar to A.4.2 and A.4.3, but true only for Θ_j's :

Lemma A.5.4.2.1: For each j : $\Theta_i \wedge \bar{A} \wedge \gamma_i \rightarrow \Box$.

Proof of A.5.4.2.1:

According to lemma A.4.2 : $\Theta_i \wedge \bar{A} \wedge \gamma_i \wedge \neg B \rightarrow \Box$. Observe, that $\Theta_i \wedge \bar{A} \wedge \gamma_i$ and $\neg B$ are disjoint and any interpretation containing B=F is a model for $\neg B$. Therefore, according to lemma A.5.1, $\Theta_i \wedge \bar{A} \wedge \gamma_i \rightarrow \Box$.

■ (lemma A.5.4.2.1)

Lemma A.5.4.2.2: $\bar{A} \wedge X \wedge \Theta_i \rightarrow \bar{A} \wedge X \wedge (\neg \alpha_i \vee \neg \gamma_i)$ (reminder: X is the conjunction of such Ψ_1 clauses, that don't contain not B nor $\neg B$)

Proof of A.5.4.2.2:

We must prove, that if $\bar{A} \wedge \Theta_i \wedge X$ [1] is T, than $\bar{A} \wedge X \wedge (\neg \alpha_i \vee \neg \gamma_i)$[2] is T. If [1] is T than $\bar{A}$=T, Θ_i=T and X=T.

But from lemma A.5.4.2.1 and the assumptions that $\bar{A}$=T, Θ_i=T, it follows, that γ_i=F. Therefore [2] is true

■ (lemma A.5.4.2.2)

Now we'll prove b.4.1 for our case. Observe, that Φ is a conjunction of Θ_j's, Δ_i's, clauses from Ψ_1, that weren't deleted corresponding to each Δ_i and X clauses. Therefore, we can say, that:

$\bar{A} \wedge \Phi = A \wedge X \wedge \Theta_1 \wedge \ldots \wedge \Theta_t \wedge (\Delta_1 \wedge (\neg \alpha_1 \vee \neg B \vee \neg \gamma_1)) \wedge \ldots \wedge (\Delta_d \wedge (\neg \alpha_d \vee \neg B \vee \neg \gamma_d)) = (A \wedge X \wedge \Theta_1) \wedge \ldots \wedge (A \wedge X \wedge \Theta_t) \wedge (A \wedge X \wedge (\Delta_1 \wedge (\neg \alpha_1 \vee \neg B \vee \neg \gamma_1))) \wedge \ldots \wedge (A \wedge X \wedge (\Delta_d \wedge (\neg \alpha_d \vee \neg B \vee \neg \gamma_d)))$.

According to lemma A.4.3, for each i : $(A \wedge X \wedge (\Delta_1 \wedge (\neg \alpha_1 \vee \neg B \vee \neg \gamma_1))) \rightarrow (A \wedge X \wedge (\neg \alpha_1 \vee \neg \gamma_1))$

According to lemma A.5.4.2.2, for each i : $(A \wedge X \wedge \Theta_i) \rightarrow (\bar{A} \wedge X \wedge (\neg \alpha_i \vee \neg \gamma_i))$

Therefore $\bar{A} \wedge \Phi \rightarrow (\bar{A} \wedge X \wedge (\neg \alpha_1 \vee \neg \gamma_1)) \wedge \ldots \wedge (\bar{A} \wedge X \wedge (\neg \alpha_w \vee \neg \gamma_w))$. But by regrouping, we get $(\bar{A} \wedge X \wedge (\neg \alpha_1 \vee \neg \gamma_1)) \wedge \ldots \wedge (\bar{A} \wedge X \wedge (\neg \alpha_w \vee \neg \gamma_w)) = \bar{A} \wedge X \wedge (\neg \alpha_1 \vee \neg \gamma_1) \wedge \ldots \wedge (\neg \alpha_w \vee \neg \gamma_w)$, and according to A.4.4 $\bar{A} \wedge X \wedge (\neg \alpha_1 \vee \neg \gamma_1) \wedge \ldots \wedge (\neg \alpha_w \vee \neg \gamma_w) \rightarrow \Box$.

Therefore $\bar{A} \wedge \Phi \rightarrow \Box$ (obvioulsy, if $F_1 \rightarrow F_2$ and $F_2 \rightarrow F_3$ than $F_1 \rightarrow F_3$).

■ (A.5.4.2)

Appendix B - Jerusat Solver Detailed Description

Here we'll provide extensive information about the Jerusat solver. We'll bring the Jerusat working principles in appendix B.1. In appendix B.2 we'll analyse the performance of Jerusat.

Appendix B.1 - Jerusat Working Principles

First we'll bring a table of configurable parameters to Jerusat with a description of each parameters usage. There is one or more parameter for each aspect of Jerusat implementation listed in section 5.1. In the table below the parameters are grouped according to it. In addition, a few technical parameters are grouped under " Miscellaneous".
Each parameter may be of one of the following types: Long(32 bits signed integer), ULong(32 bits unsigned integer), Boolean(0 or 1), Double(real number) or String. A group, a type, a range, a default value and a description are brought in the table for each parameter. The default values were set after numerous experiments (more about it in Appendix B.2).

Group	Name	Type	Range	Default	Description
Clause recording	*maxClsLen*	Long	≥ 3	6	Length threshold for clauses recorded according to non-implied variables scheme. Corresponds to *t* threshold from section 4.8. All clauses shorter or equal to *maxClsLen* are kept forever. Longer clauses are deleted after unassigning the *maxClsLen*+1 literal.
	clsDensity	Long	≥ 0	5	Value corresponding to *d* threshold from section 4.8. Please refer to that section for more details.
	ifAdd1UipClause	Bool	{0,1}	1	Should be 1 to record clauses according to 1UIP scheme (in addition to clauses recorded according to non-implied variables scheme).
	max1UIPLenToRec	Long	≥ 3	9	Length threshold for clauses recorded according to 1UIP scheme. All clauses shorter or equal to max1UIPLenToRec are kept forever. Longer clauses are deleted after unassigning the max1UIPLenToRec +1 literal.
[illegible]	*qsNum*	Long	Even, ≥ 2	20	The number of queues in priority queue for VAP heuristics.
	maxInterval	Long	≥ 1	20	The maximal value for variable to be in the maximal queue. Corresponds to *m* from section 4.4.
	ifGetFirst	Bool	{0,1}	1	If it's 1, than the variable taken from the maximal queue is the first variable in the queue(or the second one, details in ifLittleRand description), otherwise it's the variable with maximal s(A) value (see sec. 4.4 for details about s(A)).
	ifLittleRand	Bool	{0,1}	0	If ifGetFirst==1 and ifLittleRand==1 than variable taken from the maximal queue is the first or second variable according to coin sampling. If ifLittleRand==0 than the variable taken according to ifGetFirst value.
	ifLessSign	Bool	{0,1}	0	If 0 than first literal of maximal variable to check will be the literal with maximal number of appearances, otherwise it would be the other literal.

