Another Way to Browse the Search Space For Some Transformations from CSP to SAT

Richard Ostrowski and Lionel Paris and Adrien Varet
Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{richard.ostrowski, lionel.paris}@univ-amu.fr
adrien.varet@lis-lab.fr
Aix-Marseille Universite

Abstract

In this paper, we study a new way of browsing the search space when solving CSP (Constraint Satisfaction Problem) encoded in SAT (Satisfiability problem), using direct and support encodings. We exploit a different splitting rule during the exploration of the search tree from that used in classical solvers. Usually, for solving SAT problem, the Conflict Driven Clause Learning (CDCL) algorithms are based on the Davis–Putnam–Logemann–Loveland (DPLL) procedure. It consists in choosing the next variable to instantiate thanks to a heuristic, then to assign it to true (or false) and check if, after some simplifications via unit propagation, there exists a model or not under this hypothesis. If a conflict arises, a clause is learned and a backtrack (usually non-chronological, i.e. a backjump) is operated in order to check the other branch of the search tree. This splitting rule (one literal and its opposite) can be generalized to any formula between some variables, with some restrictions: firstly, its negation is easily calculated (with respect to the CNF formalism), secondly the formula must simplify the instance, and thirdly, the complexity must not be higher than the SAT problem complexity. Using, for the beginning, the direct encoding of a CSP in SAT instance, we propose an algorithm for the resulting SAT formula with a better theoretical complexity. This allows us to reach a better complexity for solving CSP too. Furthermore, this paper explains how we have adapted the existing CDCL mechanisms (clause learning, implication graph) to our algorithm.

INTRODUCTION

The satisfaction problem (SAT) and the constraints satisfaction problem (CSP) are two fairly close problems. On the one hand, SAT consists in determining whether a Boolean formula in conjunctive normal form is satisfiable. On the other hand, CSP problems are represented by a set of variables taking their values in finite domains and linked by constraints. Each constraint can be described either in terms of permitted or forbidden pairs. Solving a CSP consists in finding an instantiation of the variables satisfying all the constraints of the problem.

These two formalisms are widely used in artificial intelligence. They allow modelling and solving both academic problems (pigeon hole problem, n-queens problem) and real-world and industrial problems (formal verification, planning, etc.)

The efficiency of these solvers is largely due to the exploitation of structural properties naturally present in the coding of problems such as logical gates (or more generally Boolean functions), symmetries, equivalences of variables, etc.

Recently, a mathematical problem (Heule, Kullmann, and Marek 2016) that has remained open until now, was solved by providing a counter-example using a SAT solver. The size of the proof, which was verified, was close to 200TB.

Although it is possible to express any problem of constraints in different formalisms, it often happens that a formalism is more suitable than another. For example, it is easier to consider a graph colouring problem as a CSP than to formalize it as a set of propositional formulas. However, many works have focused on transformations from CSP to SAT. De Kleer (Kleer 1989) introduced direct encoding while Kasif (Kasif 1990) proposed the encoding of binary CSP including supports (AC) to maintain the property of arc consistency by unit propagation in SAT. Finally, Bessière *et al.* (Bessière, Hebrard, and Walsh 2003) generalized the encoding of supports (k -AC) to non-binary CSPs.

Other transformations exist (Ansótegui and Manyà 2005; Barahona, Hölldobler, and Nguyen 2014; Gavanelli 2007; Argelich et al. 2012) and we believe that our new splitting rule can be easily adapted to them.

Furthermore, CDCL solvers (Silva and Sakallah 1996) are very effective to solve the SAT problem. They use the same basis as the DPLL algorithms, but at each new choice, they maintain an implication graph with all the consequences of all choices. This graph is used when a conflict arises, to learn a new clause in order to avoid getting the same conflict and perform backjumping (non-chronological backtracking). This kind of solvers also perform *"restarting"* whose frequency can be defined in different ways. This paper introduces the way that we adapted theses methods to our algorithm.

The main contribution of this paper is to provide a CSP solving algorithm having a theoretical complexity of $\mathcal{O}(\frac{d}{2}^n \times filtering\_cost)$ instead of $\mathcal{O}(d^n \times cost\_filtering)$ for a classic algorithm. The paper is organized as follows: as

a first step, we recall some definitions concerning SAT and CSP formalisms and about the two encoding methods used. Then, we present a different way of browsing the search tree to provide a complete algorithm with the same complexity as the stochastic algorithm proposed in (Schöning 1999) and how we adapt the CDCL methods to this algorithm.

PRELIMINARIES DEFINITIONS

Boolean Satisfiability Problem (SAT)

Let $\mathscr{B}$ be a Boolean (*i.e.* propositional) language of formulas built in the standard way, using usual connectives ($\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) and a set of propositional variables. A *CNF formula* Σ is a set (interpreted as a conjunction) of *clauses*, where a clause is a set (interpreted as a disjunction) of *literals*. A literal is a positive or negative propositional variable.

Let us recall that any Boolean formula can be translated to CNF using linear Tseitin encoding (Tseitin 1968). The size of CNF Σ is defined by $\sum_{c\in\Sigma}|c|$ where $|c|$ is the number of literals in c.

