Artificial Intelligence Research and Development
I. Sanz et al. (Eds.)

doi:10.3233/FAIA230704

SAT-IT: The Interactive SAT Tracer

Marc CANÉ [a], Jordi COLL [b,1], Marc ROJO [a] and Mateu VILLARET [a,2]
[a] *Universitat de Girona, Girona, Spain*
[b] *Institut d'Investigació en Intel·ligència Artificial, CSIC, Bellaterra, Spain*
ORCiD ID: Jordi Coll https://orcid.org/0000-0002-9385-5723, Mateu Villaret https://orcid.org/0000-0002-8066-3458

Abstract. In this work we present the Interactive SAT Tracer (SAT-IT), a visual and interactive tool to monitor and illustrate the basic algorithms for solving the Boolean Satisfiability Problem (SAT). We consider three algorithms with progressively increasing sophistication: simple backtracking, its extension with unit-propagation, so called Davis-Putnam-Logemann-Loveland (DPLL), and its further extension with clause-learning and conflict driven, so called Conflict-Driven Clause Learning (CDCL). The motivation of this tool is to provide an environment where the user can see the full trace of a SAT solving process in a compact but detailed way and understand the reasons why each variable assignment, backtrack or backjump occurs. Moreover, we want the user to be able to control the evolution of the solving process at the desired pace, and let they choose what branchings (or decisions) must be done. Being able to control the solving process and having detailed information of this process, results in many pedagogical and research applications such as understanding algorithms or designing encodings.

Keywords. SAT, CDCL, DPLL, tool, tracer

1. Introduction

The Boolean Satisfiability problem (SAT) is the paradigmatic NP-complete problem [1,2]. This is the problem of deciding whether a Boolean propositional formula can be satisfied. This problem is not only relevant for being the first problem that was shown to be NP-complete, but his popularity and research interest have been continuously increasing during the last decades for its applicability as a problem-solving paradigm.

The key factors in the success of SAT are the facility to translate a plethora of hard constraint satisfaction and optimisation problems to SAT formulas, and the development of extremely efficient algorithms for solving such formulas. Despite the enormous theoretical computational complexity of SAT, nowadays we have efficient methods capable of solving huge formulas coming from problems of industrial interest such as circuit verification [3], planning [4], scheduling or timetabling [5,6], to name a few.

Except for very specific domains, there is one clearly predominating algorithm to solve SAT: the Conflict-Driven Clause Learning algorithm (CDCL). The essence of

[1]Corresponding Author: Jordi Coll, jcoll@iiia.csic.es, partially supported by Grants PID2019-111544GB-C21 and TED2021-129319B-I00 funded by MCIN/AEI /10.13039/501100011033.

[2]Corresponding Author: Mateu Villaret, mateu.villaret@udg.edu, partially supported by: Grant PID2021-122274OB-I00 funded by MCIN/AEI/10.13039/501100011033 and by ERDF A way of making Europe.

CDCL is a combination of search and inference. It consists of a (non-chronological) backtracking scheme that explores a search tree to find a solution if any exists, enhanced with the Unit Propagation (UP) rule and which uses the Resolution rule to learn new clauses from dead ends of the search tree. Therefore, understanding the basics of SAT-solving requires to get familiar with the previously mentioned inference rules and their integration into a SAT solving algorithm. In this regard, in [7] a very compact and precise rule-based framework to describe SAT solving algorithms was presented.

Inspired by the previously mentioned framework, in this work we present the Interactive SAT Tracer (SAT-IT), a visual and interactive tool to monitor and illustrate the basic algorithms for SAT solving. In order to facilitate the learning of SAT solving techniques to users that start without a background knowledge, we consider three algorithms with progressively increasing sophistication: simple backtracking, its extension with UP, so called Davis-Putnam-Logemann-Loveland (DPLL), and its further extension with conflict-driven clause learning, i.e. CDCL. In contrast to the teaching tool LearnSAT [8] our tool provides an interactive environment with some graphical support. The motivation of our tool is to provide an environment where the user can see the full trace of a SAT solving process in a compact but detailed way and understand the reasons why each variable assignment, backtrack or backjump occurs. Further information is included in the tool, such as the sequence of resolution rules involved in each conflict analysis, what clauses have been learnt, or what are the inspected literals involved in the 2-watched literals scheme for implementing UP. Moreover, the user is able to control the solving process evolution at the desired pace and choose what variables should be used for branchings (or decisions). The system displays a full log of the SAT solving process that allows to see the overall progress of the execution until the current point, and users can go back to any previous point of the solving process to reinspect or to try "what if" scenarios. In particular, one could see which variables are unit-propagated after some particular assignments. This could serve the user to get some practical insights of some properties of the encodings such as correctness or generalized arc consistency enforcement by UP.