	maxMinHeur	Double	≤ 1.0; ≥ 0.0	0.7	Corresponds to *p* from section 4.4. Defines how should we calculate s(A) for each variable. Please refer to 4.4 for more details.
	incrHeur1UIP	Long	{0,1, 2}	1	Defines whether and how should we increase the C(A) counter for literal participating in conflict according to 1UIP scheme. If 0, counter isn't increased when recording 1UIP clause. If 1, counter is increased only for conflict side literals. If 2, counter is increased for all literals visited when recording the clause.
Shrinking Scheme	*ifPrepush*	Bool	{0,1}	0	0 if no shrinking scheme is used. 1 if non implied variables shrinking scheme is used. Please refer to 4.7 for more details.
Restarts Policy	*restartAft*	ULong	≥ 1	100	Defines the initial value for the number of conflicts, Jerusat should restart after recording it.
	restartAdd	ULong	≥ 0	10	Added to restartAft after each restart.
	ifClsOnRestart	Bool	{0,1}	1	If 1, the restart will take place only after only 1st side variables are on the stack and a clause will be recorded then as described in 4.6.2. If 0, the restarts are taken regularly with no clause recording.
WCRSAT	*notCheckConfsFrom*	Double	≥ 0.0	0.1	According to this value it's chosen for each 1st side variable A whether to check conflicts for it or to proceed as in DPLL. Let x be the number of conflicts recorded when A was 1st side checked. Let y be the number of conflicts were A participated when it was 1st side checked. Than if y/x < *notCheckConfsFrom*, conflicts would be checked.

	maxConfLen	Long	≥ 3	9	Length threshold for conflicts recorded. All conflicts shorter or equal to maxConfLen are kept forever. Longer conflicts are deleted after unassigning the maxConfLen+1 literal. This parameter defines the window size for the WCRSAT algorithm. Conflicts may be checked only for variables for which all the conflicts are kept and therefore no conflict in which it participates is deleted.
Data Structures	*ifSortInImply*	Bool	{0,1}	0	If 0, no sorting is carried out during the visiting a clause for assigning. SortedFrom is updated if the clause is found to be sorted. If 1, limited sorting possible when passing once over the literals is carried out. SortedFrom is updated if the clause is sorted afterwards.
Miscellaneous	*cutOff*	Ulong	> 0	31536000	Number of seconds after which the search is cut.
	outFile	String	valid file name	stdOut	Name of the output file. Standard output is the default value.

Table B-1

In subsequent section, we'll refer to the parameters, specifying it in *italic*.

Appendix B.1.1 - Implementing WCRSAT Algorithm + New Pruning Methods

Jerusat implements the WCRSAT algorithm (4.2.5.1) as well as the pruning methods from 4.3.

The decision whether to check conflicts or go on in DPLL-style is taken for each A as described below. Let x be the number of conflicts recorded when A was 1st side checked. Let y be the number of conflicts were A participated when it was 1st side checked. Than if

y/x < *notCheckConfsFrom*, conflicts would be checked. Observe, that if *notCheckConfsFrom*=0, than the conflicts aren't checked at all.
Another important decision to take is what is the WCRSAT window size. It's defined by *maxConfLen* parameter. All conflicts shorter or equal to *maxConfLen* are kept forever. Longer conflicts are deleted after unassigning the *maxConfLen*+1 literal. Conflicts may be checked only for variables for which all the conflicts are kept and therefore no conflict in which it participates is deleted.

Appendix B.1.2 - Clause Recording

Jerusat enables recording clauses according to non-implied variables and 1UIP schemes. It uses the literals' density relevance based learning technique described in 4.8 for clauses recorded according to non-implied variables scheme.

Parameter *maxClsLen* defines the length threshold for clauses recorded according to non-implied variables scheme. It corresponds to *t* threshold from section 4.8. All non-implied variables scheme clauses shorter or equal to *maxClsLen* are kept forever. Longer clauses are deleted after unassigning the *maxClsLen*+1 literal. *clsDensity* corresponds to *d* threshold from section 4.8. Please refer to section 4.8 for more details.

If parameter *ifAdd1UipClause* is 1, than clauses are recorded according to 1UIP scheme as well. Parameter *max1UIPLenToRec* defines the length threshold for such clauses. All 1UIP scheme clauses shorter or equal to *max1UIPLenToRec* are kept forever. Longer clauses are deleted after unassigning the *max1UIPLenToRec*+1 literal.

Appendix B.1.3 - Restarts Policy

There are 2 restart policies Jerusat is working with. The one used is defined by *ifClsOnRestart*.
If it's 0, than restart is taken after *restartAft* conflicts and after each conflict *restartAft* is increased by *restartAdd*.

If it's 1, than restarts are taken after *restartAft* conflicts, but only after all the non-implied variables on the stack are 1st side variables. A clause defining searched path is added after each restart as it's described in 4.6.2. After each conflict *restartAft* is increased by *restartAdd*.

Appendix B.1.4 - Heuristics

Jerusat uses the VAP heuristics described in 4.4. The number of queues is *qsNum*, the maximal value for s(A), s.t. A can be in the maximal queue (corresponding to *m* from 4.4) is *maxInterval*. *maxMinHeur* corresponds to *p* parameter from section 4.4.

Now we'll describe the algorithm for increasing the c(A)'s for literals. Let us suppose that A is assigned T/F. c(A)/c(¬A) is increased by 1, when variable of A/¬A participates in derivation of conflict clause according to non-implied variables scheme, i.e., every time when any literal of variable of A/¬A is added to ConflictReason according to algorithm brought in Appendix A.2. Observe that if A is assigned T, than c(A) is increased, otherwise c(¬A) is increased.
If *incrHeur1UIP*=0, than this is the only place when c(A) may be increased for every A.
If *incrHeur1UIP*=2, than the counter is also increased for variables participating in derivation of 1UIP clauses. If A is assigned T, than c(A) is increased, otherwise c(¬A) is increased.
If *incrHeur1UIP*=1, than the counter is also increased for variables participating in derivation of 1UIP clauses and being on the conflict side of the 1UIP cut. If A is assigned T, than c(A) is increased, otherwise c(¬A) is increased.

ifGetFirst and *ifLittleRand* define what variable is returned when needed.
If *ifGetFirst=0* than the variable in the maximal queue with maximal s(A) is returned.
If both are 1, than variable is taken from the maximal queue and is the first or the second variable according to coin sampling.
If *ifLittleRand*=0 and *ifGetFirst*=1 than the first variable in the maximal queue is returned.