A *truth assignment* of a Boolean formula is an assignment of truth values $\{true, false\}$ to its variables. A literal l is satisfied (resp. falsified) under I if l is positive and $I[l] = true$ or l is negative and $I[l] = false$ (resp. l is negative and $I[l] = false$ or l is positive and $I[l] = true$). A *model* of a formula is a truth assignment that satisfies the formula. Accordingly, the decision problem associated to SAT consists in determining if the formula admits a model.

A functional dependency is of the form $y = f(x_1,...,x_n)$ where f is a standard connective among $\{\vee, \wedge, \Leftrightarrow\}$ and where y and x_i are propositional variables.

Constraints Satisfaction Problem (CSP)

A CSP is a statement $P = (X, D, C, R)$ where $X = \{X_1, X_2, \ldots, X_n\}$ is a set of n variables, $D = \{D_{X_1}, D_{X_2}, \ldots, D_{X_n}\}$ is a set of finite domains where D_{X_i} is the domain of possible values for X_i, $C = \{C_1, C_2, \ldots, C_m\}$ is a set of m extensional constraints, where the constraint C_i is defined on a subset of variables $\{X_{i_1}, X_{i_2}, \ldots, X_{i_{a_i}}\} \subset X$.

The arity of the constraint C_i is a_i and $R = \{R_1, R_2, \ldots, R_k\}$ is a set of k relations, where R_i is the relation corresponding to the constraint C_i. R_i contains the forbidden combinations of values for the variables involved in the constraint C_i. A binary CSP is a CSP whose constraints are all of arity two (binary constraints). A CSP is non-binary if it involves at least a constraint whose arity is greater than 2 (a n-ary constraint).

An *instantiation* I is a mapping which assigns each variable X_i a value of its domain D_{X_i}. A constraint C_i is satisfied by the instantiation I if the projection of I on the variables involved in C_i is different of all tuples of R_i. An instantiation I of a CSP P is consistent (or called a solution of P) if it satisfies all the constraints of P. A CSP P is consistent if it admits at least one solution. Otherwise P is not consistent.

For the rest of the paper, we denote by n the number of variables of the CSP, by m its number of constraints, by a its maximal constraint arity and by d the size of its largest domain.

Encoding CSP into SAT

There are different classical transformations from CSP to SAT. Some are more efficient than others because they allow to find good properties with respect to unit propagation or to model in a more efficient way certain constraints of the initial problem. In this article we focus on the direct and support encodings and show in perspective adaptations to other transformations.

Direct Encoding This is the most used and the oldest one. It was introduced by De Kleer(Kleer 1989).In this encoding, we associate a propositional variable to each value in the domain of each CSP variable. For example, a CSP variable X with domain $D_X = \{v_1, v_2, \ldots, v_k\}$ is associated with k Boolean variables: $x_{v_1}, x_{v_2}, \ldots, x_{v_k}$. The assignment x_{v_i} = *true* means that value v_i is assigned to variable X. These propositional variables appears in three kind of clauses :

- ***at-least-one* clauses:** There is one such clause per CSP variable. They encode the domains of the CSP, expressing the fact that each CSP variable must be assigned to one value of its domain. Lets consider a CSP variable X with domain $D_X = \{v_1, v_2, v_3\}$. The clause $c_X = x_{v_1} \vee x_{v_2} \vee x_{v_3}$ is added to encode D_X. These clauses will be noted *a.l.o.* clauses in the sequel.
- ***at-most-one* clauses:** There is one such clause for each pair of values in each domain. These are binary clauses encoding the fact that a CSP variable can only be assigned to one value of its domain. Lets consider the previous example. For the domain $D_X = \{v_1, v_2, v_3\}$, we have to add 3 binary clauses: $\neg x_{v_1} \vee \neg x_{v_2}$, $\neg x_{v_2} \vee \neg x_{v_3}$ et $\neg x_{v_1} \vee \neg x_{v_3}$. These clauses will be noted *a.m.o* clauses in the sequel. It is usually possible to get rid of this set of clauses. Indeed, if a model assigning several values to the same CSP variable is found, it is possible to choose one randomly and to ignore others. Another way to see that we can possibly get rid of these *a.m.o* clauses is to see that they are blocked clauses (Kullmann 2001). A clause c is blocked if there is a literal l in this clause such that all the resolutions produced on this literal are tautological. The suppression of the blocked clauses preserves the satisfiability of the formula: they do not participate in the proof of the inconsistency of the formula.
- **conflict clauses:** There is a conflict clause for every *forbidden tuple* of each constraint of the CSP. They express the constraints by coding all the combinations forbidden by them. For example, consider a constraint between three variables CSP X, Y and Z and $[u \in D_X, v \in D_Y, w \in D_Z]$ a constrained tuple C_{XYZ} (that is, $[u, v, w] \in R_{XYZ}$). The conflict clause $c_{XYZ} = \neg x_u \vee \neg y_v \vee \neg z_w$ is added to prohibit the simultaneous assignment of X to u, Y to v and Z to w.

