UCLouvain

epl

École polytechnique de Louvain

Study of SAT solvers improvement techniques

Author: **Florian FITVOYE**
Supervisors: **Charles PECHEUR, Xavier GILLARD**
Readers: **Charles PECHEUR, Xavier GILLARD, Siegfried NIJSSEN**
Academic year 2018–2019
Master [120] in Computer Science and Engineering

Abstract

The satisfiability problem (SAT) is a well known problem in propositional logic. Actually, SAT was the first problem to be proven NP-complete by Cook in 1971 [1]. SAT solvers have been developed to solve this specific problem efficiently and have been the subject of remarkable improvements since the mid 90s. Many different solvers and techniques were devised to enhance the solver. This started a race to come up with more and more performing solvers. Each solver has its own architecture making it difficult to compare the techniques with one another.

This thesis proposes an extension of Rsolve solver. Different techniques were added to this solver and thus share the same architecture. A analysis of the different techniques of the same kind between them is shown. These techniques are the variable selection heuristics, the restart strategies and inprocessing techniques. Sharing the same architecture allows fair comparison, eschewing implementation tricks and optimisations implemented by some state of the art solvers.

Contents

Chapter 1

Introduction

SAT problem

The SAT problem is the problem of deciding if a given Boolean formula can evaluate to True. A Boolean formula is a logical statement which is either True or False.

SAT is a decision problem. That is, we want to know if there exists a solution satisfying some boolean formula given as an input. If an interpretation satisfying the formula exists, the problem is said *satisfiable* or SAT. In this case, it is enough to show a working interpretation, it proves that the instance is satisfiable. For a decision problem, a single solution is sufficient. There is no need to find all the solutions. If none exists, the problem is said *unsatisfiable* (UNSAT) and a proof of the absence of solution should be provided.

Consider the following example : $l_1 \vee l_2 \wedge l_3 \vee l_1$ is a Boolean formula. It is composed of literals (l_i) separated by logical AND ($\wedge$) or logical OR ($\vee$). To know if the Boolean formula can be satisfied, an interpretation that satisfies the formula must be found. An interpretation is a complete assignment, where a value is assigned to each variable present in the instance. In boolean logic, only two values are possible, either True or False. A variable is generally denoted by a letter and a number (i.e. x_5) and has 2 literals, one positive (x_5) and one negative ($\overline{x_5}$). These literals are present in the Boolean formula. With the example above, (l_1 = True, l_2 = False, l_3 = False) is an interpretation that satisfies the formula and is therefore a solution. Another interpretation, (l_1 = False, l_2 = True, l_3 = False) does not satisfies the formula.

The rise of SAT solver

Since mid 90s, the SAT problem has been subject to a real interest and remarkable improvements with the development and enhancement of SAT solvers.

However, numerous algorithms and techniques have been proposed in various forms in order to solve the SAT problem. All the techniques aim at reducing the computational time needed. Highly efficient solvers are now actively being used, either directly or as a core engine of a larger system, to solve real-world problems from many application domains from noise prediction in integrated circuits [2] to termination analysis in term rewrite systems[3] but also in planning [4]. In some applications, the use of SAT provides remarkable performance improvements.

The most famous and influential solvers are GRASP, MiniSAT [5], BerkMin [6], GLUCOSE [7], LINGELING [8] and much more. Every year, their performance increases and is demonstrated during a SAT competition.

Comparison of techniques

In the literature, All the solvers are comparing their techniques with others techniques from other solvers. But the architecture and representation of the core of the solver are not the same. Furthermore, the techniques are relying on structure tricks to perform better, making the comparison difficult. In this thesis, a unique solver is provided, with the same core component for all the techniques, making it easier to perform a fair comparison. Based on this solver, a comparative study will be performed to compare the impact of the different techniques used in practice.

Contribution

In that context, the contribution of this master's thesis is to provide a unique solver which is very modular and acting as a toolbox, containing several techniques and making it easy to switch from one to another and to compare them efficiently, on the same architecture.

To that end, the thesis first presents the principles of SAT solving. It explains in details how SAT solvers are working with some theoretical background. This thesis focuses on the most popular solver nowadays, Conflict Driven Clause Learning solvers (CDCL).

Then, it presents Rsolve, the base solver in which the techniques have been added. In that part, we present the architecture and implementation of the solver.

After that, it presents resolve, the base solver in which the techniques have been added. In that part, we present the architecture and implementation of the solver.

Finally, a comparative study is shown to compare the different techniques between them.

Chapter 2

Background

The objective of this chapter is to present the SAT problem and to show the techniques used to solve it. To this end, we will start by showing that SAT problem is a difficult problem (NP-complete). Then, the formula decomposition into the core components, the clauses will be shown. After that, the different solver types will be presented to finally focus on one type, the Conflict-Driven Clause Learning (CDCL) solvers.

2.1 NP problems

In Computational complexity theory, there are two families of problems, the first is called **P**. It contains problems for which an efficient algorithm is known. When we talk about efficiency, it is meant that there *exists* an algorithm that finds the solution in *polynomial time*, for any instance of the problem.

The second family is called **NP**, which stands for Non-deterministic polynomial. The NP family regroups problems for which no polynomial time algorithm is known. Meaning that we don't have an a priori knowledge of the time it will take to solve the instance.

Due to the property of *efficiency*, the solution of a P problem can be found in an acceptable amount of time, even for large input size. But for NP problems, as the input size increases, the time needed grows exponentially, leading to estimated time of a year or even more than a decade for a not so large instance. Let's consider an example : we have an instance with 50 variables. If we try every possible assignments ($2^{50} \simeq 10^{16}$) and each assignment takes no more than a nanosecond, we need 13 days to process all the possibilities !

SAT is also known to be the first NP-complete problem, thanks to Cook and his theorem [1], published in 1971 showing that SAT is a NP-complete problem. A NP-complete problem is a NP problem having the following property, every NP problem can be reduced to it by a polynomial reduction. It shows that SAT is computationally harder than other NP problems.

The NP class is very important nowadays and is even at the heart of one of the *Millennium questions*. The mathematicians are working on a proof to decide whether $NP = P$ or $NP \neq P$. Finding an efficient algorithm for one of the NP-complete problem like SAT would be a proof that $P = NP$ since every problem in P can be reduced to a NP-complete problem by definition.

2.2 CNF

In Boolean logic, a formula is in conjunctive normal form (CNF) or clausal normal form if it is a conjunction of one or more disjunctions :

$$\bigwedge_{j=1}^{m}(\bigvee_{i=1}^{n} l_i)$$

where m is the number of disjunctions and n the number of components in the disjunction j.

Transformation to CNF is potentially exponential. The naive approach uses De Morgan's laws[1] and the distributive property to convert it to CNF. However, this can result in an exponential increase in equation size. The Tseytin transformation [9] outputs a formula whose **size has grown linearly** with respect to the input formula. The new expression obtained after the Tseytin transform is *equisatisfiable*, meaning that it is satisfiable if, and only if, the original input equation is satisfiable.

2.3 Clause

An instance of the SAT problem consists of a CNF expression. This form allows to express the Boolean expression as a conjunction of clauses. The general expression of CNF is

$$\bigwedge_{j=1}^{m}(\bigvee_{i=1}^{n} l_i)$$

[1] $\overline{A \wedge B} = A \vee B$ and $\overline{A \vee B} = A \wedge B$

and allows the expression to be split in smaller independent components, the clauses. The expression can be rewritten as

$$\bigwedge_{j=1}^{m}(C_j)$$

where C_j are the clauses. A clause of size n is simply a disjunction of n literals : $l_1 \vee l_2 \vee ... \vee l_n$.

The length of an instance is very huge and can go up to millions of literals. The good news is that, in practice, the solver will not directly work with that huge monolithic formula. It can directly work with these smaller clauses. Mathematically, because the formula is in CNF, each underlying clause needs to be satisfied independently in order to satisfy the whole instance. Indeed, let C_i be one of the clauses, C_i is unsatisfiable and thus will always be False in any interpretation, with the outer logical AND of previous expression, the whole expression will be evaluated to False and thus, the instance is unsatisfiable.

For this reason, the initial problem will be represented by a set of clauses. It will be easier to work with these because they are smaller and the unsatisfiability of one proves the unsatisfiability of the initial expression.

2.4 Solvers

To address the SAT problem, SAT solvers have been developed. These SAT solvers provide a "black-box" procedure. They receive a formula as input and output a solution if one exists or output the proof that none exists. But because of the NP-completeness of the SAT problem, all known algorithms admit a worst-case for which an exponential run time is required.

The last few years have seen great progress in the performance of SAT solvers. Despite the fact that SAT is NP-complete, these solvers are more and more used as a general-purpose tool. This is the case in many fields like software and hardware verification, planning, automatic test pattern generation, scheduling but also challenging problems from algebra[10].

SAT competitions are organised each year and led to the development of many clever solver implementations. Thanks to that, new techniques appear, and real-world instances as well as challenging crafted benchmark problems are proposed. Every year, new benchmarks come up at the SAT competition [11]. These solvers can often solve hard structured problems with over a million variables and several million constraints [12]. Nevertheless, many of the crafted and industrial instances remain open even for the best performing solvers.

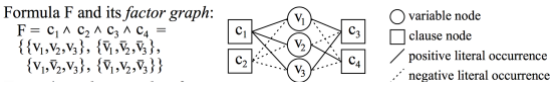

Figure 2.1: Factor graph

2.4.1 Types of solvers

There exist several types of solvers, here are some types of solvers :

- Stochastic Local Search (SLS) solver. As the name suggests, this type of solver uses local search and stochastic approaches to find a solution. It tries to locally minimize the number of conflicts to eventually come up with an interpretation without conflict (a solution). This kind of solver is considered as one of the best solving techniques for randomly generated problems and more recently also have shown great promise for several types of hard combinatorial problems. YalSAT [13], probSAT [14] are examples of SLS solvers.

- Conflict Driven Clause Learning (CDCL) solver. This is the kind of solver this thesis focuses on. It will be discussed later with greater details.

- Message passing Inspired Decimation (MID) solver. It iteratively computes the probability of a variable assignment (bias) and then simplifies the formula under that assumption. That paradigm is called message passing because marginal probabilities are computed by exchanging messages in the factor graph (see Fig 2.1) of the formula until equilibrium (no variation greater than a given threshold). Unfortunately, the convergence of that model is not guaranteed. It works well for crafted and random instances. Dimetheus is an example of MID solver [15] [16].

- Portfolio of solvers. This kind of solver implements a lot of algorithms and uses the computer cores to run different solvers in parallel like MiniSAT [5]. Some solver also use machine learning to decide which one to apply like SATzilla [17].

For the rest of this thesis, the focus will be set on one type of solvers : the CDCL solvers. As of today, this kind of solver is the most widely used and the most promising one nowadays.

2.5 Conflict Driven Clause Learning (CDCL)

CDCL solvers are based on a search algorithm, called DPLL and uses fast Boolean Constraint Propagation (BCP) which will be discussed later. This kind of solver is

the most used ones due to the unmatched efficiency and performance of DPLL in solving real-world problems. The CDCL solvers are constantly *learning* new clauses during the search. When they encounter a conflict, they analyse the causes of that conflict and they derive (learn) a new clause from it. This new clause is then added to clauses databases, which contains the problem clauses and the learned clauses so far. A conflict occurs when the current partial assignment requires that both a literal and its negation must be True for the formula to be satisfied. This, of course, cannot happen in a solution. Thus we know before the assignment is complete that any complete assignment containing the current partial assignment is not a solution.

The heuristics are improving the search thanks to this. Finding a conflict *earlier* in the tree (not in a leaf but in any node) allows a pruning of the search space. We will not visit the subtrees of an already conflicting node because we know they can not contain any solution.

Most CDCL solvers also have a restart strategy. In some cases, the current branch of the search tree is not promising enough. In these cases, we want to get out of this branch and look somewhere else in the search tree where there is a higher chance to find a solution. This is done with the restart strategy, with different heuristics, it looks if the current branch is promising or not. Based on the heuristic, it decides to either stay on the branch, if it makes progress and looks promising, or leave the current branch for another one. There exist different restart strategies, some of them will be discussed in a later section.

2.5.1 DPLL

Davis-Putnam-Logemann-Loveland (DPLL) is an algorithm that was introduced in 1962 by Davis, Putnam, Logemann and Loveland and improves the earlier David-Putnam algorithm. This algorithm is a complete, backtracking-based search algorithm. This algorithm consists in a depth-first binary search, where a choice is made on each branch of the search tree. In a SAT solver, this choice is made by selecting a variable and a value. An unassigned literal is thus chosen and assigned to *True*. When a leaf is reached, we have a complete interpretation. If the interpretation is a solution, the search stops, the solution is returned and the instance is SAT. Otherwise a backtrack is performed, unassigning all literals until the last literal that was assigned to *true* during a branch choice(chronological backtracking). This last literal is then assigned to false and the search continues. By doing so, all the possible interpretations are computed. If the whole tree was explored without finding a solution then there is no solution and the given instance is UNSAT. This is a proof by exhaustion.

This algorithm [18] is naive in the sense that it tries all the possible interpretations and has a complexity of $\mathcal{O}(2^n)$ where n is the number of variables in the instance. As said before, there is no known polynomial algorithm to solve the SAT problem. DPLL is an exponential algorithm that can be used for SAT but needs to be improved to be more efficient. Over the past decades, researches have been done to improve this algorithm by reducing the search space with different techniques. This algorithm has been used in SAT for the first time with the GRASP solver [19] [20].

The main idea behind all the techniques is to reduce as much as possible the search space (which is exponential), by pruning the initial tree. First, we can have a variable selection heuristic and/or value selection heuristic. These heuristics determine which variable (resp. value) to choose in the next branching during the search. For example, a variable heuristic could simply consist in following the numerical order of the variables, while the value heuristic could be to always assign the Truth value on the left-end side of the search and the false value on the right-end side. Of course, there exist some clever heuristics that I will discuss later.

2.5.2 Fast BCP

One of the main components of the CDCL solvers is the Boolean Constraint Propagation (BCP) process. This represents approximately 80% of the run time in most of the actual solvers. Therefore, an efficient BCP is the key to any SAT solver. The idea behind the BCP is the following: given a formula and a set of assignments defined by the current state of the search, find any necessary assignment. A necessary assignment is a literal assignment that is mandatory. Meaning that if we have assigned the opposite literal instead, at least one of the clause would have become false. Once such a necessary assignment is found, it is added to the current set of assignments. The process is repeated as long as new necessary assignments are found or a conflict is identified. In practice, necessary assignments are determined exclusively by the *unit clause rule*. The solver will process as follows : it searches after unit clauses (clauses not yet SAT that only have one unassigned literal). In order for these clauses to be SAT, the algorithm assigns the literal and propagates this assignment to every other clause. This process can lead to a chain reaction, by deriving new unit clauses and thus assigning a large amount of literals.