The literal returned out of the 2 literals of the "best" variable is defined by *ifLessSign*. If it's 0 than first literal is the literal with maximal number of appearances, otherwise it would be the other literal.

Appendix B.1.5 - Data Structures

The data structure for clause managing is WLCC, described in 4.5. According to observation (1) at the end of section 4.5.2, we don't sort the clause at the end of a visit to it. A limited sorting, that leaves the complexity of clause visiting linear is performed if *ifSortInImply*=1, otherwise no sorting is performed, but just a check whether the clause is already sorted. The SortedFrom field of each clause is updated only if needed.
An important fact to mention is that the clause recording engines are built in such way that every newly recorded clause is sorted.
A few words about the data structure for conflicts' management for CRSAT implementation. The conflicts have the same literals as clauses recorded according to non-implied variables scheme. One data structure is used to save both. It is deleted only if both the clause and the conflict it represents are removed from the system.
The interesting fact is, that if we record every new clause sorted according to decision level (which we do), a 1st side literal, which conflicts are saved for is not visited as a clause literal, till it becomes unsatisfied, since it's CN value stays unchanged. Therefore each literal may be conflict literal or clause literal.
Below is a description of a (simplified) low-level structure for conflict/clause managing:

Field		Comments
Number Of Literals		Number of clauses literals
SortedFrom		Necessary for managing WLCC data structure(see sec. 4.5)
Flags		Auxiliary flags for managing the data structure
1	Literal Number	Beginning of literal boxes. Suppose this literal is a conflict literal. Contains the order, literal number and pointer to another cell of this data structure, which points to the next recorded conflict for this literal.
…		…
n	Literal Number	End of literal boxes. Suppose this literal is a clause literal. Contains the order, literal number and pointer to another cell of this data structure, which points to the next literal in this clause (as described in the section about WLCC).
Pointer to next clause for clause head literal		Beginning of pointer boxes
Pointer to next clause for literal next after clause head		
Pointer to next clause to be deleted		If this clause is to be deleted after unassigning of some number of its literals, it's part of the list of clauses to be deleted. The variable after unassignment of which the clause should be deleted "holds" the list.
Pointer to next conflict to be deleted		If this conflict is to be deleted after unassigning of some number of its literals, it's part of the list of conflicts to be deleted. The variable after unassignment of which the conflict should be deleted "holds" the list.
A pointers to next conflict of first conflict literal		Beginning of list of pointers to next conflict for every conflict literal
…		…
A pointers to next conflict of last conflict literal		End of list of pointers to next conflict for every conflict literal

Table B.1.5

The table consists of description of various fields of the CC(conflicts/clauses) low level data structure. As you can see there are three major parts in it :

1. General data, including the number of literals, the SortedFrom field and various flags.

2. Literal boxes
3. Pointer boxes. Each clause may be member in different linked lists, such as list of clauses, where a certain literal is watched, list of clauses to be deleted after unassignment of some literal etc. Those pointers point to next clause in appropriate list.

Appendix B.1.6 - Shrinking Scheme

Currently Jerusat allows to implement the non-implied shrinking scheme (4.7.1). It's turned on if *ifPrepush* is 1 and turned off otherwise.

Appendix B.2 - Jerusat Performance Analysis

Appendix B.2.1 - Choosing the Benchmarks Set

There is no customary way to compare the performance of different SAT solvers or different configurations of one solver. There is no set of benchmarks known to represent well the CNF functions. We tried to choose a reasonable set of benchmarks received from different problems.

Below we list the benchmark families we used with short description of each. All the benchmarks, but FVP-UNSAT.2.0, with an extended description can be found at [44]. FVP-UNSAT.2.0 is from http://www.ece.cmu.edu/~mvelev/. We'll sometimes use "light" versions of big families, i.e. the same families, but with lesser number of instances. We also list it in the table below and refer to it as FAMILY_lt.

Name	Short Description	Number of instances : satisfiable-unsatisfiable
qg	SAT-encoded Quasigroup (or Latin square) instances	22:9-13
jnh	Random SAT instances with variable length clauses	50:16-34
sw100-8-lp0-c5	"Morphed" Graph Colouring, 5 colourable	100:100-0
ais	All Interval Series	4:4-0
uuf150-645	Uniform Random-3-SAT, phase transition region, unforced filtered	100:0-100
uf150-645	Uniform Random-3-SAT, phase transition region, unforced filtered	100:100-0
aim-200	Artificially generated Random-3-SAT	24:16-8
phole	Pigeon hole problem	5:0-5
parity16	Instances for problem in learning the parity function	10:10-0
bf	Circuit fault analysis: bridge fault	4:0-4

flat200-479	Flat graph colouring	100:100-0
bmc	SAT-encoded bounded model checking intances	13:13-0
logistics	Planning	4:4-0
pret150	Encoded 2-colouring forced to be unsatisfiable	4:0-4
ii16	Instances from a problem in inductive inference	10:10-0
ii32	Instances from a problem in inductive inference	17:17-0
blocksworld	Planning	7:7-0
dubois	Randomly generated SAT instances	12:12-0 (without dubois100.cnf)
beijing	SAT Competition Beijing	16:15-1
beijing_lt		14:13-1(beijing without 3bitadd_31.cnf and 3bitadd_32.cnf instances)
ssa	Circuit fault analysis: single-stuck-at fault	8:4-4
hanoi	SAT-encoding of Towers of Hanoi	2:2-0
fvp-unsat.2.0	Formulas generated in the formal verification of correct superscalar microprocessors.	22:1-21
fvp-unsat.2.0_lt		7:0-7(fvp-unsat.2.0 without all *i*pipe… instances for i>3)

Table B.2.1

Appendix B.2.2 - How to Compare Different Solvers/Configurations

After selecting the benchmarks, a question that is raised is how to compare different Jerusat configurations and how to compare Jerusat vs. other solvers. One solution is to compare the sum of average times on every set of benchmarks. But this solution gives huge importance to big and hard benchmarks. A possible solutions for comparing Jerusat configurations in terms of magnitude is as follows:
Let C(i,x) be the average time of Jerusat run for some benchmarks family i and for configuration x. Let C(i,y) be the average time of Jerusat run for the same benchmarks family for configuration y. Computing C(i,x)/C(i,y) gives a way to compare the performance of configuration x vs. configuration y and to express the result by a number independent of size and complexity of benchmark family.