In the rest of the paper, we review some preliminary concepts in Section 2, we present the SAT-IT rule-based notation in Section 3, we describe the considered algorithms for SAT solving in Section 4, we describe the SAT-IT tool in detail in Section 5 and we conclude in Section 6.

2. Preliminaries

A propositional *variable* x can take Boolean truth values, i.e. 0 (meaning *false*) or 1 (meaning *true*). A *literal* l is a variable x, or a negated variable $\neg x$. A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_k$. A *propositional formula in Conjunctive Normal Form* (CNF) is a conjunction of m clauses $c_1 \wedge \cdots \wedge c_m$. A CNF is usually seen as a set of clauses. An *assignment* is a mapping from propositional variables to $\{0, 1\}$. An assignment *satisfies* (respectively *falsifies*) literal x if $x = 1$ (respectively $x = 0$), and satisfies (respectively falsifies) literal $\neg x$ if $x = 0$ (respectively $x = 1$). An assignment is often expressed as a set of satisfied literals, e.g. $x = 1, y = 0, z = 1$ can be expressed as $\{x, \neg y, z\}$. A literal l is *defined* in an assignment M if $l \in M$ or if $\neg l \in M$, and otherwise is *undefined*. An assignment is *complete* if all the variables are assigned a value, otherwise it is *partial*.

Given an assignment, a clause is satisfied if at least one of its literals is satisfied, it is falsified if all its literals are falsified, and it is *unit* if all its literals are false but one which is unassigned. A CNF is satisfied if all its clauses are satisfied. An assignment satisfies a formula F if it satisfies all the clauses of F. We call an assignment satisfying a formula F a *model* of F. Given two formulas F and F', we say that F' is a *logical consequence* of F, denoted $F \models F'$, if every model of F is a model of F'. This notation can be extended to clauses and assignments: a clause C and a negation of a clause $\neg C$ are also formulas, and an assignment M can be seen as a conjunction of literals, i.e. also as a formula. Then, $M \models C$ means that assignment M satisfies clause C and $M \models \neg C$ means that M falsifies C.

Definition 1 *The* Boolean satisfiability *(SAT) problem consists in determining whether there exists a model for a given CNF formula.*

The unit propagation and the resolution rules play a central role in SAT solvers.

Definition 2 *Given an assignment M and a clause $C \vee l$ such that $M \models \neg C$ and l is undefined in M, (i.e., $C \vee l$ is a unit clause given M), the* unit propagation *(UP) rule extends M to $M \cup \{l\}$.*

Definition 3 *Given two clauses $C \vee l$, $C' \vee \neg l$, the* resolution *rule derives the clause $C \vee C'$. It holds that $(C \vee l) \wedge (C' \vee \neg l) \models C \vee C'$.*

Modern SAT solvers implement the Conflict-Driven Clause Learning (CDCL) algorithm. It is an algorithm that combines search and inference. Moreover, CDCL solvers include a series of techniques which are crucial to achieve the best performance, such as *restarts*, the *variable-elimination inprocessing* and the *learnt clause removal* [9]. However, for now SAT-IT only focuses on the basic search and inference components.

In particular, SAT-IT is designed to explain the search component of SAT-solving algorithms by starting with a naive backtracking scheme which is incrementally enriched, obtaining the Davis-Putnam-Logemann-Loveland (DPLL) algorithm at a first step, and the CDCL algorithm at a second step. The most basic backtracking scheme tries to find a model by successively assigning arbitrary truth values to the variables (it makes *decisions*). If a clause is falsified (it finds a *conflict*), it undoes a decision (it does a *backtrack*) and continues the search, thus forming a search tree. If this basic backtracking scheme is improved by applying UP whenever possible, we obtain the DPLL algorithm. The *decision level* of an assigned literal l is the number of decisions made before being assigned (including l), and the decision level of an assignment (or of a point of the search tree) is the maximum decision level of its literals. If a conflict is found at decision level 0, the formula is unsatisfiable. Finally, CDCL improves DPLL with clause learning. When it reaches a conflict, *conflict analysis* is performed to derive a new clause explaining the conflict, and it does a non-chronological backtrack (or *backjump*). This new clause is inferred with a sequence of applications of the resolution rule, and is added to the formula (it is *learnt*). We refer the reader not familiar with SAT to [10,11,12,7,13] for more detailed explanations about DPLL and CDCL.