Unit clause rule Given a CNF clause, two different cases are possible for a non-satisfied clause:

- The clause has two or more unassigned literals.
- The clause has only one unassigned literal (unit clause).

In the first case, nothing can be deduced but in the second, the literal is the last one unassigned, meaning that if we want the clause to be SAT, this literal must to be set to True. This is called the unit clause rule. Clauses that are reduced to a single unassigned literal are said *unit*. BCP will assign the remaining literal of a unit clause and propagate it to the other clauses. This is done for each new unit clauses since the last BCP run. This process stops when either there is no more unit clause or there is a conflict. If a conflict occurs, a conflict analysis and a backtrack are performed, this will be discussed later. If there is no more unit clause, no more knowledge can be gathered in the current state and a decision needs to be made in order to continue the search.

To continue the search, a decision needs to be made because the BCP can not find more necessary assignments at the current point with the current clauses. The algorithm will thus choose a variable among the unassigned ones and assign it to a value. As a new literal is assigned, a new propagation will be performed. This process will cycle until a solution is found.

The literal picked during the decision and the literals assigned with BCP will constitute a *decision level*. A decision level is a level in the search tree and is defined by all the literals assigned due to the given decision. These literals are : the decision choice and all the necessary assignments found with BCP. All literals of the same level are also called a *block*. The number of the current level is an indication of the number of decisions made up to that point in the search.

Consider the following example, let assume the following clauses : $C_1 = l_1 \vee l_2 \vee l_3$, $C_2 = l_2 \vee l_3$, $C_3 = \overline{l_3} \vee l_4$, $C_4 = \overline{l_4} \vee l_5$, $C_5 = l_6$. Assuming that the solver makes the following decisions, the levels are :
level 0 : l_6, forced by C_5 which is unit.

1. decision : l_1, level 1 : $[l_1]$. There is only the decision literal because no unit clause is derived by the decision.

2. decision : $\overline{l_2}$, level 2 : $[\overline{l_2}, l_3, l_4, l_5]$. C_2 becomes unit, causing l_3 to be a necessary assignment. As l_3 is assigned, C_3 becomes unit and l_4 is a necessary assignment. As l_4 is assigned, C_4 becomes unit and l_5 is a necessary assignment. This level illustrates the possible chain-reaction as mentioned above. As all variable have a value, we have a complete assignment that satisfies all the clauses.

BCP will tend to *reduce* the depth of the search space. Indeed, as the necessary assignments are done between two decisions, the newly assigned literals will not be

a decision choice later in the search. If we compare with the basic DPLL algorithm, where all the variables in the instance are decision choice, this lowers the number of decisions. Thus the depth of the search space can be lowered.

An easy way to know if a clause is unit is to keep a counter of the literals with value *false* in the clause and increment this counter each time a new literal's value is set to *false*. Once the counter reaches the clause length minus one, the clause is unit and if the clause is not already satisfied, we need to set the last literal value to 1. When a new literal is assigned, we need to update the counter of every clauses that own this variable. This is a good method but as the BCP is the key component of the solver, representing a great proportion of the solver time, some clever methods have been developed. In practice for the current solvers, the BCP represents more than 80% of the total run time. In practice, we only want to know if a clause is unit, not how many literals are still unassigned. An improvement can be done with the **Two-Watched literals** scheme.

Two-Watched literals With Two-Watched literals, one no longer retains a counter for each clause, but instead "watches" two unassigned literals, these two literals can be randomly picked. These two are said to be *watched* because they are the only ones to matter. As long as both are unassigned, we don't need to care about all the other literals values in the clause. It is the case because the clause cannot be unit since at least the two watched literals are unassigned.

Assume now that one of the two watched literals becomes assigned. As the clause needs two watched literals, this literal must be replaced. There are two possibilities, either there is at least another unwatched and unassigned literal which becomes watched (can be picked randomly). Or there is no other unwatched and unassigned literal, meaning the clause is unit and the only watched literal remaining must be assigned to True and propagated throught BCP.

An additional benefit of this approach arises during the backtrack. With the counters approach, all the counters of each variable/clause need to be decremented. But with the Two-Watched literals approach, this is no longer needed.

Let's consider all the possibilities of backtrack on a fully assigned clause:

- the clause stays fully assigned (the backtrack does not affect any literals in the clause). In this case, all the information has yet been propagated, nothing needs to be done.
- the clause becomes unit : Cannot happen. Indeed, a unit clause is the consequence of a literal assignment. Therefore, the two last literals of the

clause are on the same block. As the backtrack revert entire blocks, these two literals must be reverted together.

- two or more literals in the clause are reverted and become unassigned : Since the backtrack reverts the latest assigned literal first, the two watched literals will always be the two first reverted literals of the clause.

By consequence, in any case, we have either a fully assigned clause, either two unassigned literals watched. It cannot happen that only one of the two watched literals is unassigned and another literal in the clause is unassigned and unwatched after a backtrack.

Consider now a not fully assigned clause. In this case, after the backtrack, the watchers are still unassigned and we still watch two unassigned literals, no matter what appends with the other literals.

In conclusion, if two or more literals are reverted, the watched literals will always be unassigned first and we keep the property that the clause cannot be unit as the two watchers are unassigned. This property is kept without doing anything on backtrack.

2.5.3 Early termination

Another improvement to the DPLL algorithm is the early termination. If there is no solution on a branch, we want to stop the search on that branch as soon as possible. It is possible to stop *before* a complete assignment. Indeed, if one of the clauses is UNSAT with the current partial assignment, the clause will still be unsat with the complete assignment and so the formula. Furthermore, if a variable should be negative for a clause and positive for another, it is also Unsat. Theses behaviour is called a conflict. Detecting a conflict during the search will reduce the search space.

Here is a little example with the following clauses, $C_1 = a \vee b \vee c \vee e$, $C_2 = \overline{a} \vee d$, $C_3 = b \vee \overline{a}$ and $C_4 = \overline{d} \vee \overline{b}$. Consider the following partial assignment : $(a = True)$. Then, with the BCP, $d = True$ from C_2, $b = True$ from C_3 and $b = False$ from C_4. Even if variables c and e haven't yet have value, we know that the branch $(a = True)$ has no solution, because the variable b must be True in C_2 and False in C_3. So we leave that branch to search on another $(a = False)$, where there might be a solution ($a = False$, $b = False$, $c = True$, $d = True$, $e = False$). This reduces the search space as on the initial branch $(a = True)$ we avoid the branch on variables c and e.

The early termination is done through conflict detection.

2.6 Clauses learning

The main particularity of the CDCL solvers is that they are **Conflict Driven** and they learn new clauses *during the search*. These new clauses are learned from a conflict. A *conflict* occurs when a variable must have its literals (positive and negative literals) assigned to true due to different clauses and the solver decisions. Which is impossible because a variable cannot be true and false at the same time. Once a conflict is found, the solver tries to understand the *causes* of the conflict and how it happened, this process is called **conflict analysis**. Once the causes of the conflict are found, a new clause (called **learned clause** because it is learned during the search) is derived. This clause is there to ensure the solver will not make the same decisions leading to the conflict by making explicit the necessary constraints. This new clause is redundant with respect to the initial formula. But it explicits new constraints that were hidden in the initial formula. It was hidden because the representation of the formula is done through clauses. The solver works with the clauses independently, meaning that constraints implying multiple clauses are hidden. The new clause regroups the information of at least two clauses, making it a new information for the solver eyes. Every clause is kept on a clauses database. This database gathers the clauses of the initial formula and the clauses learned during the search.

2.6.1 Conflict analysis

The conflict analysis is another key component of the solver. It will allow the solver to derive new clause and see new constraints between literals. There are several steps to derive the new clause. First of all, after the identification of the conflict, the implication graph needs to be computed. Then a cut must be found in that implication graph. Finally the new learned clause can be derived and added to the clauses database. After this, the solver will perform a non-chronological backtrack so that the backtrack point must be found as well. This is an overview of all the steps needed to perform the conflict analysis. I will now explain more in details how these steps are done.

Implication graph

To analyse the causes of a conflict, the solver will implicitly construct an *implication graph*. It is a directed graph that retraces the reasons that have led to the assignment and it is built as follows :

- Add a vertex for each decision literal.

- For each unit clause $(l_1 \vee l_2 \vee ... \vee l_k)$ asserting a literal l_k, add a vertex l_k if not already present in the graph and add edges from all literals l_i of the clause (with $i \neq k$) to l_k.

If both literals of a variable are present in the graph, there is a conflict, otherwise nothing needs to be done. Once the implication graph is built, an analysis can be done to determine the cause of the conflict and derive a new clause. In practice, the graph is implicitly built. Furthermore, it will only be built if we know in advance that there is a conflict.

In order to find a new clause, the implication graph must be split in two. A Bipartition Cut should be found such that all decision variables are on one side, called *Reason side*, and all conflicting literals are on the other, called *Conflict side* (illustrated on Figure 2.2). The remaining variables can be on the Reason side or on the Conflict side, the only constraint is that there should be no edge from the conflict side to the reason side. To derive the new clause, simply collect the literals from the reason side having an edge that crosses the cut.

There exist many such cuts, each cut leads to a different learned clause, so there can be multiple clauses derived from a unique implication graph. The clauses differ one from another by the fact that a literal l_i of the first clause is replaced by a subset of the literals asserting l_i in the second. Keeping all the clauses will lead to a storage problem. Since the knowledge gain is derived from this cut, which one should we choose ? Multiple techniques exist to select this cut. An interesting one is the First UIP (FUIP) which will now be explained.

First UIP

UIP stands for Unique Implication Point, this point is a vertex of the implication graph at the current decision level that dominates both vertices corresponding to the conflicting literals [21]. This point is the cause that has led to the occurrence of the conflict at the current Decision Level. The First UIP technique will thus search for the first such point in the graph.

This technique cuts the implication graph right before the first UIP encountered on the path starting from the conflict node and ending at the decision variable. This is illustrated on Figure 2.2 where the FUIP is the vertex x_4. This will minimize the number of nodes with edge crossing the cut on the path conflict nodes - decision variables. Notice that there might be other edges crossing the cut but they refer to *previous* decision variables, not to the decision variable of the current level.

This techniques is widely used, as in MiniSAT [5] and zChaff[22] for example.

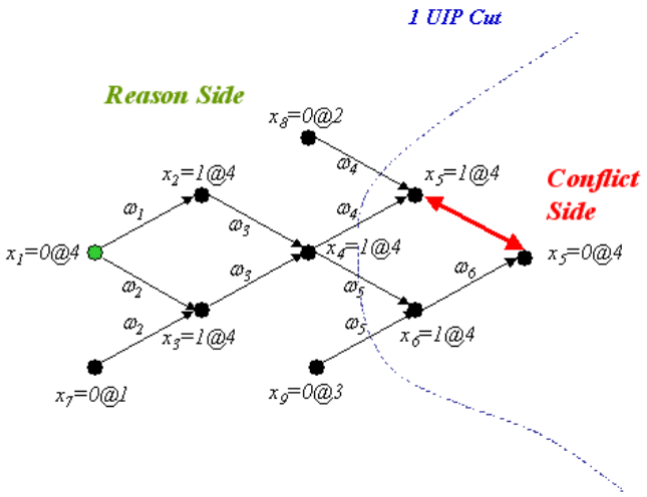

Figure 2.2: First UIP cut

Deriving the Conflict Clause

The new learned clause is the conflict clause derived by the cut of the implication graph when a conflict appears. This conflict clause is composed of all the nodes belonging to the reason side that have edges leading to the conflict side. These nodes are the reason of the conflict (for example x_4, $\overline{x_8}$, $\overline{x_9}$ in Fig 2.2). Therefore, these literals need to be negated in the new clause. The conflict clause is thus : $\neg(x_4 \wedge \overline{x_8} \wedge \overline{x_9})$ which corresponds to $(\overline{x_4} \vee x_8 \vee x_9)$ in CNF. The database is updated with the derived conflict clause and the search will continue with this new constraint. To take into account directly this new clause, a backtrack must be performed.

Backtracking on conflict

The backtrack is the action of undoing the last decision choice and thus the last variable assignments that were deduced from it are reverted. As we will see, in CDCL solver, this basic backtrack will not be performed this way due to the clause learning mechanism.

As we want to take into account the new clause directly, this will affect the backtracking process. Indeed, as the learned clause is fully assigned and UNSAT under the current assignment, we will backtrack until at least one of the literals becomes unassigned.

With the basic chronological backtracking (only undo the last level), the learned clause could still be fully assigned (and therefore unsat) if its variables were all assigned before the reverted level, resulting in a waste of resources. To overcome this behaviour, we need to perform a non-chronological backtrack, which means we need to backtrack **several levels** until at least one of the variables becomes unassigned. By doing so, the new clause is no long UNSAT under the current assignment which is a prerequisite to find a solution.

As said before, the purpose of all these techniques is to *reduce* the search space of the initial search tree. By backtracking over several levels (we are sure there is no solution on these levels), the effect of the non-chronological backtracking can be seen directly on the tree. The learned clause will also be useful in the remaining part of the search, it is thus stored in the clauses database.

2.6.2 Drawbacks of clause learning

After seeing the utility of learned clause and how they are derived, we will now see the drawbacks of the clauses learned.

Storage

SAT instances already have a relatively huge number of initial clauses called *problem clauses*. These clauses determine the instance and are stored in a database. For every conflict discovered, the solver learns a new clause and stores it in the database. The number of conflicts and thus of learned clauses can be very large. As the learned clauses are stored, this can quickly lead to *memory problem*.

Not only the number of learned clauses can cause problems, but also their size. As the search progresses, the learned clauses can become very large, due to the number of choices already made in the current assignment (decision variables). These two behaviours results in a huge space needed for the clauses database.

Computational inefficiency

The Boolean Constraint Propagation efficiency is linked with the number of clauses. To propagate a literal, the solver first needs to find all the clauses in the database that watch this literal. Then, for each of these clauses, a new unassigned literal must be found in order to replace the propagated literal. Therefore, as the number of clauses increases, the time for a unit propagation also increases. The BCP will take more and more time as the database grows. This behaviour needs to be minimized as the BCP is the bottleneck of the SAT Solver as discussed before.