The following expression gives a way to compare performance of 2 configurations on a set of benchmarks families :

$$\mathbf{R(x,y)} = \sum_{\text{for every family i}} \left(\frac{C(i,x)}{C(i,y)}\right)^{k} \cdot m \text{ , where } k = \begin{cases} 1 : \text{if } C(i,x)>C(i,y) \\ -1 : \text{if } C(i,x)<C(i,y) \\ -\infty : \text{if } C(i,x)=C(i,y) \end{cases} ; m = \begin{cases} 1 : \text{if } C(i,x)>C(i,y) \\ -1 : \text{if } C(i,x)<C(i,y) \\ 0 : \text{if } C(i,x)=C(i,y) \end{cases}$$

If R is positive, than we consider y to be “better” than x , otherwise x is better than y.

The above way of comparing the performance gives too much power of influence to one family of benchmarks, where one configuration outperforms the other in many orders of magnitude. The solution below gives less power to extreme values:

$$\mathbf{L\ (x,y)} = \sum_{\text{for every family i}} \left(\log \frac{C(i,x)}{C(i,y)}\right)^{k} \cdot m \text{ , where } k = \begin{cases} 1 : \text{if } C(i,x)>C(i,y) \\ -1 : \text{if } C(i,x)<C(i,y) \\ -\infty : \text{if } C(i,x)=C(i,y) \end{cases} ; m = \begin{cases} 1 : \text{if } C(i,x)>C(i,y) \\ -1 : \text{if } C(i,x)<C(i,y) \\ 0 : \text{if } C(i,x)=C(i,y) \end{cases}$$

Comparison of Jerusat to another solvers may be done in exactly the same way.
R(Jerusat,Limmat), L(Jerusat,zChaff), etc. are defined well.

Appendix B.2.3 - Choosing the Optimal Configuration

Now we can describe how did we choose the optimal configuration or in another words how did we choose the default values for each parameter appearing in table B.1.
Jerusat has as many as 18 non-technical parameters (listed in table B.1), 2 of which are real, 9 – natural and 7 – boolean. Obviously it is impossible to check even small fraction of the parameters' space. We have to strictly define what parts of parameters' space we want to check. We used the following solution:
We built a script, which receives a few groups of parameters from table B.1. The groups have to be disjoint and the union of it gives full list of parameters. It also receives a few (1 or more) tuples for each group of parameters (say, consisting of p parameters). Each tuple consists of p members. Each member of each tuple corresponds to a certain parameter's value.
For example, it may receive:

Parameter groups	**Tuples**
maxClsLen, clsDensity, ifAdd1UipClause, max1UIPLenToRec	*1. {6,5,1,9}* *2. {7,6,0,9}*
qsNum, maxInterval, ifGetFirst, ifLittleRand, ifLessSign, maxMinHeur, incrHeur1UIP	*1. {20, 20,1,1,0,0.7,1}* *2. {8,8,0,0,1,0.5,2}*
ifPrepush	*1. {1}* *2. {0}*
restartAft restartAdd ifClsOnRestart	*1. {1000, 0, 0}* *2. {2000,100, 1}* *3. {∞, 0, 0}*
notCheckConfsFrom, maxConfLen	*1. {0.1,8}*
ifSortInImply	*1. {1}*
cutOff outFile	*1. {2700, out.txt}*

The script tries to find the best configuration as follows:
The first configuration the script runs Jerusat with, consists of parameter values, corresponding to the first tuple for each parameters group. In our example, it would be

maxClsLen=6; clsDensity=5; ifAdd1UipClause=1; max1UIPLenToRec=9

qsNum=20; maxInterval=20, ifGetFirst=1, ifLittleRand=1, ifLessSign=0, maxMinHeur=0.7, incrHeur1UIP=1;

ifPrepush=1; restartAft=1000; restartAdd=0; ifClsOnRestart=0;
notCheckConfsFrom=0.1; maxConfLen=8; ifSortInImply=1; cutOff=2700; outFile=out.txt.

Let this configuration be x. The script defines x to be the "best configuration".

Then it passes to the next configuration. Parameters from the 1st group receive values defined by the 2nd tuple for the 1st group. The other parameters stay the same.
Let this configuration be y.
Than the script calculates L("best configuration", y). If L("best configuration" ,y)>0, the script defines y to be the best configurations, otherwise x stays to be the best configuration.
It goes on in this manner till there is no more tuples to check for the 1st group. Than it passes to the 2nd group. It takes the "best configuration" and changes the 2nd group parameter values to correspond to the 2nd tuple of the 2nd group. It compares the results of Jerusat according to this configuration to the results according to "best configuration" and changes the best configuration if needed.
It goes on similarly until it reaches the last tuple of the last group.
Than it returns the "best configuration".

We have run this script a lot of times starting from different configurations and providing it different groups of parameters.
The configuration, the script converged to in most of the cases is the default configuration provided in table B.1.

Appendix B.2.4 - Jerusat vs. Limmat and zChaff

In section 5.2 we provided a table comparing the performance of default configuration of Jerusat with the performance of Limmat(version 1.0) and zChaff(version z2001.2.17). Here we'll provide 2 tables comparing Jerusat vs. Limmat and Jerusat vs. zChaff using the method from B.2.2.
The fourth column of table B.2.4.1 contains for each benchmarks family i :

$$\left(\frac{C(i,Jerusat)}{C(i,Limmat)}\right)^{k} \cdot m, \text{ where } k = \begin{cases} 1 : \text{if } C(i,Jerusat) > C(i,Limmat) \\ -1 : \text{if } C(i,Jerusat) < C(i,Limmat) \\ -\infty : \text{if } C(i,Jerusat) = C(i,Limmat) \end{cases} ; m = \begin{cases} 1 : \text{if } C(i,Jerusat) > C(i,Limmat) \\ -1 : \text{if } C(i,Jerusat) < C(i,Limmat) \\ 0 : \text{if } C(i,Jerusat) = C(i,Limmat) \end{cases}$$

The fifth column of table B.2.4.1 contains for each benchmarks family i :

$$\left(\log \frac{C(i,Jerusat)}{C(i,Limmat)}\right)^{k} \cdot m, \text{ where } k = \begin{cases} 1 : \text{if } C(i,Jerusat) > C(i,Limmat) \\ -1 : \text{if } C(i,Jerusat) < C(i,Limmat) \\ -\infty : \text{if } C(i,Jerusat) = C(i,Limmat) \end{cases} ; m = \begin{cases} 1 : \text{if } C(i,Jerusat) > C(i,Limmat) \\ -1 : \text{if } C(i,Jerusat) < C(i,Limmat) \\ 0 : \text{if } C(i,Jerusat) = C(i,Limmat) \end{cases}$$

Observe that

- the summary of the fourth column is R(Jerusat, Limmat).
- the summary of the fifth column is L(Jerusat, Limmat).

The table B.4.2.2 is similar to B.4.2.1, but it compares Jerusat and zChaff.

One can see, that Jerusat outperforms both Limmat and zChaff.