Let $\mathscr{P}$ be a CSP and Σ_{Dir} the CNF formula resulting from the direct encoding of $\mathscr{P}$. Walsh proved that performing Arc consistency filtering on $\mathscr{P}$ is stronger than performing unit propagation on Σ_{Dir} (Walsh 2001). He also demonstrated, as it had been by Génisson and Jégou (Génisson and Jégou 1996) that with equivalent branching heuristics, FC applied to $\mathscr{P}$ and DP applied to Σ_{Dir} were equivalent.

The complexity of the direct coding of a CSP is in $\mathscr{O}(md^a)$, remembering that m represents the number of con-

straints of the CSP, a the largest arity of the CSP constraints and d the size of the largest domain of the CSP. More precisely, in the direct encoding of a CSP in CNF, we find n *a.l.o.* clauses of size d. Each CSP variable add $\frac{d(d-1)}{2}$ *a.l.o.* clauses of size 2. Finally each constraint can contain at most d^a forbidden tuples of length a. This gives a total complexity of $(nd + 2n\frac{d(d-1)}{2} + mad^a) \equiv \mathcal{O}(md^a)$.

Support Encoding (AC-Encoding) The support encoding was introduced by Kasif (Kasif 1990) and specially designed for the binary CSP. In this encoding, we find the same propositional variables as for the direct encoding. There is also the same *a.l.o.* and *a.m.o.* clauses. The only difference lies in the conflict clauses that are replaced by clauses called support clauses. Firstly, remember that a support of a value is defined as follows :

Definition 1. *Let $P = (X, D, C, R)$ a CSP, $x, y \in X$ two variables, $v_x \in D_x$ and $v_y \in D_y$ two values. We say that v_y is a support of v_x in D_y if and only if the affectation $(x, y) = (v_x, v_y)$ is allowed by the constaint between x and y.*

- **support clauses :** A support clause is added for each pair (*value, list of supports*) of each binary constraint. It encodes the fact that as long as a *value* remains in the domain of one of the variables involved in the constraint, then at least one of these supports must remain in the domain of the second variable involved. For example, consider X and Y two CSP variables and the pair $(v \in D_X, \{s_1, s_2, \ldots, s_k\} \in D_Y)$ where v is a value of the domain X and $\{s_1, s_2, \ldots, s_k\}$ are the supports of $X = v$ in the domain of Y for the constraint C_{XY}. The clause $\neg x_v \vee y_{s_1} \vee y_{s_2} \vee \ldots \vee y_{s_k}$ is added to express the fact that as long as v is not removed (filtered) from the domain of X, at least one of its supports in D_Y should not be deleted. This clause is equivalent to $x_v \rightarrow (y_{s_1} \vee y_{s_2} \vee \ldots \vee y_{s_k})$. So we see that if all the supports are removed (falsified), $\neg x_v$ is implied, that is, v is removed from the domain of X when filtering by arc consistency.

We see the value of this encoding for binary CSPs compared to direct coding. This encoding allows recovering the arc consistency in the CNF. Let $\mathscr{P}$ be a CSP and Σ_{Sup} the CNF formula resulting from the support encoding of $\mathscr{P}$. Kasif showed that the unit propagation applied to Σ_{Sup} is equivalent to arc consistency filtering on $\mathscr{P}$. This allowed Gent (Gent 2002) to show that with equivalent branching heuristics, DP applied to Σ_{Sup} is equivalent to MAC applied to $\mathscr{P}$.

The complexity of the support encoding of a CSP is in $\mathcal{O}(nd^2 + md^2)$. Precisely, we find the same *a.l.o.* and *a.m.o.* clauses in (nd^2), and for each clause, there is $2d$ possible pairs (*value, list of supports*), each list being able to contain at most d values $(2md^2)$. The total complexity is hence $(nd^2 + 2md^2) \equiv \mathcal{O}((n+m)d^2)$.

Some classes of CSP instances are tractable and translating them using some common standard encodings results in SAT instances which do not fall into known tractable classes (Petke and Jeavons 2011).

A NEW ALGORITHM

A New Splitting Rule

The idea of this article is to propose a new splitting rule, more general than what is currently used for CDCL like solvers. These solvers are based on the following proposition:

Proposition 1. *Let Σ be a CNF and v a propositional variable. Σ is satisfiable if and only if $(\Sigma \wedge v)$ is satisfiable or $(\Sigma \wedge \neg v)$ is satisfiable.*

By repeating this property to the different variables of Σ, we can build a backtracking algorithm to test if an interpretation satisfies Σ. We propose to generalize this property as follows:

Proposition 2. *Let Σ be a CNF and f a Boolean formula built on the propositional variables of Σ. Σ is satisfiable if and only if $(\Sigma \wedge f)$ is satisfiable or $(\Sigma \wedge \neg f)$ is satisfiable.*

In the same way, by considering successively different Boolean functions (as points of choice), we can also build a backtracking algorithm allowing this time to determine if the set of Boolean functions satisfies the original formula. The proposition 1 can be seen as a special case of the proposition 2 where the formula f is reduced to a single literal.

However, the chosen functions must respect some constraints: (1) in order to remain in the CNF formalism, the negation of the formula must be computed easily (and quickly), (2) the formulas chosen as a point of choice must simplify Σ, (3) the complexity, in terms of the number of nodes, must be lower or equal than 2^n where n is the number of variables of the formula Σ. The last point requires us to focus on the functional dependencies between the propositional variables of the problem.