Redundancy

As the number of clauses in the database grows, some clause might contain the **same information** as other clauses. For example the clause C : $(x_1 \vee x_2 \vee x_3 \vee x_4)$ is redundant with the clause $C_1 : (x_1 \vee x_2 \vee x_3)$ since C contains C_1. So if C_1 is satisfied, C is also satisfied. Redundant clauses are bad in term of storage and for computational efficiency reason as discussed before. This will be discussed later with the *learned clauses minimization*.

Conclusions

Keeping all the learned clauses is impractical in term of memory overhead, but also due to the excessive time spent in the propagation. An efficient database management is mandatory to have an efficient CDCL solver. The database management will be discussed later.

2.7 Search

The CDCL solvers will search for a complete assignment that satisfies the formula. This search is based on the DPLL algorithm and improved with BCP and early termination as discussed before. But there is another aspect of the DPLL algorithm that can be improved : the decision made on which variable to branch on next. This decision is taken when the BCP algorithm can not infer more assignments. After a decision, the BCP is run again until there are no more possible assignments and that cycle repeats again. To select this variable to branch on, there exist techniques called **Variable selection heuristic**.

2.8 Variable and Value selection heuristics

These heuristics are the core components for the decisions of the solver, decisions that will guide the search and determines the path taken by the solver. At each branching, the variable heuristic will select the variable to branch on. Then the value heuristic will select the value to assign (either True or False) to the selected variable.

Notice that the selected variable must be unassign. Therefore, both value are possible. If it was not the case and one value could not be selected, the last BCP would have detected it and the variable would have been assigned before the variable selection. The variable selection heuristic can be seen as an online sorting algorithm of variable scores, creating a rank of the most promising variables.

There exist many heuristics that are quite different one from another, some are based on inner properties whereas others are simply random or sequential. It is difficult to determine a good strategy because some of them will have a lower number of decisions but a higher number of BCP operations whereas others might have a lower number of BCP operations and a higher number of decisions. What really matters is which strategy is the fastest. No clear answer exist to this question. In this section, I will present the most common ones in the most popular solvers.

2.8.1 Variable selection heuristics

Each heuristic has its own strategy and generally focuses on the properties of the variables. It could be the number of clauses containing the variable, the variable number or its activity for example. The activity of a variable is in practice often used. The activity of a variable is defined by the number of times a clause containing that variable is used during the search to derive a learned clause. The activity reflects the utility of the variable to derive new clauses. We will now have a look at some of the state of the art heuristics for the Variable selection.

Random

This heuristic selects the next variable randomly among the available ones. Commonly denoted as RAND, this heuristic shows good results for the randomly generated formulas.

Variable State Independent Decaying Sum (VSIDS)

The VSIDS strategy was introduced in Chaff [23] and works as follows :

1. Each literal has a counter, initialized to 0.
2. When a clause is added to the database, the counters corresponding to the literals in the clause are incremented. This is also known as *bumping* the variables.
3. On a branching, the literal with the highest counter is chosen. Ties can be broken randomly (i.e. in Chaff). VSIDS is thus a Variable and Value selection strategy as it does not choose a variable but directly a literal.
4. All the counters are periodically divided by a constant. The period is often a fixed number of conflicts. This operation is known as *rescoring*.

Notice VSIDS has many meta-parameters to play with (Tie-breaker, period, constant).

VSIDS will thus select in priority the literal that appears the most in the database. Furthermore, with the constant division, VSIDS will favor the recently added clauses by giving them more weight.

One key property is the *independent* state, which means the counter does not take into account the actual value of the variable. It has a very low overhead because the increments occur only when there is a conflict.

The strategy shown above is the one presented in Chaff [23]. There are other variants as the one shown in BerkMin [6], they are not using the conflict clause but all the clauses responsible of the conflict to update the counters, improving the *sensitivity* of the decision heuristic. Another difference is that they add a counter for the variable and branch on the variable instead of the literal. BerkMin also reinforces the "young first" property by selecting the next free variable to assign directly among the free variables of the last conflict clause. This could be delayed in Chaff for a period of time when a variable switches value. They introduce another value selection heuristic based on the number of binary clauses in the neighborhood of the literal. This metric is a rough estimation of the power of BCP after assigning the literal to False. This technique is also present in PicoSAT [24].

EVSIDS

It is a slightly different heuristics than VSIDS. Instead of dividing all the scores after a period of time, EVSIDS (exponential VSIDS) [25] updates the score by adding an exponential factor g^i with g small and greater than 1 (typically $1.01 < g < 1.2$) and i the conflict index. However, this will grow exponentially and rescoring is still needed because the increment might cause an overflow problem.

SUM

SUM proposes an alternative to the rescoring scheme, instead of adding 1 to the variable score, SUM add the conflict-index. The conflict-index is the total number of conflicts that have occurred so far. This will favour recently learned clauses but the effect is smaller than in VSIDS and EVSIDS.

average conflict-index decision score (ACIDS)

As in the previous heuristics, ACIDS [25] also keeps a score for each literal. What changes is the score update. The update is done as follows $s' = (s + i)/2$, with i the conflict-index (the number of conflicts so far). Let's rewrite the variable score as s $= s_c + s_p$ with s_c the contribution of the current conflict and s_p the contribution of

the previous conflicts. For INC and SUM, most of the time, we have $s_p > s_c$ but in ACIDS, at any point in the search, we have $s_p < s_c$. This gives a larger weight to the current conflict than the previous ones.

Variable Move To Front

VMTF was initially introduced because the VSIDS heuristic had a main drawback. This drawback is that the periodic score decay is indirect, causing a delay. Until variable scores are updated, recent clauses are ignored. This is not a good behaviour as the newly learned clauses explicit a new constraint and tell why the last conflict appeared.

VMTF keeps a counter for each literal and an ordered list of variables. Variable v_0 precedes variable v_1 if $cnt(\bar{v_0}) + cnt(v_0) > cnt(\bar{v_1}) + cnt(v_1)$.

When a clause is learned during search, the respecting counters are incremented for each literal l in the clause. Then, some variables of the clause are moved to the front of the ordered list of variables. This number is a small constant (m), e.g. 8. If the clause contains less than m literals, all the variables are moved. The moved variables are positioned at the front of the list in an arbitrary order. When a decision must be taken, the first free variable (v) is selected. This variable is set to True if $r(v) > r(\overline{v})$, false if $r(\overline{v}) > r(v)$, and randomly otherwise [26].

The Figure 2.3 from [25] provides a summary of the variable heuristics. It shows the variable score updates when the variable is bumped. The second column shows the updates when the variable is not bumped. Only VSIDS and NVSIDS periodically update score without bump. h_i^{256} corresponds to the rescoring factor in the initial implementation in Chaff [23].

	variable score s' after i conflicts		
	bumped	not-bumped	
STATIC	s	s	static decision order
INC	$s+1$	s	increment scores
SUM	$s+i$	s	sum of conflict-indices
VSIDS	$h_i^{256} \cdot s + 1$	$h_i^{256} \cdot s$	original implementation in Chaff [5]
NVSIDS	$f \cdot s + (1-f)$	$f \cdot s$	normalized variant of VSIDS [6]
EVSIDS	$s + g^i$	s	exponential dual of NVSIDS [6,7]
ACIDS	$(s+i)/2$	s	average conflict-index decision scheme
VMTF	i	s	variable move-to-front [8]

Figure 2.3: Variable Heuristics score update comparison

2.9 Restarts

Restart is a mechanism to restart the search from the beginning, reverting every decisions made so far. This mechanism has no sense if the decisions made are exactly the sames than before the restart. But the solver keeps the same state as before the restart. It thus keeps the variable ordering list and the learned clauses, leading to a different order of decisions made. After a restart, the SAT solver can make better decisions. These decisions might be considered as better even if, in practice, often only the order of the decisions changes but not the variables themselves.

Initially, the restart was introduced due to an observed behaviour in the Industrial and structured instances. This behaviour is called the **Heavy-tail** behaviour. [27] On these instances, it was observed that, despite the fact that they are large instances, they could easily be satisfied or refuted. Therefore, if the solver has been stuck in a part of the search space for too long, there should be an easier solution elsewhere.

Several strategies have been developed, based on different observations and different metrics as random, number of conflicts or the last learned clause.

In this section, some of the restart strategies will be presented.

2.9.1 InOut

PicoSAT [24] for example has a very aggressive restart scheme, firing restarts with a high frequency. This technique is called InOut and is based on the number of conflicts encountered. This strategy keeps in memory 2 values. The first one is the minimum value, often set to an initial value of 100 conflicts. This value is the minimum number of conflicts needed to perform a restart. The second value is the maximum value. This is the maximum number of conflicts needed to perform a restart, also set to the initial value of 100.

When a number of conflicts since last restart is equals to the minimum value, a restart is performed. In addition to this, PicoSAT fixes the last learned clause, this clause cannot be deleted. This ensures the search will not follow the same path as before the restart. The minimum value is increased by 10% after a restart until the maximum value is reached. Once this maximum value is reached, the current value restarts from his initial value of 100 and the maximum value is increased by 10%.

Fig 2.4 from [24] provides a graphical overview of the InOut restart. The number of restarts is shown on the horizontal axis. The vertical axis shows the number of conflicts needed to trigger the next restart.

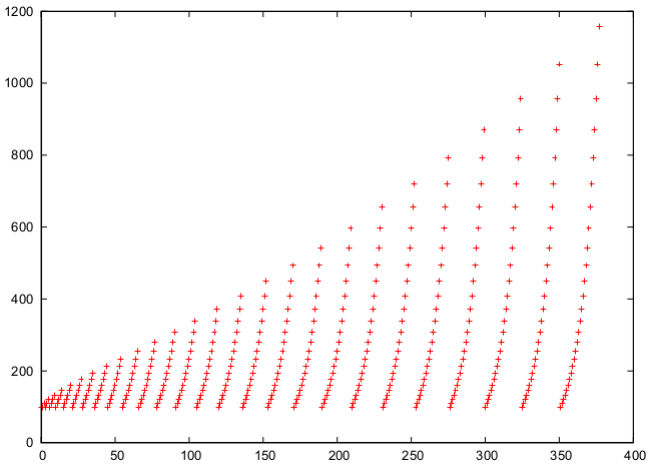

Figure 2.4: InOut restart scheme : number conflicts for next restart vs number restarts

The fast restarts were also introduced due to the observation of the Heavy-tail behaviour. Randomising the search should help the solver get out of these parts of the search space where refutation or a solution are not found fast enough.

2.9.2 Luby

In [28], Luby et al. describes an optimal (universal) strategy $S^{\text{univer}} = (t_1, t_2, t_3, \ldots)$ to restart the search. The strategy has a certain randomness in it but the convergence to a solution is still guaranteed given enough time. t_i is called a unit run length : it corresponds to the number of conflicts before a restart. To apply this strategy, first it waits for t_1 conflict, then restarts the search and lets t_2 new conflicts appear before the next restart and so on.

The optimal strategy looks like the following :

$$S^{\text{univer}} = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots)$$

Where the number corresponds to the number of conflicts before a restart. As performing a restart after only one conflict is a nonsense, the unit run length is thus used to shift a given quantity Q to obtain the number of conflicts before the restart.

$$\text{Next limit, } L_i = \text{Q} \ll t_i$$

More formally, the optimal strategy is obtained as follows :

$$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases}$$

with $i, k = 1, 2, \ldots$. Notice that all the run lengths t_i are powers of 2 and that each sequence is repeated before a new power of 2 is tried.

Let's see an example with an initial quantity of 100. With this initial quantity, the next limit can be very huge : $L_{31} = 100 << 16 = 6553600$ which is in practice so huge that the solver rarely exceeds 32 restarts.

2.9.3 Glucose

The glucose restart is based on a metric called *Literals Blocks Distance* or LBD. The LBD is defined as follows. Given a Clause C, partition the literals into n subsets according to the current assignment such that literals are partitioned with respect to their decision level. The LBD of clause C is exactly n.

Each Clause has thus its own LBD score, which is computed and stored when the clause is produced. It can be seen that learned clauses with an LBD of 2 are very important. They only contain one variable of the last decision level (they are FUIP). Later, this variable will be *glued* with the block of literals above, no matter the size of the clause. There are so important that they were named *glued clause*.

Another interesting property of LBD is that given a conflict graph, any first UIP asserting clause has the smallest LBD value over all other UIP's [7]. As we are looking for a clause of minimum LBD, FUIP is very interesting to derive new clause from the implication graph during the conflict analysis.

Now that the LBD value has been explained, the restart strategy can be explained. The idea behind the restart strategy is that we want the solver to produce as many good clauses as possible (w.r.t. their LBD score). When the last produced clauses have a too high LBD value, the search is considered on a bad path and a restart is performed. To this end, two LBD averages are compared

- The current average (avg_{cur}): Average of a sliding window of constant size on the LBD value of the last produced clauses.
- The global average (avg_{glo}): Average LBD of all the clauses produced so far.

Formally, a restart is performed if

$$avg_{current} * K > avg_{glo}$$

The magic constant K is called the margin ratio and provides different behaviours [7]. This restart strategy thus compares these two average scores to see if the solver is currently producing interesting clauses in term of LBD. If it is the case, the search keeps going on. But when it is no more the case, a restart is performed.

GLUCOSE waits that its window is full before comparing the average LBDs. A typical size for the window is 100. This prevents to aggressive restarts. Note that, unlike the strategies presented before which follow a sequence, we do not know in advance when the search will restart. This could lead to no restart at all if the LBD score of the produced clauses keeps decreasing.

2.10 Efficient DB Management

Ideally, it would be preferable to keep all the learned clauses in memory, this would maximize the constraints explicitly exposed during the search. But in practice, as the amount of learned clauses increases during the search, memory problems can appear if the number of clauses is not limited. Furthermore, if the amount of clauses increases, as the BCP efficiency is correlated with the number of clauses and BCP is the main component of today's CDCL solvers, the overall efficiency of the solver will suffer.

In conclusion, the solvers performances are tightly related to the number of clauses. This is why all the clauses cannot be kept in memory and an efficient database management is required. The purpose of the database management is to evaluate the quality of the learned clauses to delete the less important ones.