Benchmarks Family	**Jerusat** Average in seconds (7200 for a cut instance)	**Limmat** Average in seconds (7200 for a cut instance)	$\left(\frac{C(i,J)}{C(i,L)}\right)^{k} \cdot m$	$\left(\log \frac{C(i,J)}{C(i,L)}\right)^{k} \cdot m$
aim-200	0.03125	0.050833	-1.62667	-0.2113
ais	0.01	99.78325	-9978.33	-3.99906
beijing	1010.768	959.8232	1.053077	0.02246
bf	0.24275	0.418	-1.72194	-0.23602
blocksworld	8.475143	8.204571	1.032978	0.014091
bmc	28.383	67.42838	-2.37566	-0.37578
dubois	0.035	0.0125	2.8	0.447158
flat200-479	0.38391	2.4815	-6.46375	-0.81048
fvp-unsat.2.0	5063.531	4985.998	1.01555	0.006701
hanoi	74.522	3603.853	-48.3596	-1.68448
ii16	0.1773	6.43	-36.2662	-1.5595
ii32	0.900176	0.362706	2.481836	0.394773
jnh	0.0202	0.0316	-1.56436	-0.19434
logistics	0.3605	0.731	-2.02774	-0.30701
parity16	1.9605	11.0173	-5.61964	-0.74971
phole	172.7922	40.2456	4.293443	0.632806
pret150	1.1535	0.07	16.47857	1.21692
qg	15.47486	80.90673	-5.22827	-0.71836

ssa	0.05	0.09125	-1.825	-0.26126
sw100-8-lp0-c5	0.0021	0.0095	-4.52381	-0.6555
uf150-645	0.35824	0.68928	-1.92407	-0.28422
uuf150-645	1.22619	4.51746	-3.68414	-0.56634
SUM	6380.858	9873.155	-10072.4	-9.87846

Table B.2.4.1

Benchmarks Family	**Jerusat** Average in seconds (7200 for a cut instance)	**zChaff** Average in seconds (7200 for a cut instance)	$\left(\frac{C(i,J)}{C(i,zC)}\right)^{k} \cdot m$	$\left(\log\frac{C(i,J)}{C(i,zC)}\right)^{k} \cdot m$
aim-200	0.03125	0.599208	-19.1747	-1.28273
ais	0.01	48.39675	-4839.68	-3.68482
beijing	1010.768	993.4041	1.017479	0.007525
bf	0.24275	1.044	-4.30072	-0.63354
blocksworld	8.475143	13.65114	-1.61073	-0.20702
bmc	28.383	2346.6	-82.6762	-1.91738
dubois	0.035	0.494417	-14.1262	-1.15003
flat200-479	0.38391	6.76601	-17.6239	-1.2461
fvp-unsat.2.0	5063.531	3445.406	1.469647	0.167213
hanoi	74.522	3605.123	-48.3766	-1.68464
ii16	0.1773	19.1849	-108.206	-2.03425
ii32	0.900176	0.931235	-1.0345	-0.01473
jnh	0.0202	0.1485	-7.35149	-0.86638
logistics	0.3605	5.7635	-15.9875	-1.20378
parity16	1.9605	20.8417	-10.6308	-1.02657
phole	172.7922	23.724	7.283434	0.862336
pret150	1.1535	2.9635	-2.56914	-0.40979
qg	15.47486	65.92155	-4.25991	-0.6294
ssa	0.05	0.405375	-8.1075	-0.90889
sw100-8-lp0-c5	0.0021	0.1103	-52.5238	-1.72036
uf150-645	0.35824	2.81278	-7.85166	-0.89496
uuf150-645	1.22619	12.66249	-10.3267	-1.01396
SUM	6380.858	10616.95	-5246.64	-21.4922

Table B.2.4.2

Appendix B.2.5 - Influence of Jerusat Parameters on Its Performance

In this section we'll analyse the influence of Jerusat parameters on its performance. One subsection will be dedicated to each parameters group corresponding to 6 aspects of Jerusat implementation brought in section 5.1 and in the 1st column of table B.1. In each subsection we'll bring the performance of Jerusat with a few configurations that are similar to default, but with changes in one parameter. We'll compare the performance of such configuration with the performance of a default one according to the L(x,y) measuring provided in B.2.2. The cut off time for all the experiments of this section is set to 45 minutes (2700 seconds). 2700 seconds were added for an instance which has been cut off.

The performance was checked on a computer with 128Mb of memory, "x86 Family 6 Model 5 Stepping 2 Genuine Intel ~350 Mhz" processor and "Microsoft Windows 2000 Professional" operating system. We used the light version of fvp-unsat.2.0 and beijing families to save running time.

B.2.5.1 - Clause Recording

Table B.2.5.1 provides comparison of the default configuration with 3 configurations, that have the same parameters as default, but :

1. *clsDensity* = 0 (instead of 5) for the 1st configurations
2. *clsDensity* = 15 (instead of 5) for the 2nd configurations
3. *ifAdd1UIPClause* = 0 (instead of 1) for the 3d configuration.

We don't provide here performance corresponding to different *maxClsLen* and *max1UIPLenToRec* parameters, since it would inflate this work's size. It has been checked however according to B.2.2 principles and found that the current default values are the best for *maxClsLen* and *max1UIPLenToRec*.

One can see that not adding 1UIP clause really hurts the performance. This is especially true for blocksword, beijing_lt, bmc, hanoi and fvp-unsat.2.0_lt benchmark families. Observe, that the default configuration performs better that one with *clsDensity* = 0. It means that the new relevance-based learning technique, provided in 4.8 improves the performance.