3. SAT-IT Rules

In [7] there was presented a rule-based framework to describe the DPLL algorithm and some extensions such as CDCL, or DPLL(T) for solving Satisfiability Modulo Theories

DECIDE:		
$M \parallel F$	$\Longrightarrow M\ l^d \parallel F$	**if** $\begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$
BACKTRACK:		
$M\ l^d\ N \parallel F \cup \{C_i\}$	$\Longrightarrow M\ \neg l^k \parallel F \cup \{C_i\}$	**if** $\begin{cases} M\ l^d\ N \models \neg C_i \\ N \text{ contains no decision literals} \end{cases}$
FAIL:		
$M \parallel F \cup \{C_i\}$	$\Longrightarrow FailState$	**if** $\begin{cases} M \models \neg C_i \\ M \text{ contains no decision literals} \end{cases}$
UNITPROPAGATE :		
$M \parallel F \cup \{C_i\}$	$\Longrightarrow M\ l^i \parallel F \cup \{C_i\}$	**if** $\begin{cases} C_i \text{ has the form } C' \vee l \\ M \models \neg C' \\ l \text{ is undefined in } M \end{cases}$
BACKJUMP:		
$M\ l^d\ N \parallel F \cup \{C_i\}$	$\Longrightarrow M\ l'^j \parallel F \cup \{C_i\}$	**if** $\begin{cases} M\ l^d\ N \models \neg C_i, \text{ and} \\ \text{there is some clause } C_j \text{ such that:} \\ \quad C_j \text{ has the form } C' \vee l', \\ \quad F \cup \{C_i\} \models C_j \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^d\ N \end{cases}$
LEARN:		
$M \parallel F$	$\Longrightarrow M \parallel F \cup \{C_j\}$	**if** $\begin{cases} \text{each literal of } C_j \text{ occurs in } F \text{ or in } M \\ F \models C_j \end{cases}$

Figure 1. Rule-based representation of SAT-IT.

(SMT). This framework allows to formally reason about the algorithms using a simple representation. SAT-IT works with (a slight modification of) a subset of the rules presented in [7], that is listed in Figure 1, and described after providing some notation.

A *state* is defined as a pair of the form $M \| F$, where M is a (partial) assignment and F is a formula in CNF, i.e. a finite set of clauses. Every rule, of the form $M \| F \Longrightarrow M' \| F'$, can only be applied if a set of conditions over $M \| F$ are satisfied, and its application changes the state from $M \| F$ to $M' \| F'$. Then, the process of solving a SAT formula F is represented as a sequence of states $S_0 \Longrightarrow S_1 \Longrightarrow S_2 \Longrightarrow \cdots \Longrightarrow S_n$ obtained by a sequence of rule applications, where $S_0 = \{\} \| F$. If F is satisfiable, then S_n is of the form $M \| F'$, where M is a model of F. Otherwise (i.e., F is unsatisfiable), S_n is the special state $FailState$. A partial assignment is not represented as a set of literals but as a sequence M of literals tagged with superscripts. We write together assignments and literals to denote concatenation, e.g. $M\ l$ is the sequence M followed by literal l.

The rules used by SAT-IT are the following:

DECIDE: Extends the partial assignment with an undefined literal l (it *decides* l).

BACKTRACK: This rule can be applied when an existing clause C_i is falsified by the partial assignment, and the partial assignment contains some decided literal. The last decided literal l is negated, and all literals after l are removed from the assignment.

FAIL: This rule can be applied when an existing clause C_i is falsified by the partial assignment, and the partial assignment does not contain any decided literal. The result is the *FailState* (the formula is unsatisfiable).

UNITPROPAGATE: This rule applies UP on the unit clause $C_i = C' \vee l$.

BACKJUMP: This rule can be applied in the same case as the BACKTRACK rule. However, instead of negating the last decision, the rule removes the partial assignment until (and including) a decision l^d, which is not necessarily the last decision (i.e., it performs non-chronological backtracking). The backjump decision level d is determined by a clause $C_j = C' \vee l'$ which is falsified by $M\ l\ N$. More precisely, d is one plus the largest decision level of literals in C'. Note that C_j does not necessarily exist in F, C_i, but it must be a logical consequence of F, C_i. Note also that clause C' is unit with assignment M, and therefore we extend M with l'^j due to UP on clause C_j.