2.11 Predicting learned clauses quality

We need a metric to be able to manage the database correctly. Cleaning out too many clauses will break the learning benefit. Furthermore, deleting a good learned clause during the search can be very dramatic in practice. To avoid deleting good clause, we need to define what a "good clause" is. This is why we need to predict the quality of the learned clauses. Predicting the quality of a learned clause is one of the main concerns of state-of-the-art solvers [29]. Several metrics exist to predict the quality, some of them will be explained.

2.11.1 Bounded Learning

The bounded learning consists in bounding the database to reduce the memory overhead. There exist several types of bounded-learning. Two main types can be

distinguished. The first restricts the addition of clauses to the databases whereas the second discards clauses from the databases. A hybrid strategy can also be relevant.

Size-bounded learning Size-bounded learning is bounding the number of the learned clauses. This number cannot be greater than a certain number. This strategy is relevant since larger database is computationally expensive. The size changes during the search. In practice, it grows to prevent the good clauses from deletion.

An example of size-bounded learning is the Glucose solver. Glucose solver performs aggressive deletion of learned clause [7]. Glucose performs a database reduction every

$$I + S * R \text{ learned clauses.}$$

where I is an Initial value, S is a Scaling factor and R is the number of database Reduction performed so far. I and S are parameters and vary in the different versions of Glucose (typical values : 2000-20000 for I and 200-500 for S). A database reduction deletes half of the actual learned clauses, the number of learned clause is thus bounded by 2 times the current limit.

Relevance-bounded learning Relevance-bounded learning uses a metric to *discard* no longer relevant clauses according to the metric. The relevance is related to the quality of the clause.

A metric can be the length of the clause for example. Longer clauses are less likely to become unit as they are more literals to assign. Smaller clauses are thus favoured like clause of length 2 which only needs one literal assigned to become unit. Another metric could be the LBD value, this metric is similar to the length clause for the motivation and the analysis of the relevance. In fact, for a given clause C, we have :

$$2 <= LBD(C) <= Length(C)$$

The LBD value can be seen as the number of decisions needed for the clause to be fully assigned. As LBD is bounded by the length of the clause, it can be seen as a more refined metric than the length.

Another metric can be the clause activity. Has the clause been used recently during the search ? Is the clause a reason of many clauses ? Good learned clauses are identified by their recent usefulness during conflict analysis.

Relevance-bounded learning keeps clauses that are useful *locally* in the current search. During a database reduction, the less relevant clauses are discarded.

2.12 Phase saving

During the search, some solutions of sub-problems may be found before a conflict occurs, then a backtrack is performed and these solutions are lost in the process when the variable assignments are erased. Later in the search, the same sub-problems might appear and we need to restart the sub-problems from scratch. The same behaviour occurs with the restarts. Hardly acquired knowledge is lost and it results on a waste of resources as the solver solves multiple times the same sub-problems.

To overcome this problem, *phase saving* were introduced. The idea is quite simple and works well. Since the backtrack can erase partial solutions, we simply keep them elsewhere in memory. This requires an array called *saved-literal array*. Every time the solver performs a backtrack and erases assignments, each erased assignment is saved in the saved-literal array. This array is then used when a decision needs to be made. Once the variable is selected, the last value assigned to that variable before the last restart is present in the array and this value will be given to the variable. The phase saving heuristic is thus a kind of value selection heuristic, based on the last known value of the selected variable.

2.13 Pre/Inprocessing

More and more state-of-the-art CDCL SAT solvers, as PRECOSAT [30], CRYPTOMINISAT [31], and LINGELING [8], are using preprocessing and inprocessing. The principle behind the processing is to simplify the given formula, either before the actual satisfiability search (preprocessing) or during the search (inprocessing). To do so, solvers need to analyse the inner properties of the formula. They are analysing the structure of the formula [32] and the learned clauses [33] to develop new algorithms [34].

There exists many diverse formula simplification techniques. Most of the techniques are based on logical mathematics. They can focus on deleting clause ([35], [36], [37]) or deleting variable inside a clause ([35], [38]). These techniques have become an essential part of the SAT solving tool chain. Further than that, some of the current strongest SAT solvers are interleaving more and more formula simplification and CDCL search [39]. This adds more reasoning to the solver.

As many techniques exist [40], only the idea behind and the main techniques will now be presented.

2.13.1 Clause Deletion

The first category of formula simplification is the clause deletion. The idea behind the clause deletion is to delete the unnecessary clauses. Deleting these clauses is useful because it improves the solver performance and also reduces the database as discussed before. But we need to be careful because a clause can only be deleted if it preserves the solution. Some interesting clauses are the ones which hold no information (as the tautologies) or *redundant* information, meaning that their information is yet present in one or several clauses. A tautology is a clause which *always* evaluates to True. Once these clauses are spotted, they can be deleted safely and it preserves the solution.

Let's see an example with the following clauses.

$$C_1 = l_1 \vee l_2 \vee l_3, C_2 = l_1 \vee l_2 \vee l_4 \vee \overline{l_3},$$

$$C_3 = l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5, C_4 = l_3 \vee l_4 \vee l_2 \vee \overline{l_3}, C_5 = l_1 \vee l_2 \vee l_3$$

C_4 is a tautology as it contains the literal l_3 and its negation $\overline{l_3}$. No matter the value assigned to the third variable, the clause evaluates to True. C_5 is the same as C_1. C_3 is subsumed by C_1. A clause C subsumes another clause D if and only if $C \subset D$. As C_1 is a subset of C_3, if C_1 evaluates to True then C_3 will also evaluate to True.

All the Tautologies, the duplicated and the subsumed clauses can be removed immediately. So the given formula with 5 clauses can be reduced to 2 clauses, $C1$ and C_2.

Another technique which also reduces the number of clauses is the *self-Subsuming Resolution*.

Self-subsuming The self-subsuming works as follows. Let two clauses, $C_1 = C_3 \vee l$ and $C_2 = C_4 \vee \bar{l}$ such that C_3 and C_4 are CNF with $C_3 \subseteq C_4$, l is a variable. With the propositional logic, we have consecutively :

$$C_1 \wedge C_2 = C_3 \vee l \ \wedge \ C_4 \vee \bar{l} = C_3 \vee C_4 = C_4$$

where the second equality comes from the resolvent[2] and the last comes from the fact that $C_3 \subseteq C_4$.

With this technique on the small example above, the two remaining clauses can be reduced to a unique clause $C = l_1 \vee l_2 \vee l_4$. The initial number of clauses of the example is reduced from 5 to 1 with these techniques.

[2]Resolvent : $\frac{P \vee Q \vee R \quad P \vee \overline{R} \vee S}{P \vee Q \vee S}$

Algorithm 2: minimizeL(F, L): minimizing a set of learnt clauses

Input: F: A CNF formula; L: a set of learnt clauses such that $L \cap F = \emptyset$.
Output: A set of learnt clauses.

```
1  begin
2      foreach C = l_1 ∨ l_2 ∨ ··· ∨ l_k ∈ L do
3          if !liveClause(C) then continue;
4          L ← L \ {C}; C ← sort(C);
5          C' ← ∅; /* C' will be the minimized clause */
6          for i ← 1 to k do
7              if (R ← UP(F ∪ L ∪ C̄' ∪ {l̄_i})) = □ then
8                  C' ← conflAnalysis(F, L, C̄' ∪ {l̄_i}, R);
9                  break;
10             else if UP(F ∪ L ∪ C̄' ∪ {l_i}) ≠ □ then
11                 C' ← C' ∨ l_i; /* add l_i into clause C' */
12         L ← L ∪ {C'};
13     return L;
14 end
```

Figure 2.5: Learned clause minimization

2.13.2 Variable Elimination

Given a CNF formula F, variable elimination removes a variable v by replacing F_v and $F_{\overline{v}}$ by $F_v \otimes_v F_{\overline{v}}$. Where F_v is the set of clauses containing the literal v. After the variable elimination, the value v is completely removed from each clause of the formula F. The variable elimination can be done in several ways.

Let's see an example of one of these : the clause distribution [41].

$F_{\overline{x}} \setminus F_x$	$(x \vee c)$	$(x \vee \overline{d})$	$(x \vee \overline{a} \vee \overline{b})$
$(\overline{x} \vee a)$	$(a \vee c)$	$(a \vee \overline{d})$	$(a \vee \overline{a} \vee \overline{b})$
$(\overline{x} \vee b)$	$(b \vee c)$	$(b \vee \overline{d})$	$(b \vee \overline{a} \vee \overline{b})$
$(\overline{x} \vee \overline{e} \vee f)$	$(c \vee \overline{e} \vee f)$	$(\overline{d} \vee \overline{e} \vee f)$	$(\overline{a} \vee \overline{b} \vee \overline{e} \vee f)$

We have $|F_v| * |F_{\overline{v}}| = |F_v \otimes_v F_{\overline{v}}|$ which *increases* the number of clauses ! This is a drawback even if some clauses can be discarded (the red ones in the example) as they are tautologies. The Variable Elimination lowers the number of variables and thus the number of decisions during the search but increases the number of clauses.

2.13.3 Learned clause minimization

The learned clause minimization consists on minimizing the learned clauses in order to eliminate the useless literals in the clause. To do so, we can use an algorithm as the one proposed in [42].

The algorithm is shown on Figure 2.5. This algorithm iterates over all the learned clauses that are said to be *live*. These clauses are determined by their empirical characteristics. [42] and [43] showed empirically that high LBD learned clauses are not very useful to solve practical SAT instances. Indeed, the LBD of a learned clause is correlated to the minimum number of decisions needed to make the clause failed, meaning that it is much easier to reduce the size of a learned clause than to reduce its LBD. Moreover, clauses with high LBD has generally huge size, needing more unit propagations to be minimized. These are the two parameters (size and LBD) used to determine if a clause is worth minimizing. In practice, a threshold is chosen for these two parameters, typical values are 6 for the LBD and 30 for the size.

This algorithm builds incrementally the minimized learned clause, by looking, for each live clause, at the utility of each literal. The algorithm is based on several principles :

1. Let $C = l_1 \vee l_2 \vee ... \vee l_k$ be a live clause learned during the search of size k. If propagating the literals $\overline{l_1}, \overline{l_2}, ..., \overline{l_i}$ with $i < k$ via unit propagation (with all the clauses and without C) causes the problem to be unsat, then the clause $C' = l_1 \vee l_2 \vee ... \vee l_i$ is a logical consequence of all the clauses and subsumes C, it can thus be added without affecting satisfiability. The literals in C that have this property will be collected : its negation has a path to the conflict clause in the implication graph. These collected literals are a new clause that subsumes C' and thus C.

2. Let $C = l_1 \vee l_2 \vee ... \vee l_k$ be a live clause learned during the search of size k. If propagating the literals $\overline{l_1}, ..., \overline{l_{i-1}}, l_i$ with $i < k$ via unit propagation (with all the clauses and without C) causes the problem to be unsat, then the clause $C'' = l_1 \vee ... \vee l_{i-1} \vee \overline{l_i}$ is a logical consequence of all the clauses. The resolvent of C'' and C does not contain the literal l_i and subsumes C. So the algorithm does not add l_i in the minimized clause. In practice, the opposite is done (see line 10).

It is built incrementally in practice for efficiency reasons. We can also check if the current literal l_i is already assigned before propagating. This algorithm should not be used too often and the learned clauses should only be minimized once to avoid a too high computational cost of the LCM algorithm.

Conclusion The preprocessing and inprocessing techniques add logical and mathematical reasoning to DPLL-based search. The effects are to reduce the number of clauses, their size or even the number of variables. All the effects aim at reducing the search space or improve the efficiency of the main algorithms. All the techniques

affect the computation time and possibly have a drawback as the Variable Elimination. But they can come up with a gain with the clauses/variable/search-space prune.

Chapter 3

Rsolve

Rsolve is a SAT solver written in Rust. I used this solver as a base for my implementations. In a first time, I needed to understand how the solver works, I will explain it a bit here. After this first step, I have added other techniques to the solver and finally, I will use different variantes of this solver for the comparative study of next section. All the techniques in the solver are explained in the Background Chapter.

3.1 Rust

This solver is written in Rust, Rust is a language designed to be memory safe. It will always look at it. You can have several immutable references to a memory zone. Or you can have a unique mutable reference to that zone and no immutable reference to ensure that the information of an immutable reference does not change because of the mutable reference. This is checked at compile time. It allows to get rid of this at run time.

Toolbox This solver is not designed to be a highly performing solver but to be a *Toolbox*. By toolbox, I mean that it contains several tools as resolution techniques, learning schemes, heuristics, preprocessing and inprocessing techniques. One can pick up a tool or another one interchangeably, the solver is very *modular*. This is the key property of the solver, it is designed with some trait to be able to easily change the solver, like with the *BranchingHeuristic* and the *RestartHeuristic* trait for the variable selection and the restart strategy.

3.2 Implementation

I will explain the implementation of Rsolve. First of all, I will talk about the core components of the solver. Then I will explain how the different techniques discussed in the theoretical background are implemented in practice.

3.2.1 Core of the solver

I will explain the implementation of the keys components of the solver. First of all, let's look at the bases of all solvers. The most important structure in a CDCL solver is the clause. To define a clause, we first need some lower structures. Here are the different structures needed in order to represent and use a clause correctly.

1. Variable : Each variable is identified by an unsigned number.

2. Literal : A type that wraps around a signed number that represents a literal. The number corresponds to the related variable. The sign corresponds to the literal of the variable : negative for the negative literal and positive for the positive literal.

3. Clause : A vector of literal. Another field is there to know if the clause was learned during the search or if it is a problem clause. If a learned clause or a problem clause is unit, the clause is not stored in the database, the literal is simply assigned and marked as forced.

The rest of the solver uses these core structures in order to solve the instances.

Now let's look at the solver itself. It needs several information in order to work properly. The solver itself is also a structure with different fields.

1. clauses : A vector of clauses, containing initially the problem clauses. It also contains all the learned clauses during the search. The size of the vector is limited to prevent the memory explosion problem (memory management). This limit can grow as discussed before.

2. lbd : A vector of integer, indicating the LBD of the corresponding clause, see Background section. The index of the clause vector and lbd vector is the same for a given clause.

3. watchers : A vector, indexed by literal. Each entry contains a vector. The indexes of the clauses that watch this literal are present in this vector.

4. propagation queue : A queue containing all the literals that have been falsified up to the current point of the search. Illustrated on Figure 3.1.