Benchmarks	Default	*ClsDensity* = 0		*ClsDensity* = 15		*IfAdd1UIPClause* = 0	
	Time	Time	L(Def,This)	Time	L(Def,This)	Time	L(Def,This)
aim-200	0.031667	0.027917	0.054739	0.026667	0.074634	0.08375	-0.42238
ais	0.015	0.015	0	0.0125	0.079181	0.0125	0.079181
beijing_lt	126.7712	155.4994	-0.08871	121.9904	0.016695	161.5766	-0.10536
bf	0.21775	0.15525	0.146927	0.15775	0.139989	0.333	-0.18449
blocksworld	8.249	8.742857	-0.02525	10.05743	-0.08609	92.80786	-1.05118
bmc	28.05569	154.2396	-0.74017	108.6492	-0.58801	389.8589	-1.14289
dubois	0.036667	0.0375	-0.00976	0.035	0.020203	0.43675	-1.07596
flat200-479	0.36419	0.417	-0.05881	0.36723	-0.00361	0.38086	-0.01944
fvp-unsat.2.0_lt	756.0974	841.2867	-0.04637	827.7663	-0.03933	1579.533 (4 cut)	-0.31995
hanoi	71.933	135.3895	-0.27466	235.088	-0.5143	1357.482 (1 cut)	-1.27581
ii16	0.1703	0.1763	-0.01504	0.1381	0.091021	0.1871	-0.04086
ii32	0.850706	1.314471	-0.18897	1.289647	-0.18069	0.903647	-0.02622
jnh	0.0206	0.0192	0.030566	0.0204	0.004237	0.0234	-0.05535
logistics	0.3305	0.27025	0.087406	0.28775	0.060156	1.2845	-0.58956
parity16	1.9076	3.3455	-0.24397	3.4146	-0.25285	4.1016	-0.33247
phole	172.2594	178.8166	-0.01622	160.0818	0.031841	220.8552	-0.10792
pret150	1.1135	3.41925	-0.48724	0.295	0.576868	1.036	0.03133
qg	14.84082	10.85468	0.135841	15.87227	-0.02918	13.30636	0.047398
ssa	0.0525	0.05375	-0.01022	0.0525	0	0.09875	-0.27438
sw100-8-lp0-c5	0.0023	0.0013	0.247784	0.0023	0	0.0021	0.039509
uf150-645	0.34846	0.38568	-0.04407	0.36195	-0.0165	0.33038	0.023139
uuf150-645	1.17733	1.16514	0.00452	1.20218	-0.00907	1.0908	0.033153
SUM	*1184.846*	*1495.633*	*-1.54169*	*1487.169*	*-0.6248*	*3825.725*	*-6.7705*

Table B.2.5.1

B.2.5.2 - Heuristics

Tables B.2.5.2.a, B.2.5.2.b and B.2.5.2.c provide the performance of Jerusat with a configuration similar to default, but with changes to heuristics-related parameters.
A few observations :

- the default configuration is better than all provided here according to L(x,y) measuring.
- the default configuration is much better than others on the bmc family.
- setting *ifLessSign* to 0 improves the performance on the beijing family, but hurts badly the performance on the pret150 family
- *maxMinHeur=0.5* improves the performance on the beijing family, but should be 0.7 in order to ensure good performance on the fvp-unsat.2.0_lt
- *incrHeur1UIP* should be 1 in order to ensure good performance on hanoi, fvp-unsat2.0_lt and pret150 families

Benchmarks	Default	*ifGetFirst* = 0		*ifLittleRand* = 1		*ifLessSign* = 0	
	Time	Time	L(Def,This)	Time	L(Def,This)	Time	L(Def,This)
aim-200	0.031667	0.035833	-0.05368	0.030833	0.011582	0.0375	-0.07343
ais	0.015	1.0615	-1.84983	0.18275	-1.08577	0.005	0.477121
beijing_lt	126.7712	140.8239	-0.04566	175.0484	-0.14014	88.84436	0.154391
bf	0.21775	0.14775	0.168431	0.17775	0.088149	0.16275	0.126437
blocksworld	8.249	15.80557	-0.28241	7.535143	0.03931	9.625571	-0.06703
bmc	28.05569	109.2322	-0.59033	73.89638	-0.4206	175.5603	-0.79641
dubois	0.036667	0.039167	-0.02865	0.055	-0.17609	0.041667	-0.05552
flat200-479	0.36419	0.39419	-0.03438	0.41789	-0.05973	0.45797	-0.09951
fvp-unsat.2.0_lt	756.0974	615.2343	0.089537	685.2126	0.042752	888.1131	-0.06989
hanoi	71.933	1351.035 (1 cut)	-1.27374	254.5985	-0.54893	112.914	-0.19582
ii16	0.1703	0.4987	-0.46662	0.0741	0.361396	0.8163	-0.68064
ii32	0.850706	2.625118	-0.48937	4.665118	-0.73908	2.063176	-0.38476
jnh	0.0206	0.0192	0.030566	0.0182	0.053796	0.019	0.035114
logistics	0.3305	0.298	0.044955	0.2955	0.048614	0.19275	0.234177

parity16	1.9076	5.777	-0.48122	5.4977	-0.45969	3.5538	-0.27021
phole	172.2594	173.6354	-0.00346	189.6424	-0.04175	250.2728	-0.16223
pret150	1.1135	10.96525	-0.99333	0.3125	0.55184	675.6353 (1 cut)	-2.78302
qg	14.84082	17.22314	-0.06465	14.87127	-0.00089	22.18855	-0.17467
ssa	0.0525	0.045	0.066947	0.045	0.066947	0.0625	-0.07572
sw100-8-lp0-c5	0.0023	0.0034	-0.16975	0.0024	-0.01848	0.002	0.060698
uf150-645	0.34846	0.40718	-0.06763	0.34458	0.004863	0.40505	-0.06536
uuf150-645	1.17733	1.14183	0.013297	1.1665	0.004013	1.28601	-0.03835
SUM	*1184.846*	2446.448	-6.48097	1414.091	-2.4179	2232.259	-4.9046

Table B.2.5.2.a

Benchmarks	Default	*maxMinHeur* = 0.5		*maxMinHeur* = 0.3	
	Time	Time	L(Def,This)	Time	L(Def,This)
aim-200	0.031667	0.034583	-0.03826	0.031667	0
ais	0.015	0.01	0.176091	0.14	-0.97004
beijing_lt	126.7712	132.7701	-0.02008	81.307	0.192893
bf	0.21775	0.1325	0.215742	0.15025	0.161144
blocksworld	8.249	11.358	-0.1389	21.06629	-0.40719
bmc	28.05569	105.4517	-0.57503	183.1666	-0.81483
dubois	0.036667	0.044167	-0.08082	0.05	-0.1347
flat200-479	0.36419	0.34073	0.028918	0.43284	-0.075
fvp-unsat.2.0_lt	756.0974	1031.319 (2 cut)	-0.13482	1562.155 (4 cut)	-0.31515
hanoi	71.933	176.2055	-0.38909	1355.774 (1 cut)	-1.27526
ii16	0.1703	0.8931	-0.71969	0.5437	-0.50414
ii32	0.850706	0.970176	-0.05707	0.377176	0.353235
jnh	0.0206	0.0214	-0.01655	0.02	0.012837
logistics	0.3305	0.16775	0.294509	0.25075	0.119931
parity16	1.9076	3.609	-0.2769	5.0551	-0.42324
phole	172.2594	209.0884	-0.08415	290.6004	-0.22711
pret150	1.1135	0.4025	0.441924	0.2525	0.644429
qg	14.84082	11.61632	0.106389	16.43045	-0.04419
ssa	0.0525	0.0675	-0.10914	0.0875	-0.22185
sw100-8-lp0-c5	0.0023	0.0023	0	0.0026	-0.05325
uf150-645	0.34846	0.37894	-0.03642	0.43963	-0.10093

uuf150-645	1.17733	0.98403	0.07789	1.17132	0.002223
SUM	*1184.846*	1685.868	-1.33546	3519.504	-4.08018