Indeed, by considering a functional dependency of the form $y = f(x_1, ..., x_n)$ as a point of choice,we can eliminate a propositional variable (y), at each step,by substituting it with its definition. As a result,we are sure not to have a greater complexity than 2^n.

The generalization of the splitting rule to other formulas for the SAT problem are perspectives for future work. Indeed, many points have to be studied, such as the storage of the points of choice, learning, backjumping, restarts and heuristics in order to obtain experimental results that can compete with the best current CDCL solvers.

We initially restrict ourselves to the CSP solving. Our approach rely on two well known transformations from CSP into SAT, namely the direct encoding and the support encoding.

Another Way to Browse The Search Tree

In this part, relying first of all on the direct encoding of a CSP, we propose a different way to browse the search tree. To this aim, in relation with Proposition 2, we use as formula f a functional dependencie of difference between two propositional variables of the direct encoding. However, the choice of these two variables will have to be made in a *a.l.o* clause. Let's take the example of a CSP variable X with a domain $D_X = \{v_1, v_2, v_3, v_4\}$. The clause $c_X = x_{v_1} \vee x_{v_2} \vee x_{v_3} \vee x_{v_4}$ represents the *a.l.o* clause to encode the domain D_X. By choosing, for example, as point of

choice, a formula of the type $f = (x_{v_1} \neq x_{v_2})$ it is assumed that there is a model where either x_{v_1} is true or x_{v_2} is true (only one of them must be true). This implies for the other propositional variables of the *a.l.o* clause (i.e. x_{v_3} and x_{v_4}) to have the value false. We will see later how to detect this truth value in Algorithm 1. We repeat this separation rule to other pairs of propositional variables, always choosing them in *a.l.o* clauses. If there is no solution under the hypothesis $f = (x_{v_1} \neq x_{v_2})$, we must test the other branch, namely $\neg f$. This negation becomes the formula $\neg f = (x_{v_1} = x_{v_2})$. As these two variables are part of a *a.l.o* clause, we deduce that x_{v_1} = false and x_{v_2} = false. The next point of choice, compared to the example, will consist in assuming that $f = (x_{v_3} \neq x_{v_4})$.

This algorithm is quite close to an algorithm consisting of dichotomously separating the domain of a CSP variable in two, while performing arc consistency before separating the sub-domains again (Larrosa 1997). Thus, if a CSP variable X has a domain $D_X = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, the domain is separated in two sub-domains: $\{v_1, v_2, v_3\}$ and $\{v_4, v_5, v_6\}$. The process is repeated as long as the size of the sub domains is at least equal to 2.

Another work on the partitioning of domains is used in (Sule 2013)(Sule 2014). Our algorithm differs from the works above in that we do not partition below a size of domain at least equal to 2. Indeed, suppose that there are no constraints between 3 variables X Y and Z from a CSP with $D_X = \{x_1, x_2\}, D_Y = \{y_1, y_2\}$ and $D_Z = \{z_1, z_2\}$. Partitioning domains will successively try the $\{x_1, y_1, z_1\}$, $\{x_1, y_1, z_2\}$, $\ldots, \{x_2, y_2, z_2\}$ assignments. If the remaining sub-problem is not consistent, 2^3 assignments will have been made. In our case, by selecting as points of choice $(x_1 \neq x_2)$, $(y_1 \neq y_2)$ and $(z_1 \neq z_2)$ a single assignment is sufficient.

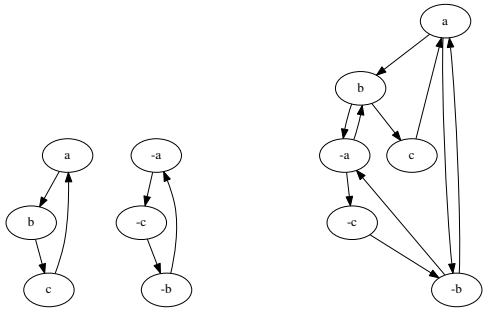
About the other types of solvers, this kind of branching has already been used especialy graph colouring solvers. The idea was the same: being able to test two different colors for a node at each branching.

From a CSP point of view, the way we parse the search tree by considering constraints of differences on domain values, consists in incrementally building a bi-valued binary CSP. Polynomial algorithms exist to determine whether such a CSP admits a solution. From a SAT point of view, for direct encoding, this parsing consists in incrementally constructing a 2-SAT formula which is also a polynomial class.

The question that must be asked here is: do we have to check, at each points of choice, whether the CSP or the 2-SAT formula is inconsistent?

In our case, for the direct encoding, determining whether a 2-SAT formula is inconsistent can be done either in calculating the strongly connected components by the Tarjan algorithm (Tarjan 1972) or by showing that there exists a variable v of the 2-SAT formula which is inconsistent by assigning it to true or false.

Lets consider for example the following 2-SAT formula $\Sigma = \{(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee a)\}$. The graph of strongly connected components is given in Figure 1a. There is no path linking a literal and its opposite in both directions. The formula is therefore satisfiable. Now, if we assume the hypothesis $(a \neq b)$, the graph becomes the one in the figure 1b. In this graph, there is a path from a to $\neg a$ and a path from $\neg a$ to a. Under this hypothesis the formula is therefore unsatisfiable.