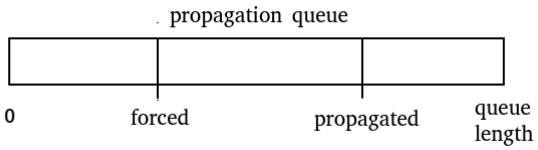

Figure 3.1: Propagation queue

With this queue, two indexes come up. The first is the *forced* index : the index up to which all the literals in the queue are direct consequences of the problem definition. It means that these literals must be false in every solution (if one exists), and thus should never be reverted during the search. The value of each literal is fixed and said to be forced by the problem.

The second index is the *propagated* index : it is the index up to which all literals have been propagated through an unit propagation at the current point of the search. The literals between the *propagated* index and the end of the queue still need to go through the unit propagation, meaning that the consequences of their assignment are not yet known. This is shown in 3.1.

5. reason : a vector containing the reason associated with each assignment. This vector is indexed by the variable. A reason is the unit clause that has led to the variable assignment. An assignment can come from two different situations. The first one is when a clause becomes unit, the last literal remaining should be assigned and the reason is that particular clause. The second situation is when the solver makes a decision. It will choose a variable with the variable selection. As the assignment does not come from a clause but a decision, there is no reason to that assignment.

6. flags : these flags are used during the conflict analysis in order to derived the learned clause. One set of flag is associated with each literal. Here are the different flags : *Clear* to reset all flags, *IsForced* to know if the literal is Forced by the problem. *IsMarked* if the literal appears in the implication graph. *IsImplied* if the literal is forced by the ongoing conflict clause. *IsNotImplied* if the literal has yet been analysed and is not forced. Finally, *IsInConflictClause* in order to find easily the backjump point.

The main function consists on a loop and continues until a solution or a refutation is found. The loop works as follows : the newly assigned literals available (if any) are propagated through BCP. If there is no conflict, a decision is made and the BCP is run again. If there is a conflict, several things are done :

1. A conflict analysis is run : The conflict is resolved. A new learned clause is derived and added to the database and a backtrack is performed.

2. The solver checks if a restart must be performed given the restart strategy. If it is the case, the restart is performed.

3. The solver looks if the database needs to be reduced to avoid memory problem. Rsolve use the database management of Glucose which bounds the size of the database as discussed before.

3.3 Conflict Analysis

One of the key component is the conflict analysis. Rsolve implements the first-UIP learning scheme as presented before with recursive minimization.

To retrieve the learned clause, the solver travels the propagation queue from the end to the begin. The literals are thus traveled in newly assigned order. All the marked literals that are not implied are collected. The marked literals that are implied should be in the learned clause with only the first-UIP scheme. But here, there is a minimization with the first-UIP scheme. This minimization detects and removes the implied literals. These literals are implied by others which remain in the clause. The first marked literal not implied collected will thus be the latest assigned of the learned clause. The solver needs to backtrack *at least* on this point. In practice, it will rollback more as we want to backtrack to the smallest level possible where the new learned clause is unit. It will thus search for the second literal marked as not implied and look at its level L. Then a backtrack is performed just after the level L. As on level L-1, the clause is not unit and L+1 is not the smallest possible level where the clause is unit.

3.4 Variable selection heuristics

In the toolbox, there is some variables selection heuristics. They all implement a trait named *BranchingHeuristic* defined to generalise the variable selection heuristics. Each branching heuristic simply needs to implement this trait to work in the solver. This makes it quite easy to implement new Branching heuristic and use them in the solver. The trait works as an adaptive binary heap. Each variable has a score. The higher the score, the faster that variable will be picked up for the next branching. The scores can be updated and the variable swims or sinks accordingly in the heap to remain ordered.

The trait has the following functions :

```rust
/// Abstraction of a variable selection heuristic.
pub trait BranchingHeuristic {
    /// Creates a new Branching heuristic capable of dealing with `capa` variables.
    fn new(capa: usize) -> Self;

    /// return true iff there is no element left in the heap
    fn is_empty(&self) -> bool;

    /// (Optional) Updates the variable's score according to the implemented heuristic
    fn bump(&mut self, var: Variable);

    /// (Optional) Ages the score of all the variables to make them appear less relevant
    fn decay(&mut self);

    /// Places the given `var` back in the heap (if not already present)
    ///
    /// # Panics
    /// - if the given variable does not fit in the range [1 .. capa]
    fn push_back(&mut self, var: Variable);

    /// Removes the element with highest score from the heap and returns it.
    ///
    /// # Return Value
    /// Returns the element with highest score on the heap.
    ///
    /// # Panics
    /// - when one tries to pop an empty heap.
    fn pop_top(&mut self) -> Variable;
}
```

Only the bump and decay functions are optional. These functions are not needed for the Naive and Random heuristics for example because the variable scores should not be updated. All the others functions are mandatory as they are used for the heap to work. On the opposite, ACIDS and VSIDS both use these two functions to update the scores during the search. So we can split in two categories : on one side, ACIDS and VSIDS who update a variable score each time the variable is bumped and on the other side, Naive and Random which do not update the variable scores. Each heuristics is using the heap except Random, ACIDS/VSIDS with the actual score during the search and Naive/Random with the initial variable

score. For the Naive heuristic, the score is initialised as follows :

$$\text{score of } v_i = N - i$$

, where N is the number of variables. For the Random heuristic, all the variables have always the same score.

ACIDS, VSIDS, Naive and Random are the four decision heuristics available in the solver. They are separated in updating score heuristics and not-updating score (static) heuristics and will be used during the experiments.

3.5 Restarts heuristics

There is also a trait for the restart heuristics. This trait only contains two functions. The first function is called `should_restart` and tells whether the solver should restart given a metric (number of conflicts or average LBD). The second function is called `set_next_limit` and set the next conflict limit to trigger a restart. Once again, all heuristics do not need all the functions. The Glucose restart, for example, depends on the current average LBD so no limit can be put beforehand, while Luby and InOut follow a predefined sequence. Luby and InOut are based on the number of conflicts to set the next limit. Whereas Glucose is based on the quality of the last clauses based on the LBD score. Therefore, for InOut and Luby, only the number of conflicts need to be provided. For Glucose, the LBD scores of the clauses currently in the glucose window need to be averaged. Luby and InOut do not use clause intrinsic information but Glucose does.

Glucose, Luby and InOut are the three restarts heuristics available in the solver and will be used during the experiments.

3.6 Pre/Inprocessing

For the preprocessing and inprocessing techniques, booleans are available on the solver state to enable/disable the different techniques. They are heavy in term of computation time. So they are triggered along with the restart.

3.6.1 Subsumption

In the solver, the subsumption is available. There are two main kinds of subsumption : the backward and forward subsumptions. They are both available. Here are the pseudo-code of the two.

```
1 //Forward subsumption
```

```
For each clause C in formula F do
    if C is subsumed by a clause D in F\C
        remove C from F

//Backward subsumption
foreach clause C in formula F do
    remove all clauses D in F\C that are subsumed by C
```

As this is time consuming with the current clause representation, I tried to change the representation of the clauses in the solver by a bitset. With a bitset, each literal has its own bit which is set if the literal is present in the clause and unset otherwise. With the bitsets, the subsumption can be easily detected. But it turns out that, in practice, the memory explodes. Indeed for each clause, each possible literal (2 times the number of variables) has its own bit. As the number of variables is huge, plus the initial number of clauses and the learned clauses, it was too much. With an instance of 200 000 variables and 100 000 clauses, there are 200 000 * 2 * 100 000 = 40 000 000 000 bits needed. This is 5 GB only for the initial clauses.

The self-subsuming resolution is also available.

3.6.2 LCM

The learned clauses minimisation, as explained in the Background Chapter, is available in Rsolve. As the LCM algorithm is heavy, the learned clauses will only be minimized once. Also, they are minimized only if they are *live* as discussed before. The live property is in practice determined by the size and the LBD score of the clause. If the clause size is below 30 and the LBD score is below 6, the clause is worth investing some time to be minimized. These thresholds are typical values used in modern solvers [29].

Chapter 4

Comparative study

Before the comparative study, we will first see how the instances are built. The instances follow a specific format, this format is called DIMACS. After that, we will see an example of a problem statement and associated instances.

4.1 DIMACS

DIMACS format is a standard interface to SAT solvers. Many SAT solvers accept this common format as input and output. Rsolve is one of them. This format is used in SAT solving competitions.

The file format is defined as follows :

- The file can start with comments : comments are lines beginning with the character c.
- After the comments, there is a line "p cnf nbvar nbclauses", starting with the character p, indicating that the instance is in CNF format. Nbvar is the exact number of variables appearing in the file and nbclauses is the exact number of clauses contained in the file.
- Then the clauses follow. Each clause is a sequence of distinct non-zero numbers between -nbvar and nbvar. The line ends with 0. It cannot contain the opposite literals i and -i simultaneously (or the clause is a tautology). Positive numbers denote the positive literals of the corresponding variables. Negative numbers denote the negative literals of the corresponding variables.

This kind of file format is very used by CDCL solvers. It allows to directly access the clauses and work with them.

4.2 Instances types

The real-life applications of the sat solvers are numerous. The most known are circuit verification [2], bound model checking for hardware [44], planning problems [45],... which are industrial problems. But SAT has also randomly generated problems, for example in cryptography. Every year, the SAT competition has a track for the industrial problems and another one for the random problems. Furthermore, each year, new and larger problems appear in the competition. Each problem has its own specificity, definition and meta-parameters. In the SAT competition, a description of the problem and instances needs to be provided. A problem can generate several instances, for example by changing an inner meta-parameter. We will now see an example of such a problem. Several instances of this problem will be used in the study.

4.2.1 snw instances

These instances were proposed by Thorsten Ehlers and Dirk Nowotka in the 2016 SAT competition [46]. Their benchmarks are based on a propositional encoding of sorting networks.

Sorting networks are a representation of data-oblivious sorting algorithms. These algorithms perform a sequence of comparisons which only depends on the length of the input, and is independent of the actual data to sort. Figure 4.1 shows a sorting network for 5 inputs. The horizontal lines, denoted channels, transport values from the left-hand side to the right-hand side. Whenever two of them are connected by a vertical line, denoted comparator, the values on the channels are compared and swapped, if necessary. The depth of a sorting network is the number of parallel sorting steps required to sort every input.

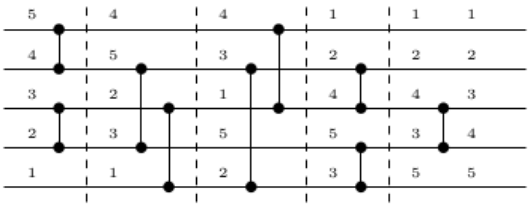

Figure 4.1: A sorting network on 5 channels, sorting the input (5, 4, 3, 2, 1).

Their names are `snw_n_d` where n denotes the number of channels, and d the number of layers.

In addition to those two parameters, some others parameters exist, corresponding to some preprocessing of the initial formulas. The preprocessing used is defined by the following infixes (added to the instance name).

- CCS : Symmetry breaks due to Codish, Cruz-Filipe and Shneider-Kamp. [47]
- Enc : reduces the number of clauses and improved encoding which allows for more propagations. [48]
- preOpt : Minimizes the number of variables in the encoding with symmetries. [48]
- pre : a preprocessing step has been applied to the formula. [48]

The instances created thus vary with the n and d meta-parameters. For each instance, one or no preprocessing was used.

4.3 Methodology

To get a clear idea of the performance of all the implemented techniques in the toolbox of rsolve and deliver results which are representative, a set of 44 instances has been created. These instances come from the main track of the 2016 SAT competition [46]. These instances have been selected from different submitters and are from different problems. From the toolbox, many solvers were created. Each solver differs from at least one technique presented in the previous sections. I will analyse the following techniques:

- Decision variable : ACIDS, VSIDS, Random, Naive.
- Restart heuristics : InOut, Glucose, Luby.
- Other techniques (inprocessing and preprocessing) : Learned Clause Minimisation, Preprocessing, Remove Literals (RL) (remove clauses with forced literals).

Every combination of decision variable and restart heuristic has been created (12 combinations). Then, from these 12 solvers, some variants have been produced with the other techniques. Solvers were given a timeout of 45 minutes : if the instance is not solved after 45 minutes, the solver is stopped and a timeout is reported. For each instance, the execution time of each solver was measured using the system time of the test machine. The whole process has been reproduced five times and the results were averaged to reduce the potential variation from a run to another.

For the experiments, the instances are separated into different categories, based on some metrics.

The first metric is the initial number of clauses. The number of clauses determines the size of the instance. As discussed before, the performance of the main algorithms is based on the number of clauses. It will be interesting to see if the initial number of clauses (without the learned clauses) has an influence. Fig 4.2 shows the distinction between three different categories :

- Instances of small size with initial number of clauses less than $2 * 10^5$. Represented in Green.
- Intermediate instances between $2 * 10^5$ and 10^6 clauses. Represented in Blue.
- Big instances with more than 10^6 clauses. Represented in Red.

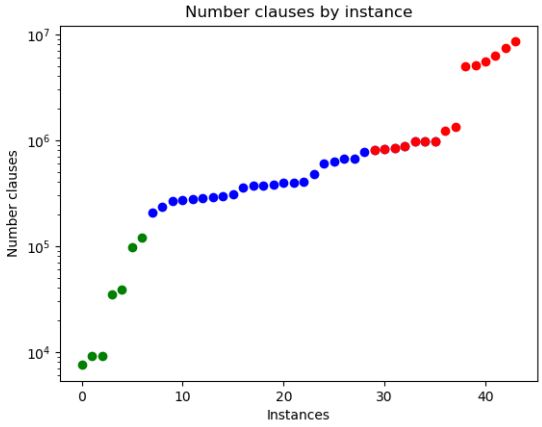

Figure 4.2: Number of clauses by instance. In Green the small instances with less than $2 * 10^5$ clauses. In Red the big instances over 1e6 clauses and in Blue, the intermediate instances (LOG scale)

The second metric is the average size of a clause. The size of a clause, is an upper-bound of the number of assignments needed to satisfy the clause (or -1 for the clause to become unit) which makes it an interesting metric.

With this metric, the instances are separated in two groups well distinguishable, illustrated on Fig 4.3. The first group are the instances with average clause size less than 5, shown in Blue and the second group has an average size over 40, in Red on the figure.