Table B.2.5.2.b

Benchmarks	Default	*incrHeur1UIP* = 0		*incrHeur1UIP* = 2	
	Time	Time	L(Def,This)	Time	L(Def,This)
aim-200	0.031667	0.041667	-0.11919	0.042917	-0.13202
ais	0.015	0.015	0	0.0175	-0.06695
beijing_lt	126.7712	78.38343	0.208796	132.5512	-0.01936
bf	0.21775	0.14525	0.175842	0.1275	0.232448
blocksworld	8.249	109.1677	-1.12169	15.92571	-0.2857
bmc	28.05569	337.5149	-1.08027	76.64569	-0.43647
dubois	0.036667	0.02	0.263241	0.0325	0.052388
flat200-479	0.36419	0.37294	-0.01031	0.42771	-0.06982
fvp-unsat.2.0_lt	756.0974	1553.88 (4 cut)	-0.31284	547.174	0.140452
hanoi	71.933	1350.943 (1 cut)	-1.27371	83.0845	-0.06259
ii16	0.1703	0.1702	0.000255	0.1531	0.046239
ii32	0.850706	1.678412	-0.29512	1.256	-0.16921
jnh	0.0206	0.02	0.012837	0.0228	-0.04407
logistics	0.3305	0.3805	-0.06118	0.23025	0.156972
parity16	1.9076	3.8966	-0.3102	4.2737	-0.35032
phole	172.2594	203.3942	-0.07216	149.9192	0.060326
pret150	1.1135	1.21625	-0.03833	2073.631 (3 cut)	-3.27004
qg	14.84082	14.15227	0.020632	12.26227	0.082887
ssa	0.0525	0.05375	-0.01022	0.05375	-0.01022
sw100-8-lp0-c5	0.0023	0.0019	0.082974	0.0017	0.131279
uf150-645	0.34846	0.37087	-0.02707	0.48271	-0.14153
uuf150-645	1.17733	1.21413	-0.01337	1.31051	-0.04654
SUM	*1184.846*	3657.033	-3.98108	3099.626	-4.20185

Table B.2.5.2.c

B.2.5.3 - Shrinking Scheme

Table B.2.5.3 provides the performance of Jerusat with a configuration similar to default, but with changes to the shrinking-scheme related parameter (*ifPrepush*).
The default configuration is much better overall than one with non-implied variables shrinking scheme applied. However the latter configuration is the best from all the checked Jerusat configurations for the fvp-unsat.2.0_lt family.
The other shrinking schemes provided in 4.7 may also be implemented and tried in future works.

Benchmarks	Default	*ifPrepush* = 1	
	Time	Time	L(Def,This)
aim-200	0.031667	0.036667	-0.06367
ais	0.015	0.7685	-1.70955
beijing_lt	126.7712	163.4886	-0.11047
bf	0.21775	0.19275	0.052964
blocksworld	8.249	14.95014	-0.25824
bmc	28.05569	88.78392	-0.50031
dubois	0.036667	0.039167	-0.02865
flat200-479	0.36419	0.49487	-0.13316
fvp-unsat.2.0_lt	756.0974	295.7611	0.407637
hanoi	71.933	560.927	-0.89198
ii16	0.1703	0.1842	-0.03407
ii32	0.850706	1.790353	-0.32316
jnh	0.0206	0.02	0.012837
logistics	0.3305	0.3355	-0.00652
parity16	1.9076	5.5888	-0.46683
phole	172.2594	190.6998	-0.04417
pret150	1.1135	675.4105	-2.78288 (1 cut)
qg	14.84082	13.70609	0.034544
ssa	0.0525	0.06125	-0.06695
sw100-8-lp0-c5	0.0023	0.0024	-0.01848
uf150-645	0.34846	0.43525	-0.09659

uuf150-645	1.17733	1.30261	-0.04392
SUM	*1184.846*	2014.98	-7.07161

Table B.2.5.3

B.2.5.4 - Restarts Policy

Table B.2.5.4 provides the performance of Jerusat with a configuration similar to default, but with changes to restarts-strategy related parameters.

The main conclusions from the data provided, is that

1. Restarting improves the performance
2. New restarts policy from 4.6.2 improves the performance

Benchmarks	Default	*restartAft* =1000		restartAft = ∞		*ifClsOnRestart* = 0	
	Time	Time	L(Def,This)	Time	L(Def,This)	Time	L(Def,This)
aim-200	0.031667	0.03125	0.005752	0.03	0.023481	0.027917	0.054739
ais	0.015	0.0125	0.079181	0.015	0	0.015	0
beijing_lt	126.7712	148.5715	-0.06891	186.1076	-0.16674	194.2849 (1 cut)	-0.18542
bf	0.21775	0.27775	-0.1057	0.27525	-0.10177	0.14525	0.175842
blocksworld	8.249	6.443714	0.107265	13.398	-0.21064	2.855571	0.460708
bmc	28.05569	54.84746	-0.29114	67.51785	-0.3814	80.93554	-0.46012
dubois	0.036667	0.0275	0.124939	0.026667	0.138303	0.030833	0.075251
flat200-479	0.36419	0.54919	-0.17839	0.55448	-0.18256	0.59427	-0.21266
fvp-unsat.2.0_lt	756.0974	724.765	0.018381	1293.595 (3 cut)	-0.23322	1448.705 (1 cut)	-0.2824
hanoi	71.933	72.5345	-0.00362	37.1635	0.286812	658.447	-0.96159
ii16	0.1703	0.1671	0.008238	0.1671	0.008238	1.1265	-0.82052
ii32	0.850706	1.067	-0.09838	0.936882	-0.04191	0.622235	0.135825
jnh	0.0206	0.0196	0.021611	0.0194	0.026065	0.022	-0.02856
logistics	0.3305	0.353	-0.0286	0.348	-0.02241	0.12	0.43999
parity16	1.9076	4.5853	-0.38088	4.5933	-0.38164	5.3455	-0.4475
phole	172.2594	187.6476	-0.03716	170.7954	0.003707	411.93	-0.37864
pret150	1.1135	0.76075	0.165448	455.9553	-2.61223	21.50025	-1.28575

qg	14.84082	15.98532	-0.03226	16.92532	-0.05708	28.20909	-0.27893
ssa	0.0525	0.0625	-0.07572	0.06125	-0.06695	0.05125	0.010465
sw100-8-lp0-c5	0.0023	0.0021	0.039509	0.0025	-0.03621	0.002	0.060698
uf150-645	0.34846	0.34775	0.000886	0.34975	-0.0016	0.60986	-0.24308
uuf150-645	1.17733	1.30967	-0.04626	1.3457	-0.05805	2.38634	-0.30683
SUM	*1184.846*	1220.368	-0.77582	2250.183	-4.0678	2857.966	-4.47848

Table B.2.5.4

B.2.5.5 - WCRSAT

Tables B.2.5.5.a and B.2.5.5.b provide the performance of Jerusat with a configuration similar to default, but with changes to the *notCheckConfsFrom* parameter. Remember, that *notCheckConfsFrom=0.0,* means that CRSAT doesn't check conflicts like the regular DPLL algorithm.