(a) (b) hypothesis $(a \neq b)$

Figure 1: Strongly Connected Components Graphs

Remark 1. *In general, it is the last hypothesis (or choice point) that builds bidirectional bridges between the strongly connected components.*

We can deduce the following proposition:

Proposition 3. *Let Σ be a consistent 2-SAT formula, and constraint of difference $(l_1 \neq l_2)$ representing our point of choice. The formula $(\Sigma \wedge (l_1 \neq l_2))$ is unsatisfiable if and only if $(\Sigma \wedge l_1 \wedge \neg l_2)$ is unsatisfiable and $(\Sigma \wedge \neg l_1 \wedge l_2)$ too.*

Unit propagation in this case is sufficient to determine the satisfiability of the formula.

In the case where the unit propagation leads to a conflict on one of both branches, the other branch is implied. If there is no contradiction on both branches, we take advantage of the work carried out by the unit propagation to infer literals implied on both sides. This work (of implied literals) is the filtering method of our algorithm.

If we consider the following formula: $\Sigma = \{(x_{v1} \vee x_{v2} \vee x_{v3}) \wedge (\neg x_{v1} \vee \neg x_{v2}) \wedge (\neg x_{v1} \vee \neg x_{v3}) \wedge (\neg x_{v2} \vee \neg x_{v3})\}$ representing a domain clause of a CSP variable. Under the hypothesis $(x_{v1} \neq x_{v2})$ we deduce an implied literal $\neg x_{v3}$.

Filtering the domains of other variables is done by the detection of implied literals. For example, for the formula $\Sigma = \{(x_{v1} \vee x_{v2} \vee x_{v3}) \wedge (\neg x_{v1} \vee \neg x_{v2}) \wedge (\neg x_{v1} \vee \neg x_{v3}) \wedge (\neg x_{v2} \vee \neg x_{v3}) \wedge (y_{v1} \vee y_{v2} \vee y_{v3}) \wedge (\neg y_{v1} \vee \neg y_{v2}) \wedge (\neg y_{v1} \vee \neg y_{v3}) \wedge (\neg y_{v2} \vee \neg y_{v3}) \wedge (\neg x_{v1} \vee \neg y_{v1}) \vee (\neg x_{v2} \vee \neg y_{v1}))\}$. And, under the hypothesis $(x_{v1} \neq x_{v2})$, it is possible to imply the literal $\neg x_{v3}$ and $\neg y_{v1}$.

Algorithm 1 summarizes our approach. It takes as input a CNF formula resulting from the direct encoding of a CSP problem. At each step, we choose a domain clause (line 1). If none is available, it means that we are on a satisfiable branch and the algorithm stops. In the opposite case, we choose two unassigned variables to constitute our point of choice (line 6). If only one variable is available, we propagate it and continue the search (line 8). In case of contradiction, a backtracking is performed. If two unassigned variables are available (for example x and y), the point of choice becomes

$(x \neq y)$. Then we test (line 11) the propagation of $\{x, \neg y\}$ on one side and $\{\neg x, y\}$ on the other side. If both branches lead to a conflict by this propagation, we directly infer $\{\neg x, \neg y\}$. In the opposite case, the unit propagation of the literals in common (calculated previously) is carried out and the search is continued (line 17).

Remark 2. *From a CSP point of view, our algorithm consists in iteratively restricting the sizes of the variable domains to 2 while performing a filtering. In case of a conflict, we can directly test two other values of the domain of the last chosen variable.*

Remark 3. *Regarding the support encoding, search tree exploration carried out by our algorithm works in an identical way and allows inferring more literals. Indeed, the unit propagation is equivalent to the arc consistency. Finally, we do not need to check the satisfiability of the formula produced by the points of choice, because the unit propagation, performed in both branches, detects this inconsistency.*

Proposition 4. *The theoretical complexity of our algorithm from a CSP point of view is* $\mathcal{O}((\frac{d}{2})^n \times filering\_cost)$ *where d represents the size of the largest domain and n the number of variables.*

Algorithm 1 FCEquiv

Require: Σ, L
Ensure: Solve ϕ
1: $cla \leftarrow choose\_clause\_domain(\Sigma)$
2: **if** $(cla = null)$ **then**
3: **return** $Satisfiable$
4: **end if**
5: **while** (true) **do**
6: $(x, y) \leftarrow choose\_couple\_litteral(cla)$
7: **if** $(x \neq null$ AND $y = null)$ **then**
8: $solve(\Sigma, level + 1)$
9: **return**
10: **end if**
11: $L \leftarrow test\_and\_filter(\Sigma, x, y)$
12: **if** $(\bot \in L)$ **then**
13: **if** $(BCP(\Sigma, \{\neg x, \neg y\}) = \bot$ **then**
14: **return** $\bot$
15: **end if**
16: **else**
17: **return** $solve(\Sigma, level + 1)$
18: **end if**
19: **end while**

ADAPT CDCL COMPONENTS

A New Implication Graph

As a reminder, CDCL algorithms maintain an implication graph at each new choice. It is used when a conflict happens to learn a new clause, in order to avoid to get the same conflict later and to achieve non-chronological backtracking.