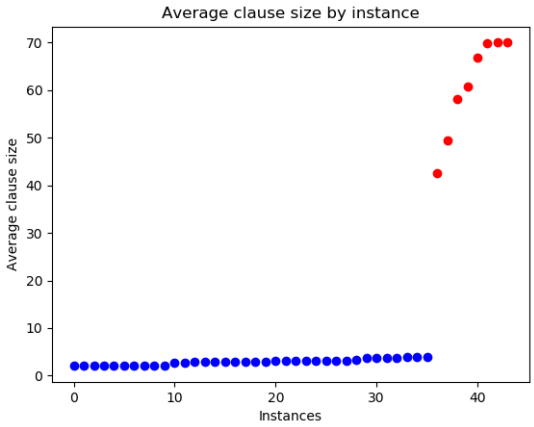

Figure 4.3: Average clause size by instance. In Blue, the instances with average clause size less than 5. In Red, the instances with average clause size over 40.

4.4 Results

The results, observations and analysis of the experiments will be provided in this section. Each subsection details the type of technique compared, the observations done based on the metrics and the analysis. The figures will show the general behaviour of all the compared solvers. The instances will be ordered by time for each technique to be able to see its general behaviour. Without this, the plots would have been a cloud of points. Nevertheless, the big instances, as they are less, will not be ordered by time. None of these solvers performs better than another on ALL the instances, this is why the general behaviour is shown here.

4.4.1 Variable selection heuristics

The first comparison is done with the variable selection heuristics. Four different heuristics are available : Naive, Random, ACIDS and VSIDS. The two first heuristics do not use inner properties to pick the next variable to branch on. This fact can clearly be seen on the results. The Naive heuristic is shown on Fig 4.4. The difference with the base solver (VSIDS) is clear, the naive heuristic is not performing well at all and is able to solve less than 5 instances. This illustrates the power of the variable heuristics which actually use metric to pick the next variable to branch on.

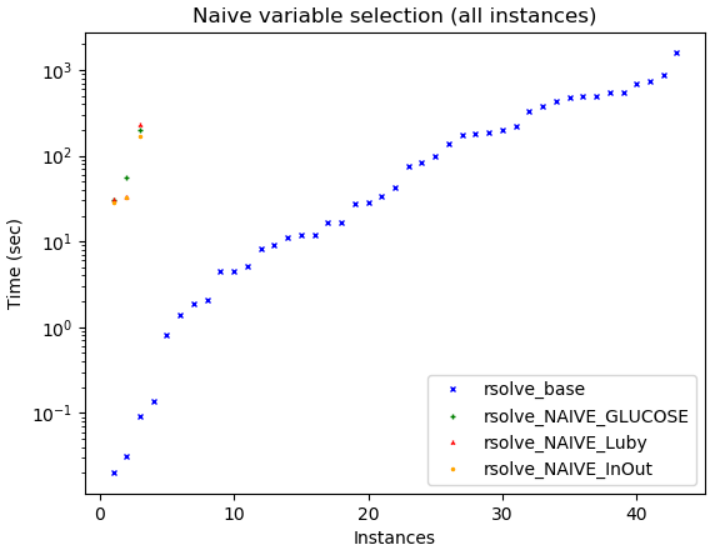

Figure 4.4: Naive selection heuristic on all instances

The same behaviour occurs for the Random heuristic, illustrated on Fig 4.5. It is able to solve less than 5 instances. But the Random heuristic performs well on random and cryptography instances, comes up with a solution whereas ACIDS and VSIDS heuristics timeout.

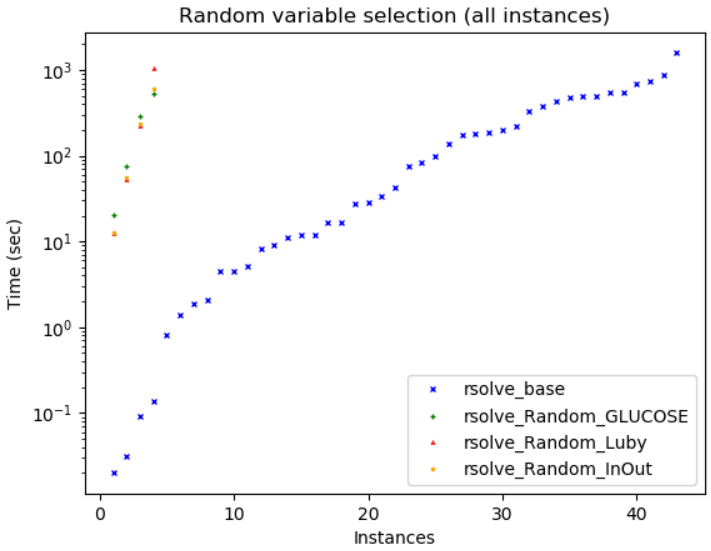

Figure 4.5: Random selection heuristic on all instances

As it is difficult to compare a timeout with a non-timeout, the Naive and Random heuristics will not be shown in the rest of the experiment. There are still shown here to illustrate the power of a branching heuristic which uses a metric (VSIDS and ACIDS vs Naive). Even if Random heuristic does not perform well on industrial instances, this heuristic is still interesting on Cryptography and randomly generated instances.

Table 4.1 shows the global behaviour of the ACIDS and VSIDS heuristics.

Solver	solved instances	Total time (sec)	Average time (sec)
`ACIDS_Glucose`	41	8292	202
`ACIDS_InOut`	43	25398	590
`ACIDS_Luby`	40	13457	336
`VSIDS_Glucose`	42	6512	125
`VSIDS_InOut`	42	21939	522
`VSIDS_Luby`	42	11263	268

Table 4.1: Global results of ACIDS and VSIDS

It seems that, globally, the VSIDS heuristic is performing better than the ACIDS heuristic. Let's examine the results more in details. Figures 4.6, 4.7 and

4.8, respectively show the variable selection on the small instances, intermediate instances and big instances.

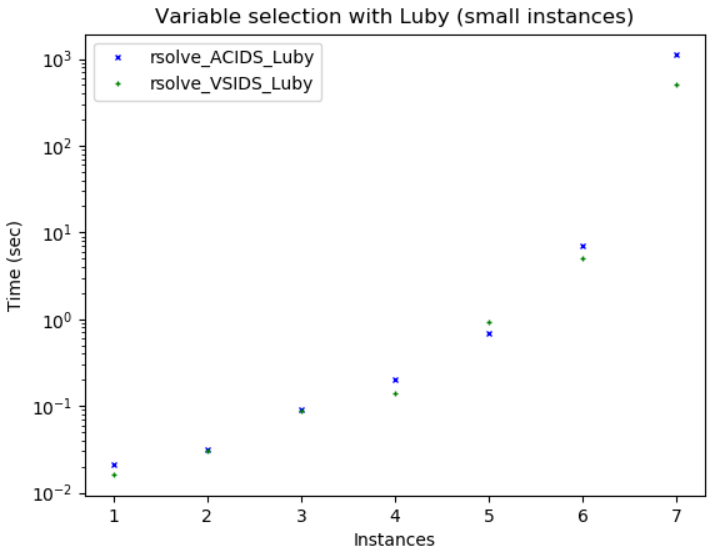

Figure 4.6: Time on instances of small size with Luby : Variable selection comparison (LOG scale)

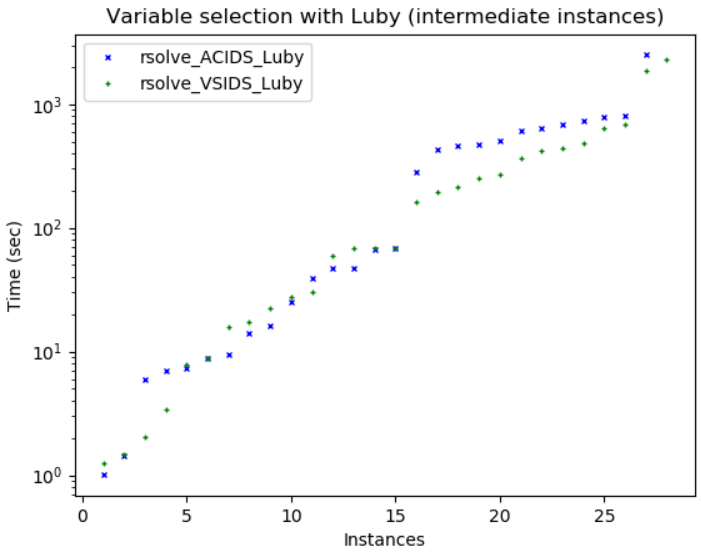

Figure 4.7: Time on instances of intermediate size with Luby : Variable selection comparison (LOG scale)

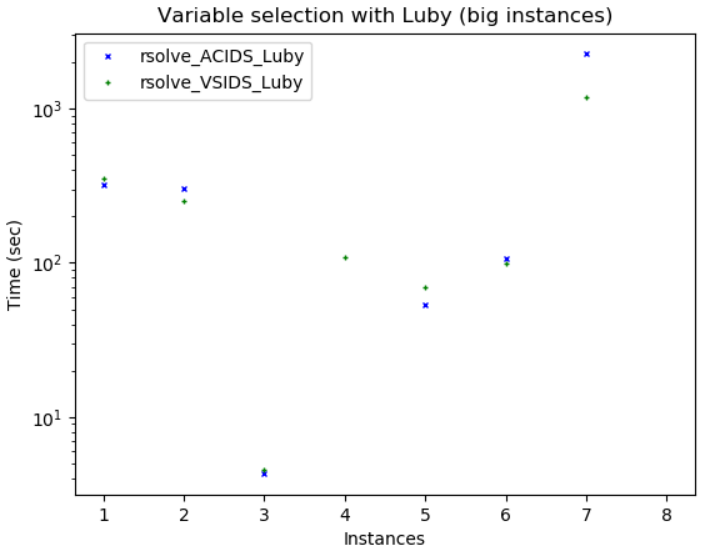

Figure 4.8: Time on instances of big size with Luby : Variable selection comparison (LOG scale)

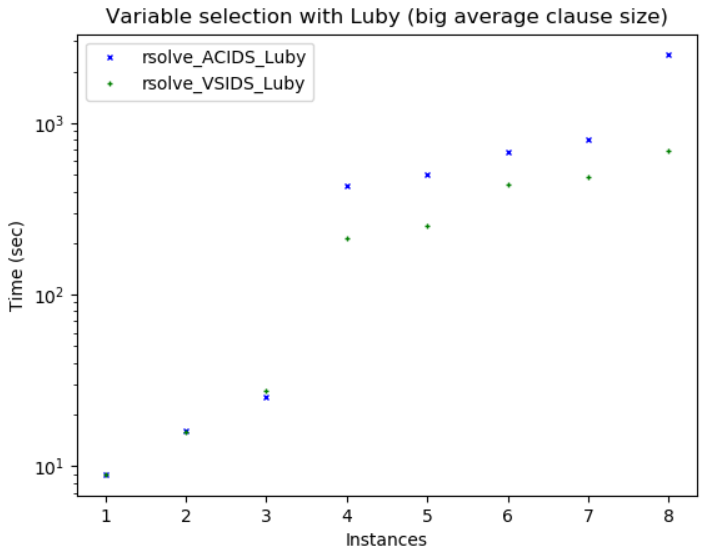

Figure 4.9: Time on instances of big average clause size with Luby : Variable selection comparison (LOG scale)

Observations

- For the small instances Fig 4.6 , the two selection heuristics present nearly the same behaviour. No clear domination can be seen.
- On big instances (Figure 4.8, not ordered by time), the same observation is done as for the small instances.
- For the intermediate instance size (Fig 4.7), there is a difference between the two variable selections, where VSIDS performs globally better than ACIDS. It is interesting to see that it is the intermediate size that changes and not the small or big size.
- Figure 4.9 shows the instances with big average clause size. The difference between ACIDS and VSIDS is more visible. On big average clause size, for the instances requiring at least 100 seconds, VSIDS performs better than ACIDS.

VSIDS performs thus better than ACIDS on intermediate instances and when the instance has big average clause size requiring at least 100 seconds. For the other instances, their performances are globally the same.

4.4.2 Restart strategies

The restart strategies will be compared with respect to the instance size. Then a summary table will show the global behaviour on all instances.

Total time

The total time of the solver is only composed of the time of the search. During the experiment, the ACIDS variable selection heuristic has been selected. Figures 4.10, 4.11 and 4.12 show the restart techniques on respectively the small instances, intermediate instances and big instances. For each solver, the instances are sorted by total time. Like this, the general behaviour is illustrated. Warning : it is not because a solver is always lower in the plot than another solver that is it the case for every instance. These figures show the general behaviour.

Observations and analysis

- For the small instances, illustrated on Fig 4.10 , none of the strategies dominates the others, they are nearly equivalent.
- For the intermediate instance size (Fig 4.11), the InOut strategy (red marker) is outperformed by the other strategies. It could be explained by the too aggressive restart strategy of InOut. Fig 4.13 illustrates that InOut performs much more restarts than the two other restart strategies. Note that, on the same figure, we clearly see the steps of the Luby strategy corresponding to the Luby sequence. These steps show that the current next limit is huge (huge number of conflicts needed) before the next step in the sequence. When this limit is reached, as the sequence restarts, a huge number of restarts is then performed. On most of the instances, the Luby and Glucose strategies behave equivalently. There is only a few instances where the Glucose is better.
- On big instances (Figure 4.16, not ordered by time), the same observation is done for the InOut strategy. For the Luby and Glucose, there are more distinguishable : the Glucose strategy is performing better.