Observe, that the default configuration with *notCheckConfsFrom=0.1* is better than each of the others in terms of the overall performance according to L(x,y). However, the configurations with *notCheckConfsFrom=0.0* and *notCheckConfsFrom=0.3* are better than the default according to average times sum. This is due to the better performance on the (hard) fvp-unsat.2.0_lt family.

Observe also, that the default configuration is better than one with *notCheckConfsFrom=0.0* for satisfiable families bmc, hanoi and beijing_lt, but is worse for unsatisfiable families fvp-unsat.2.0_lt and phole.

Benchmarks	Default	*notCheckConfsFrom* = 0.0		*notCheckConfsFrom* = 0.2	
	Time	Time	L(Def,This)	Time	L(Def,This)
aim-200	0.031667	0.03125	0.005752	0.042167	-0.12437
ais	0.015	0.015	0	0.01	0.176091
beijing_lt	126.7712	160.0415	-0.10121	95.00007	0.125297
bf	0.21775	0.18275	0.076101	0.17025	0.106871
blocksworld	8.249	9.280571	-0.05117	10.93871	-0.12256
bmc	28.05569	76.63623	-0.43641	172.8432	-0.78963
dubois	0.036667	0.035	0.020203	0.036667	0
flat200-479	0.36419	0.35697	0.008696	0.35579	0.010134

fvp-unsat.2.0_lt	756.0974	572.6607	0.12068	862.775	-0.05732
hanoi	71.933	94.8915	-0.1203	241.7575	-0.52645
ii16	0.1703	0.1753	-0.01257	0.1493	0.057155
ii32	0.850706	0.912176	-0.0303	6.616706	-0.89086
jnh	0.0206	0.0202	0.008516	0.0204	0.004237
logistics	0.3305	0.3405	-0.01295	0.29025	0.056399
parity16	1.9076	1.8734	0.007857	1.9926	-0.01893
phole	172.2594	152.7032	0.052335	176.6234	-0.01087
pret150	1.1135	1.17375	-0.02289	1.051	0.025088
qg	14.84082	15.95727	-0.0315	11.57477	0.107945
ssa	0.0525	0.0525	0	0.0525	0
sw100-8-lp0-c5	0.0023	0.0029	-0.10067	0.0027	-0.06964
uf150-645	0.34846	0.38791	-0.04658	0.34442	0.005065
uuf150-645	1.17733	1.1626	0.005468	1.16074	0.006163
SUM	*1184.846*	1088.893	-0.66094	1583.808	-1.93019

Table B.2.5.5.a

Benchmarks	Default	*notCheckConfsFrom* = 0.3		*notCheckConfsFrom* = 0.5	
	Time	Time	L(Def,This)	Time	L(Def,This)
aim-200	0.031667	0.030833	0.011582	0.0325	-0.01128
ais	0.015	0.0125	0.079181	0.015	0
beijing_lt	126.7712	142.6135	-0.05114	136.2307	-0.03125
bf	0.21775	0.15025	0.161144	0.16775	0.113296
blocksworld	8.249	14.75271	-0.25247	6.98	0.072546
bmc	28.05569	96.75608	-0.53766	51.78531	-0.26619
dubois	0.036667	0.041667	-0.05552	0.035833	0.009984
flat200-479	0.36419	0.35458	0.011614	0.43713	-0.07928
fvp-unsat.2.0_lt	756.0974	657.7814	0.060496	870.469	-0.06118
hanoi	71.933	81.6025	-0.05478	115.932	-0.20728
ii16	0.1703	0.1643	0.015577	0.1342	0.103462
ii32	0.850706	1.028	-0.08221	0.650471	0.116552
jnh	0.0206	0.0178	0.063447	0.021	-0.00835
logistics	0.3305	0.27275	0.083407	0.31025	0.02746
parity16	1.9076	1.9797	-0.01611	3.7902	-0.29817
phole	172.2594	168.0434	0.010761	192.278	-0.04775

pret150	1.1135	1.777	-0.203	1.842	-0.2186
qg	14.84082	12.17427	0.086015	12.10418	0.088522
ssa	0.0525	0.05125	0.010465	0.0475	0.043466
sw100-8-lp0-c5	0.0023	0.0019	0.082974	0.0022	0.019305
uf150-645	0.34846	0.3958	-0.05532	0.40751	-0.06799
uuf150-645	1.17733	1.15715	0.007509	1.16881	0.003154
SUM	*1184.846*	1181.159	-0.62403	1394.842	-0.69957

Table B.2.5.5.b

B.2.5.6 - Data Structures

Tables B.2.5.6 provide the performance of Jerusat with a configuration similar to default, but with changes to the *ifSortInImply* parameter. Observe, that the default configuration's performance is better. This means that sorting the clause in the process of clause visiting is not paying off.

Benchmarks	Default	*ifSortInImply* = 1	
	Time	Time	L(Def,This)
aim-200	0.031667	0.033333	-0.02228
ais	0.015	0.015	0
beijing_lt	126.7712	171.1076	-0.13025
bf	0.21775	0.21525	0.005015
blocksworld	8.249	10.67971	-0.11216
bmc	28.05569	79.09385	-0.45012
dubois	0.036667	0.035833	0.009984
flat200-479	0.36419	0.38554	-0.02474
fvp-unsat.2.0_lt	756.0974	872.2014	-0.06204 (1 cut)
hanoi	71.933	478.278	-0.82275
ii16	0.1703	0.1842	-0.03407
ii32	0.850706	0.837176	0.006962
jnh	0.0206	0.0188	0.039709
logistics	0.3305	0.353	-0.0286
parity16	1.9076	3.7041	-0.2882
phole	172.2594	174.132	-0.0047

pret150	1.1135	0.3475	0.505735
qg	14.84082	16.66527	-0.05035
ssa	0.0525	0.0525	0
sw100-8-lp0-c5	0.0023	0.0024	-0.01848
uf150-645	0.34846	0.36645	-0.02186
uuf150-645	1.17733	1.19072	-0.00491
SUM	*1184.846*	1809.9	-1.50811

Table B.2.5.6