With the intention to improve our algorithm's efficiency, we wanted to use these methods. But, because we choose constraints of difference and CDCL algorithms choose simple literals as points of choice, our implication graph must have a different behaviour. At each new choice $(x \neq y)$ (and after having tested the two sub-branches $(x \wedge \neg y)$ and $(y \wedge \neg x)$, we'll update it as follows:

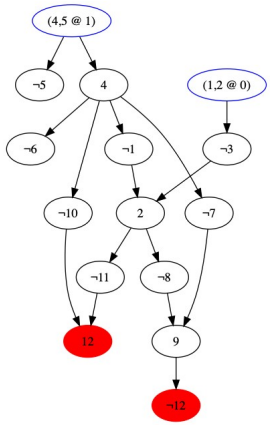


Figure 2: Example of implication graph

1. **One sub-branch is satisfiable**: create a node $(x \neq y@p)$ (where p is the current decision level), a new node l for each literal l in the satisfiable sub-branch and add a directed edge between two nodes v_1 and v_2 if the literal v_2 is a consequence of v_1.
2. **Two sub-branches are satisfiable**: create a node $(x \neq y@p)$, a new node l for each literal l in the intersection of the two sub-branches and, for each choice $(x'\neq y'@p')$ responsible of the deduction of l in one of the two sub-branches, add an arc between $(x'\neq y'@p')$ and l.
3. **Two sub-branches are unsatisfiable**: Develop the graph for each sub-branch (in the same way that case 1). Then, for each branch, start from the literal(s) responsible(s) of the conflict and explore the graph backward in order to find all responsible choice(s). The figure 2 shows an example of implication graph with 12 and ¬12 as conflict literals.

Two New Clause Learning Methods

Now, we are able when a conflict happens to get the set of responsible choices (we just have to take the union of conflict choices in each sub-branches), let $C = \{(x_1 \neq y_1@p_1), (x_2 \neq y_2@p_2)..., (x_c \neq y_c@p_c)\}$ be this set. We've tested two ways to represent a conflict clause.

The first one consists in taking all choices $(x_i \neq y_i@p_i) \in C$ and add to a new clause the literals in the associated *a.l.o* clause, except x_i and y_i. For example, we can consider a problem who has the following *a.l.o* clauses:

$$\{x_1 \vee x_2 \vee x_3, x_4 \vee x_5 \vee x_6, x_7 \vee x_8 \vee x_9\}$$

Then let's assume that a conflict happens due to the following choices $\{(x_1 \neq x_2@0), (x_4 \neq x_5@1), (x_7 \neq x_8@2)\}$, we learn the clause $c = x_3 \vee x_4 \vee x_5$. Now, if after a restart, we choose $(x_1 \neq x_2@l_1)$ and $(x_4 \neq x_5@l_2) \forall l_1, l_2$, $\neg x_3$ is a consequence of the first choice (because x_3 is in the same *a.l.o* clause as x_1 and x_2), and $\neg x_6$ will be a consequence of the second choice for the same reason. Because of this x_3 and x_6

will be deleted in c. Then, c will become unitary and x_9 will be deduced, the consequence of the affectation of x_9 will be $(\neg x_7 \wedge \neg x_8)$, ie $(x_7 = x_8)$ which is equivalent to $\neg(x_7 \neq x_8)$. So, the previous conflict is avoided.

The advantage of this method is that we only learn one clause and we don't learn new variables. But on the other hand, it doesn't work for problems with $d > 3$ (d is the size of the biggest domain). For example, if we keep the previous example and we add x_{10} to the last *a.l.o* clause, then the learned clause will be $c = x_3 \vee x_6 \vee x_9 \vee x_{10}$. After the choices $(x_1 \neq x_2 @ l_1)$ and $(x_3 \neq x_4 @ l_2) \forall l_1, l_2$, c will not became unitary, so $\neq x_7$, $\neq x_8$ will not be deduced and we can always choose $(x_7 \neq x_8 @ l_3) \forall l_3$.

In order to deal with this issue, we implemented a second method. This one is quite similar to the previous, in the idea, we just used new variables as propagators to directly infers the negation of the both literals of a choice when necessary. Let's take a problem which has the followings *a.l.o* clauses as example:

$$\{x_1 \vee x_2 \vee x_3 \vee x_4, x_5 \vee x_6 \vee x_7 \vee x_8, x_9 \vee x_{10} \vee x_{11} \vee x_{12}\}$$

Then let's assume that a conflict happens due to the following choices $\{(x_1 \neq x_2 @ 0), (x_5 \neq x_6 @ 1), (x_9 \neq x_{10} @ 2)\}$, then we'll learn the following clauses:

$$\{c_1 = y_1 \vee y_2 \vee y_3, c_2 = \neg y_1 \vee \neg x_1, c_3 = \neg y_1 \vee \neg x_2,$$
$$c_4 = \neg y_2 \vee \neg x_5, c_5 = \neg y_2 \vee \neg x_6, c_6 = \neg y_3 \vee \neg x_9,$$
$$c_7 = \neg y_3 \vee \neg x_{10}\}$$

Now, let's assume that after a restart we made the choice $(x_1 \neq x_2 @ l_1)$. During the filtering procedure, we'll test the sub-branches $(x_1 \wedge \neg x_2)$ and $(\neg x_1 \wedge x_2)$. In the first one, $\neg y_1$ will be a consequence of x_1, and of x_2 in the second one. So, $\neg y_1$ is in the intersection of the two sub-branches and will be deduced by the filtering procedure. Then, if we choose $(x_5 \neq x_6 @ l_2)$, $\neg y_2$ will be deduced for the same reason. In this case, c_1 will become unitary, y_3 will be deduced, thus $\neg x_9$ and $\neg x_{10}$ will be direct consequences of y_3. Now, we cannot choose $(x_9 \neq x_{10} @ l_3)$ again, so the conflict is avoided.