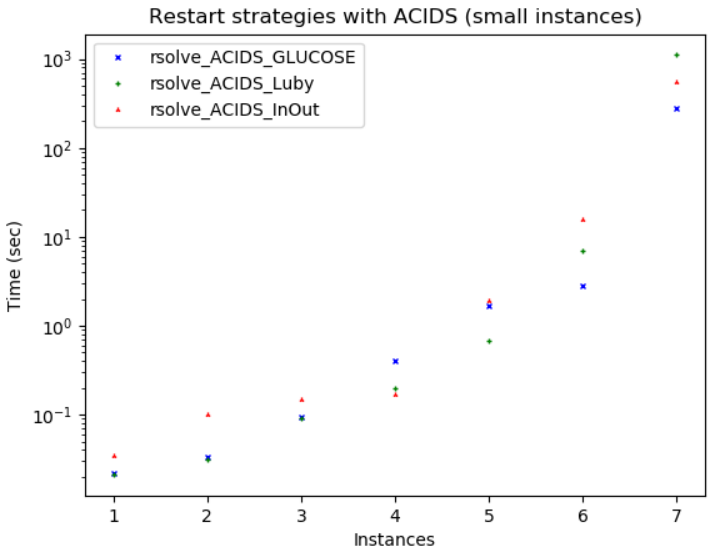

Figure 4.10: Time on instances of small size with ACIDS : Restarts strategies comparison (LOG scale)

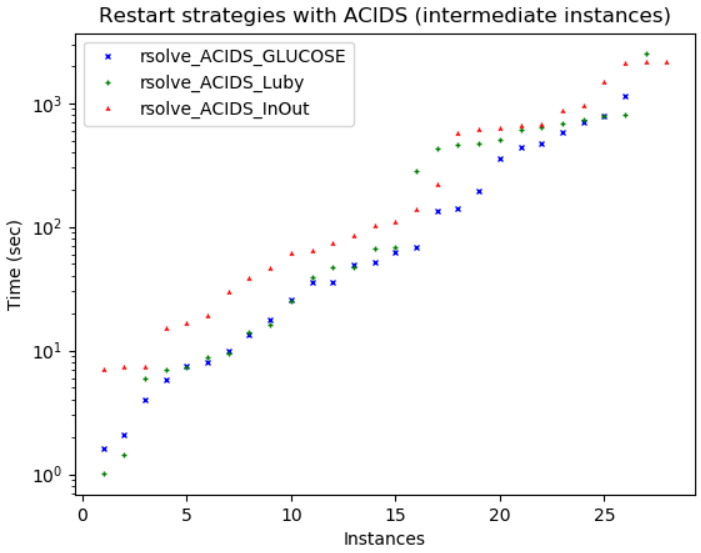

Figure 4.11: Time on instances of intermediate size with ACIDS : Restarts strategies comparison (LOG scale)

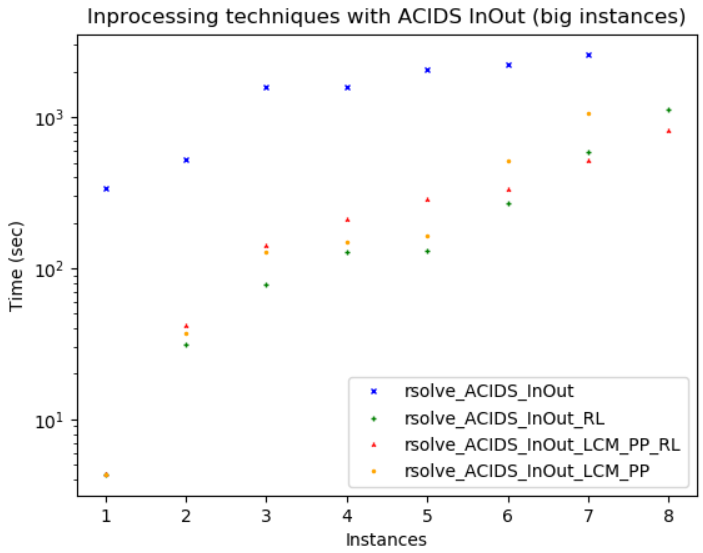

Figure 4.12: Time on instances of big size with ACIDS : Restarts strategies comparison (LOG scale)

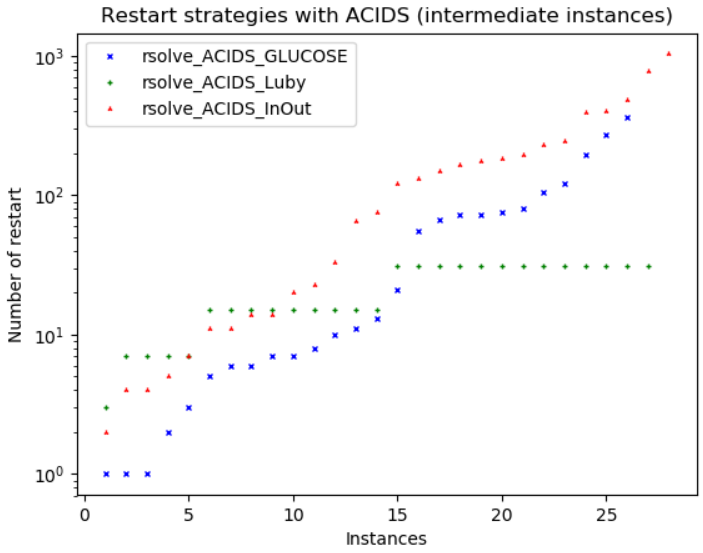

Figure 4.13: Number of restarts performed on instances of intermediate size with ACIDS : Restarts strategies comparison (LOG scale)

4.4.3 Preprocessing and inprocessing

With the preprocessing and inprocessing, the solver should be more *guided* by interleaving search and the different techniques. These techniques are there to reduce the size of the formula and eventually, the complexity of the formula. I will show the total time on the instances and see if there is an impact of these techniques on the instances.

Total time The total time of the solver is the time of the search plus the time used by the different techniques. For the experiment, the same variable heuristic (ACIDS) and restart (InOut) are kept constant. We will see the comparison of the techniques based on the size of the instances. These comparisons are illustrated on Figures 4.14, 4.15 and 4.16. As before, for each solver, the instances are sorted with respect to the elapsed time.

Observations and analysis

- For the small instances Fig 4.14 , there is a little gain with the inprocessing but not a significant one. Yet we can see that the solvers with inprocessing techniques seem to perform slightly better.
- For the intermediate instances (Fig 4.15), the RL solver (green marker) clearly outperforms the solver without inprocessing. It shows that simply removing the clauses with a forced literal has a huge impact on the performance. The two solvers with LCM and preprocessing take less time at first and then are equivalent to the solver without inprocessing. It can be explained by the fact that LCM is more time consuming that RL.
- On big instances (Figure 4.16), the gain is clear. Every solver with inprocessing outperforms the solver without inprocessing on every instance. The same observation can be done with the high average clause size on Fig 4.17. This illustrates the utility of the clause minimization and clause removal on big instances.

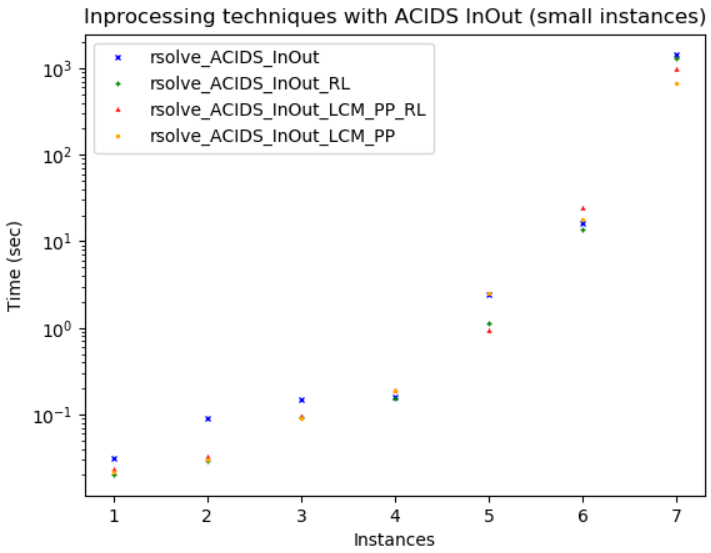

Figure 4.14: Time on instances of small size with `ACIDS_InOut` and inprocessing

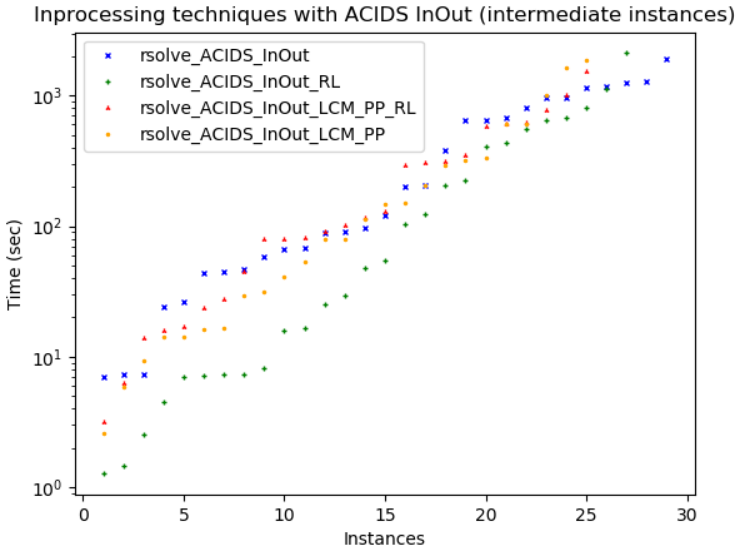

Figure 4.15: Time on instances of intermediate size with `ACIDS_InOut` and inprocessing (LOG scale)

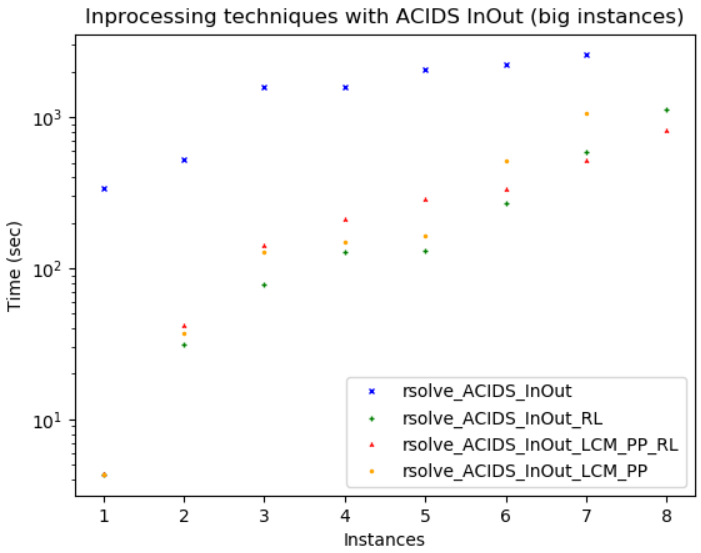

Figure 4.16: Time on instances of big size with `ACIDS_InOut` and inprocessing (LOG scale)

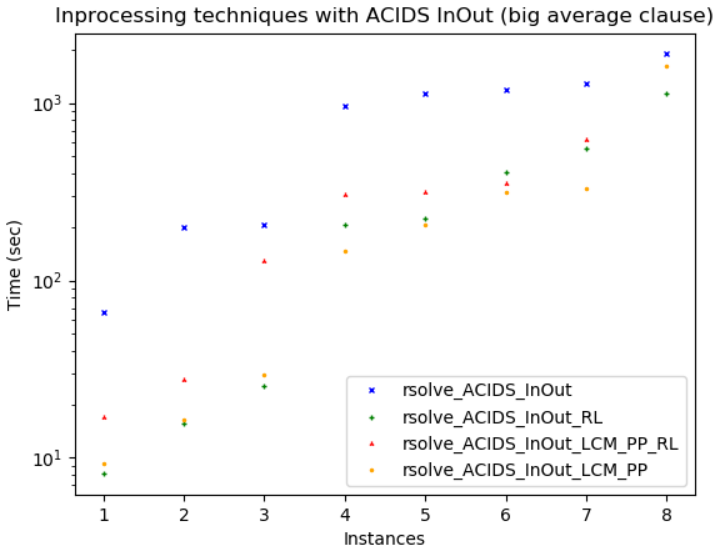

Figure 4.17: Time on instances with high average clause size with `ACIDS_InOut` and inprocessing (LOG scale)

Let's see if the restart heuristic used is changing the observations done. For the Luby heuristic (still with ACIDS), the same observations can be done : we have a gain with the inprocessing techniques. Table 4.4.3 shows the global results. Figures 4.18 and 4.19 illustrate the intermediate and big instances.

	`Luby`	`Luby_LCM_PP`
Solved instances	40	40
Total time (s)	13457.663000	10808.532000

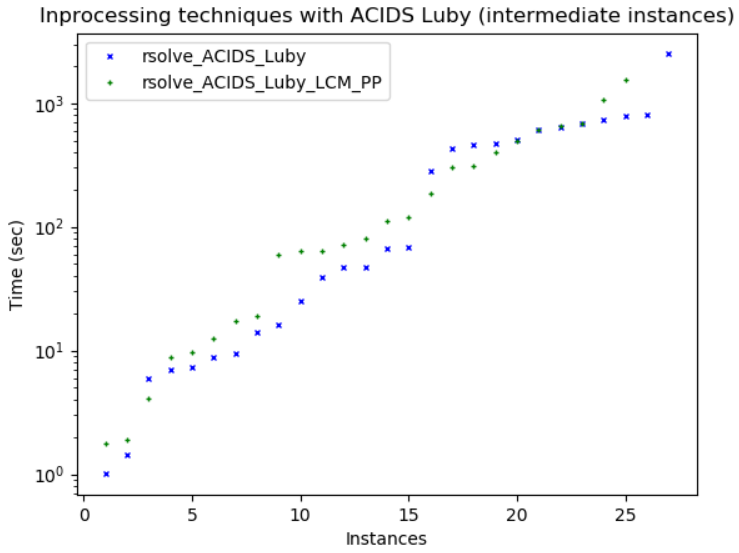

Figure 4.18: Time on instances of intermediate size with `ACIDS_Luby` and inprocessing (LOG scale)

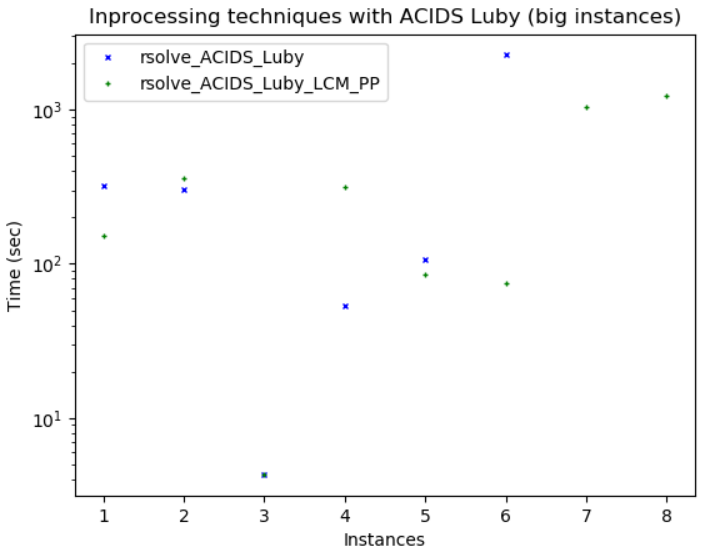

Figure 4.19: Time on instances of big size with `ACIDS_Luby` and inprocessing (LOG scale)

But for the GLUCOSE heuristic, it seems that, as the heuristic is already performing, the LCM has not a good impact (see Table 4.4.3). Figure 4.20 illustrates that, for the intermediate instances, the solver with LCM is worse than the basic GLUCOSE solver. For the big instances (Figure 4.21), the two solvers are nearly equivalent on most of the instances.