This method has the advantage to works even with problems having domains sizes greater than 3. But if a conflict that implies c choices happens, then we'll learn c new variables and $2|c| + 1$ new clauses.

Backjumping and Restarts

We saw in previous part that when a conflict happens, we can easily retrieve all choices $C = \{(x_1 \neq y_1 @ p_1), (x_2 \neq y_2 @ p_2) ..., (x_c \neq y_c @ p_c)\}$ who caused it. To perform a backjumping, we just have to go back just after the second higher level of C. For example, let's suppose we made the choices: $\{(x_1 \neq y_1 @ 1), (x_2 \neq y_2 @ 2), (x_3 \neq y_3 @ 3), (x_4 \neq y_4 @ 4), (x_5 \neq y_5 @ 5)\}$ and the set of conflict choices is: $\{(x_1 \neq y_1 @ 1), (x_2 \neq y_2 @ 2), (x_5 \neq y_5 @ 5)\}$. In this case, 2 is the second higher level, so we will go back to the level 3 and propagate $(\neg x_3 \wedge \neg y_3)$.

Concerning the restart strategy, we decide to use Luby's strategy.

EXPERIMENTAL RESULTS

In this part, we present preliminary results with and without clause learning methods.

Without Clause Learning Methods

We made a first series of tests on pigeon-hole problems who still remain a challenge for CDCL solvers because they can't exploit symmetries and clause learning is useless. These results are detailed in Table 1. The first column refers to the size of the problem, then we compare our algorithm with a basic Forward Checking solver (without clause learning, backjumping, etc...) and minisat (Eén and Sörensson 2003). These tests are done with direct and support encodings. For each instance, we wrote the execution time (in milliseconds) and the number of nodes. We can easily see that after $n = 12$, minisat shows its limits (we had a 5min timeout), but our algorithm solves it in one minute. To conclude, we can see that results for support encoding are less interesting.

Secondly, we tested our algorithm with the same configuration on random graph coloration problem. We fixed the number of vertices between 20 and 30, its roughness between 0.6 and 0.9 and the number of colors is defined according to its roughness. For the three algorithms, we chose to use the DSATUR heuristic (Brélaz 1979) (choosing the vertices with the less remaining colors). These results are detailed in Table 2. These results are similar to the previous, indeed our solver had better results than minisat, in terms of time and number of nodes. In the two tables, the results are given as follows : time(number of choices).

With Clause Learning Methods

This part presents our first preliminary results using clause learning methods. We firstly made a series of tests on random with 5 colors and a large amount of vertices (500). For each density (between 0.1 and 0.9), we tested 5 instances and took the average time/number of nodes. We compared our results with minisat and we only used the direct encoding version of the problem. We can see that our algorithm has some difficulties with a low density, but with a higher density, it is clearly better than minisat and, in terms of number of nodes, our algorithm is also better than minisat. Figure 3 sums up the behaviour of both solver:

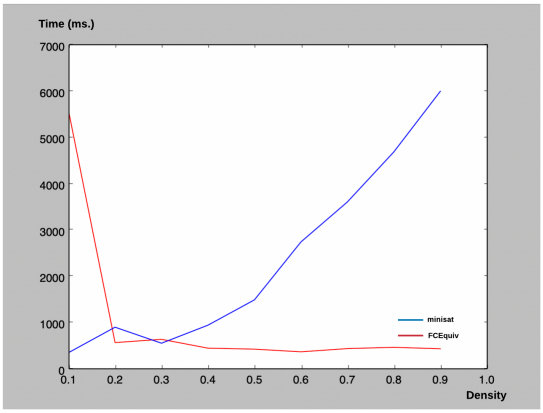


Figure 3: Behaviour on graph coloration problem (500 vertices, 5 colours)

	Direct encoding			Support encoding		
n	FCEquiv	FC	minisat	FCEquiv	FC	minisat
7	**3**(252)	3(1438)	6(1407)	6(252)	6(1438)	7(1139)
8	**6**(1660)	7(10078)	24(7664)	25(1660)	39(10078)	42(7411)
9	**27**(7004)	63(80638)	180(44545)	125(7004)	390(80638)	307(40534)
10	**140**(60161)	453(725758)	6120(1000172)	1301(60161)	4643(725758)	5959(525536)
11	**814**(313410)	5111(7257598)	124356(11828510)	8707(313410)	59995(7257598)	119739(6277093)
12	**9256**(3322496)	64081(79833598)	TIMEOUT	113000(3322496)	TIMEOUT	TIMEOUT
13	**64720**(20615382)	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT

Table 1: Pigeon-hole results without clause learning

			Direct encoding			Support encoding		
n	color	density	FCEquiv	FC	minisat	FCEquiv	FC	minisat
20	6	0.6	0(2448)	0(1438)	0(1921)	0.1(2448)	0.1(1438)	0.1(2163)
20	7	0.7	0.18(30637)	0.12(50398)	0.18(24994)	1.4(30637)	1(50398)	0.5(32776)
20	9	0.8	7(1385001)	14(3628798)	42(201141)	93(1385001)	165(3628798)	36(1200601)
20	10	0.9	6.36(1386975)	12(7257598)	124(5380698)	105(1386975)	211(7257598)	288(6256227)
30	8	0.6	25(3165468)	14(4677118)	257(8979189)	313(3165468)	159(4677118)	141(3027918)
30	9	0.7	9(1252857)	14(4354558)	44(1754928)	132(1252857)	208(4354558)	78(1490465)

Table 2: Graph coloration problem without clause learning

PERSPECTIVES AND CONCLUSION

In this article, we proposed a generalization of the commonly used splitting rule for CDCL type SAT solvers. Instead of choosing as branching strategy, a literal and its opposite, we considered a difference and an equality between two propositional variables. The idea, at this stage, is not to apply it for solving the SAT problem in a general way. We have restricted our splitting rule to CNF problems obtained from the basic transformations of a CSP to SAT (direct and support encodings). Compared to a Forward Checking algorithm, the theoretical complexity is reduced from $\mathcal{O}$ ($d^n \times filtering\_cost$) to $\mathcal{O}$ ($(\frac{d}{2})^n \times filtering\_cost$) with d the size of the largest domain and n the number of variables. The first experimental results, although preliminary, are very encouraging. The problem of pigeons holes can be solved more effectively without adding, for example a treatment on the detection symmetries. We also proposed a way to adapt the method widely used in classic CDCL solvers to our algorithm (a new implication graph, conflict clause, etc).

In perspectives, it can be interesting to consider other splitting rules respecting the three conditions like for example the *and* or the *or* functional dependencie between some variables. It can also be interesting to test our algorithm with other kinds of encoding, we think for example to the order encoding with which CDCL solvers obtained good results (Tamura, Banbara, and Soh ; Ohrimenko, Stuckey, and Codish 2009).

To conclude, this paper only presents preliminary results and many points deserve to be studied.

References

Ansótegui, C., and Manyà, F. 2005. Mapping problems with finite-domain variables to problems with boolean variables. SAT'04.

Argelich, J.; Cabiscol, A.; Lynce, I.; and Manyà, F. 2012. Efficient encodings from csp into sat, and from maxcsp into maxsat. *Journal of Multiple-Valued Logic and Soft Computing*.

Barahona, P.; Hölldobler, S.; and Nguyen, V.-H. 2014. Representative encodings to translate finite CSPs into SAT. CPAIOR'14.

Bessière, C.; Hebrard, E.; and Walsh, T. 2003. Local consistency in SAT. SAT'03.

Brélaz, D. 1979. New methods to color the vertices of a graph. *Commun. ACM* 22(4).

Eén, N., and Sörensson, N. 2003. An extensible sat-solver. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, 502–518.

Gavanelli, M. 2007. The log-support encoding of csp into sat. CP'07.

Génisson, R., and Jégou, P. 1996. Davis and Putnam were already forward checking. ECAI'96.

Gent, I. P. Arc consistency in SAT. *ECAI'02*.

Heule, M. J. H.; Kullmann, O.; and Marek, V. W. 2016. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. SAT'16.

Kasif, S. 1990. On the Parallel Complexity of Discrete Relaxation in Constraint Satisafaction Networks. *Journal of Artificial Intelligence*.

Kleer, J. D. 1989. A comparison of ATMS and CSP techniques. In *IJCAI'89*, 290–296.

Kullmann, O. 2001. On the use of autarkies for satisfiability decision. *Electronic Notes in Discrete Mathematics*.

Larrosa, J. 1997. Merging constraint satisfaction subproblems to avoid redundant search. IJCAI'97.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints*.

Petke, J., and Jeavons, P. 2011. The order encoding: From tractable csp to tractable sat. SAT'11.

Schöning, U. 1999. A probabilistic algorithm for k-sat and constraint satisfaction problems. FOCS '99.

Silva, J. P. M., and Sakallah, K. A. 1996. GRASP: A new search algorithm for satisfiability. ICCAD'96, 220–227.

Sule, V. 2013. Generalization of boole-shannon expansion, consistency of boolean equations and elimination by orthonormal expansion. *CoRR* abs/1306.2484.

Sule, V. 2014. An algorithm for boolean satisfiability based on generalized orthonormal expansion. *CoRR* abs/1406.4712.

Tamura, N.; Banbara, M.; and Soh, T. Compiling pseudo-boolean constraints to SAT with order encoding. ICTAI'13.

Tarjan, R. E. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput.* 1:146–160.

Tseitin, G. 1968. On the complexity of derivations in the propositional calculus. In *Structures in Constructives Mathematics and Mathematical Logic, Part II*, 115–125.