	`GLUCOSE`	`GLUCOSE_LCM_PP`
Solved instances	41	40
Total time (s)	8292.745000	10917.741000

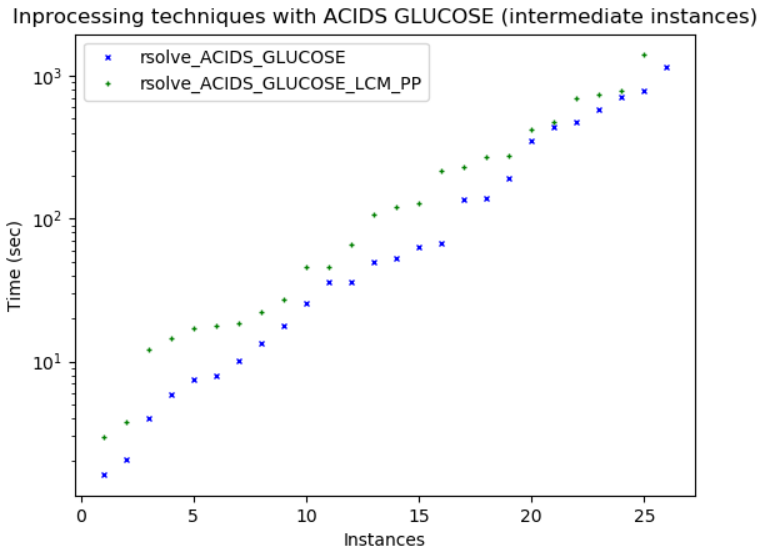

Figure 4.20: Time on instances of intermediate size with `ACIDS_GLUCOSE` and inprocessing (LOG scale)

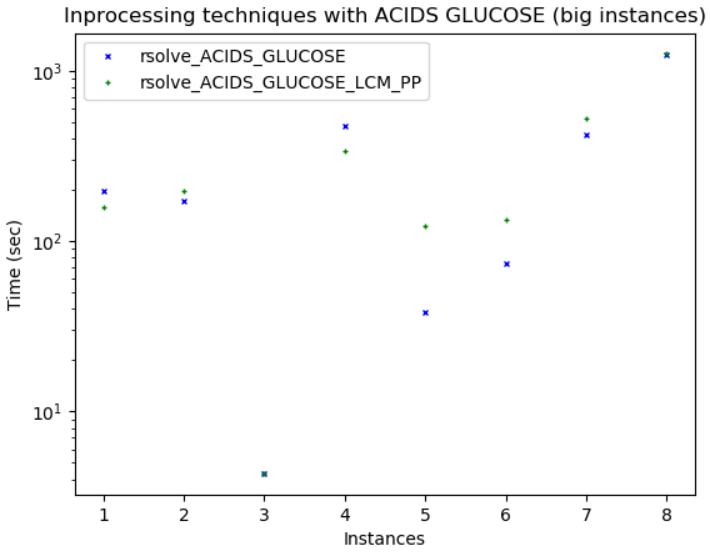

Figure 4.21: Time on instances of big size with `ACIDS_GLUCOSE` and inprocessing (LOG scale)

Conclusions on inprocessing We can see that the impact of the inprocessing is not significant for small instances but, as the size of the instances increases, the inprocessing has a higher effect on the elapsed time. This was an expected observation as the size of the instance reflects the number of clauses in memory. On intermediate instances, simply removing the clauses that have a forced literal has an impact on the elapsed time. It illustrates the fact that the performance of the main algorithms is really linked to the number of clauses. With big instance size, the LCM algorithm has more clauses to deal with. As clauses are explicit constraints on literals, it allows the algorithm to see more links and to minimize more efficiently the learned clauses. The RL still has a good impact. The same observations can be done for all the restarts. But for the GLUCOSE restart, which performs better without inprocessing, we need to have even bigger instances to see a positive impact of the LCM. Intermediate instances of the tests are not big enough for the GLUCOSE restart with LCM.

Chapter 5

Conclusion

The SAT problem is a well known problem of propositional logic. This problem is at the heart of the complexity theory as it is the first NP-complete problem that has been proven by Cook in 1971. SAT is so important that it is the heart of one of the Millennium questions. Finding a polynomial algorithm to solve SAT would prove that $P = NP$ and would revolutionize the Mathematical world. The SAT problem has many real-world applications such as planning and scheduling problems as well as electronic design and model checking.

In that context, SAT solvers have been developed. Furthermore, improvements methods have been designed to enhance the search algorithm used by the solvers. During this thesis, we have approached this topic by explaining the idea behind the different methods : how they work and how they are implemented.

This thesis is focused on the most popular and promising kind of solver : the CDCL solver. The first sections provide the necessary background to understand properly the SAT problem, the CDCL solver and the SAT improvement techniques. Concretely, the SAT problem is first defined and the notion of clause, which serves as basis to model an instance, is explained. Then, we have a look at the DPLL search algorithm which is the base of the CDCL solver. Afterwards, the specificity of the CDCL, the learned clauses, are presented by showing how they are computed and why they are useful. After the CDCL specificity, the techniques improving the DPLL algorithm are shown. First, the Variable Selection process is explained and different variants are introduced. In the same time, a word is given about the Value Selection Heuristics. Then, the restart strategies are presented. With the restart strategies, several questions arise : the learned clauses quality, the management of all these clauses and the loss of information after the restart. Therefore, those topics are detailed. Finally, we present the techniques not directly linked with the search. Those techniques try to simplify the input formula. They are known as preprocessing techniques if they are used before the search and inprocessing

techniques if there are interleaving with the search.

In the second part of this work, a solver has been extended. The architecture of this solver is described and the main components of the solver are explained. Improvement techniques have been added to this solver. The techniques implemented are mentioned and explained.

In the final part, a comparative study of the techniques has been done. This study compares the different techniques based on their behaviour.

We compare different Variable selection methods. We show that the Variable selection heuristics are required to have a good solver, compared to the naive selection heuristic. Also, some variable selection heuristics have their own domain : for example the Random heuristic for Cryptography instances. We see the difference between ACIDS and VSIDS based on the instance size and the average clause size. Those two heuristics show equivalent performance on small and big instances but VSIDS performs better on intermediate instances.

Afterwards we compare several restart strategies. It shows that the InOut restart is outperformed by the two others because the InOut strategy performs too much restarts. Glucose and Luby show similar behaviour for small and intermediate instances but are more distinguishable on bigger instances.

Finally, we compare the inprocessing techniques. We illustrate the fact that these techniques are a bit time consuming and have a clear gain as the size of the instance increases (this is the case with Luby and InOut but not for Glucose which is yet performing without the inprocessing and require even bigger instances).

Future work

At the end of this thesis, it appears that many other comparisons can be done with other techniques. It could be decision heuristics, restart strategies, inprocessing techniques or even other techniques. Rsolve is a very modular solver : we can still add other tools and use them. Moreover, it would be interesting to analyse and compare other aspects of the solvers. In this thesis, we always work with the same management of the learned clauses. It would be interesting to analyse the impact of other database managements. Another interesting aspect is the symmetry breaking. Symmetries are redundancies on the clauses and could be avoided with some symmetry breaking algorithms. The impact of the symmetry breaking can also be analysed.

List of Figures

Bibliography

[1] S. Cook. The complexity of theorem proving procedures. *Proceedings of the Third Annual Symposium on Theory of Computing, pp151-158*, 1971.

[2] P. Chen and K. Keutzer. Towards true crosstalk noise analysis. *International Conference on Computer-Aided Design,pp. 132–138.*, 1999.

[3] A. Middeldorp P. Schneider-Kamp R.Thiemann and H. Zankl C. Fuhs, J. Giesl. Sat solving for termination analysis with polynomial interpretations. *International Conference on Theory and Applica-tions of Satisfiability Testing,pp. 340–354.*, 1999.

[4] B. Selman and H. Kautz. Planning as satisfiability. *European Conference on Artificial Intelligence,pp. 359–363.*, 1992.

[5] Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat, 5*, 2005.

[6] Yakov Novikov Eugene Goldberg. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics, 155(12):1549–1561*, 2007.

[7] Laurent Simon Gilles Audemard. On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 2017.

[8] A. Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMV Technical Report 10/1, Johannes Kepler University, Linz, Austria*, 2010.

[9] G.S. Tseytin. On the complexity of derivation in propositional calculus. *Leningrad Seminar on Mathematical Logic*, 1966.

[10] H. Zhang and J. Hsiang. Solving open quasigroup problems by propositional reasoning. *Proceedings of the International Computer Symp*, 1994.

[11] Matti Juhani; Suda-Martin Heule, Marijn J. H.; Järvisalo, editor. *Proceedings of SAT Competition 2018; Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, Helsinki Institute for Information Technology, 2018.

[12] Ashish Sabharwal Bart Selman Carla P. Gomes, HenryKautz. *Handbook of Knowledge Representation - Chapter 2 : Satisfiability solvers*. 2008.

[13] G.S. Tseytin. Yet another local search solver and lingeling and friends entering the sat competition 2014. *Proceedings of SAT Competition 2014, pages 39-40*, 2014.

[14] Armin Biere. Choosing probability distributions for stochastic local search and the role of make versus break. *Lecture Notes in Computer Science, Volume 7317, Theory and Applications of Satisfiability Testing - SAT 2012, pages 16-29*, 2012.

[15] Oliver Gableske. Dimetheus. *sat competition 2016, pages 37-38*, 2016.

[16] Oliver Gableske. On the interpolation between product-based message passing heuristics for sat. *International Conference on Theory and Applications of Satisfiability Testing*, 2013.

[17] Holger H. Hoos Kevin Leyton-Brown Lin Xu, Frank Hutter. Satzilla: Portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, 2008.

[18] Hantao Zhang and Mark Stickel. Implementing the davis–putnam method. *Journal of Automated Reasoning, 24(1-2):277–296*, 2000.

[19] J. Marques-Silva and K. GRASP Sakallah. A new search algorithm for satisfiability. *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, 1997.

[20] J. Marques-Silva and K. GRASP Sakallah. A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 1999.

[21] Thomas Glase Richard Tichy. Clause learning in sat. *Seminar Automatic Problem Solving*, 2005.

[22] Yogesh Marhajan Zhaohui Fu and Sharad Malik. Zchaff sat solver. 2004.

[23] Ying Zhao Lintao Zhang Matthew W Moskewicz, Conor F Madigan and Sharad Malik. Chaff: Engineering an efficient sat solver. *Proceedings of the 38th annual Design Automation Conference, pages 530–535*, 2001.

[24] Armin Biere. Picosat essentials. *Satisfiability, Boolean Modeling and Computation 4*, 2008.

[25] Armin Biere and Andreas Fröhlich. Evaluating cdcl variable scoring schemes. *In International Conference on Theory and Applications of Satisfiability Testing, pages 405–422. Springer*, 2015.

[26] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers. *PhD thesis, Citeseer*, 2004.

[27] B. Selman C. P. Gomes and H. A. Kautz. Boosting combinatorial search through randomization. *Proceedings of AAAI*, page 431–437, 1998.

[28] A. Sinclair M. Luby and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters, Volume 47, Issue 4, pp 173-180*, 1993.

[29] Laurent Simon Gilles Audemard. Predicting learnt clauses quality in modern sat solvers. *IJCAI, volume 9, pages 399–404*, 2009.

[30] A. Biere. Precosat. *SAT 2009 Competitive Event Booklet*, 2009.

[31] M. Soos. Cryptominisat 2.5.0. *SAT Race 2010 solver description*, 2010.

[32] Giráldez-Cru J. Ansótegui, C. and J. Levy. The community structure of sat formulas. *Theory and Applications of Satisfiability Testing*, 2012.

[33] Giráldez-Cru J. Levy J. Ansótegui, C. and L. Simon. Using community structure to detect relevant learnt clauses. *Theory and Applications of Satisfiability Testing*, 2015.

[34] R. Ganian and S. Szeider. Community structure inspired algorithms for sat and #sat. *Theory and Applications of Satisfiability Testing*, 2015.

[35] Biere-A. Eén, N. Effective preprocessing in sat through variable and clause elimination. *Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg*, 2005.

[36] Järvisalo-M. Biere A. Heule, M.J.H. Clause elimination procedures for cnf formulas. *Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 357–371. Springer, Heidelberg*, 2010.

[37] Järvisalo-M. Biere A. Heule, M.J.H. Covered clause elimination. *LPAR-17 Short Papers*, 2010.

[38] Pradhan-D.K. Subbarayan, S. Niver: Non-increasing variable elimination resolution for preprocessing sat instances. *H. Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 276–291. Springer, Heidelberg*, 2005.

[39] Biere A. Järvisalo M., Heule M.J.H. Inprocessing rules. *Gramlich B., Miller D., Sattler U. (eds) Automated Reasoning. IJCAR 2012. Lecture Notes in Computer Science, vol 7364. Springer, Berlin, Heidelberg*, 2012.

[40] Armin Biere. Preprocessing and inprocessing techniques in sat. *Haifa Verification Conference*, 2011.

[41] Marijn J.H. Heule. Preprocessing and inprocessing. *SC² Summer School*, 2017.

[42] Fan Xiao Felip Manyà Zhipeng Lü Mao Luo, Chu-Min Li. An effective learnt clause minimization approach for cdcl sat solvers. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 2017.

[43] Chanseok Oh. *Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL*. PhD thesis, New York University, 2016.

[44] Tom Melham Rajdeep Mukherjee, Daniel Kroening. Hardware verification using software analyzers. *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015.

[45] Jussi Rintanen. Planning with specialized sat solvers. *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11), pages1563–1566. AAAI Press*, 2011.

[46] Marijn J. H. Balyo, Tomàš; Heule. Proceedings of sat competition 2016; solver and benchmark descriptions. *Department of Computer Science Series of Publications B , vol. B-2016-1 , University of Helsinki , Helsinki .*, 2016.

[47] L. Cruz-Filipe M. Codish and P. Schneider-Kamp. Sorting networks: The end game. *LATA*, 2015.

[48] T. Ehlers and M. Müller. New bounds on optimal sorting networks,. *CiE*, 2015.

Appendices

Appendix A

Source code

The complete source code of the project can be consulted on the GitHub repository of the project: https://github.com/Spirou07/rsolve.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**