LEARN: This rule adds a new clause to the formula. The added clause must be logical consequence of the formula.

The only rules that add a literal to the partial assignment, possibly after removing other literals, are DECIDE, UNITPROPAGATE, BACKTRACK and BACKJUMP. Like in [7], a literal l that has been added to an assignment M with the DECIDE rule will have the symbol d as a superscript, meaning that l has been assigned with the application of the DECIDE rule. However, in [7] no superscript is added in the remaining cases. SAT-IT rules introduce a small difference: literals assigned by BACKJUMP and UNITPROPAGATE contain an integer superscript i, meaning that l has been assigned by UP on the unit clause C_i; literals assigned by BACKTRACK contain the symbol k as a superscript. This representation is closer to the real implementation of CDCL SAT solvers [12], where each literal in the sequence of assignments (a.k.a. *trail*), has a *reason* to be assigned, i.e. it has been either decided or unit-propagated due to a particular case (no backtrack k reason exists in CDCL). Moreover, this extra information allows a better understanding of the development of the solving process, as well as a better identification of the clauses involved in conflict analysis. Example 1 explains how to read the trail of a given state.

Example 1 *Consider the trail* 1^d $\text{-}5^5$ 2^d $\text{-}3^k$ 4^0 *corresponding to the second line of Figure 3, left (where* x_i *is denoted just i and* $\neg x_i$ *as -i, as we explain later). Literals* 1 *and* 2 *have been decided (*DECIDE *rule), literal* -3 *is the negation of a previous decision (*BACKTRACK *rule), and literals* -5 *and* 4 *have been unit-propagated due to clauses* C_5 *and* C_0 *respectively, either by* UNITPROPAGATE *rule or by* BACKJUMP *rule. Literals* 1 *and* -5 *have decision level 1, and literals* 2, -3 *and* 4 *have decision level 2.*

4. SAT-IT Algorithms

SAT-IT supports three SAT-solving algorithms with increasing sophistication: BACKTRACKING, DPLL and CDCL. These three algorithms can be described as applications of different sets of rules from Figure 1. In fact, using these rules one could design other solving algorithms. We consider those three since they follow a natural evolution form a naive backtracking one to a full (though basic) CDCL solver. In the provided pseudocode for these algorithms, we use the auxiliary function *canApply*(R,M,F), which checks the shape of M and the conditions of rule R on the state $M \| F$ to determine the applicabil-

Algorithm 1: BACKTRACKING algorithm

Input: $F = \{C_1, \ldots, C_{|F|}\}$, a set of clauses.
Output: If F is SAT: (SAT, M), where M is a model of F. Otherwise: $UNSAT$.

$M \leftarrow \{\}$
while *True* **do**
  if *CanApply(*BACKTRACK*,M,F)* **then** $(M,F) \leftarrow$Backtrack(M,F) ;
  else if *CanApply(*FAIL*,M,F)* **then return** *UNSAT* ;
  else if *M is a complete assignment* **then return** *(SAT,M)* ;
  else $(M,F) \leftarrow$Decide(M,F) ;

Algorithm 2: DPLL algorithm

Input: $F = \{C_1, \ldots, C_{|F|}\}$, a set of clauses.
Output: If F is SAT: (SAT, M), where M is a model of F. Otherwise: $UNSAT$.

$M \leftarrow \{\}$
while *True* **do**
  if *CanApply(*BACKTRACK*,M,F)* **then** $(M,F) \leftarrow$Backtrack(M,F) ;
  else if *CanApply(*FAIL*,M,F)* **then return** *UNSAT* ;
  else if *M is a complete assignment* **then return** *(SAT,M)* ;
  else if *CanApply(*UNITPROPAGATE*,M,F)* **then** $(M,F) \leftarrow$UnitPropagate(M,F) ;
  else $(M,F) \leftarrow$Decide(M,F) ;

Algorithm 3: CDCL algorithm

Input: $F = \{C_1, \ldots, C_{|F|}\}$, a set of clauses.
Output: If F is SAT: (SAT, M), where M is a model of F. Otherwise: $UNSAT$.

$M \leftarrow \{\}$
while *True* **do**
  if *CanApply(*BACKJUMP*,M,F)* **then**
    $C_j \leftarrow$ConflictAnalysis(M,F)
    $(M,F) \leftarrow$Backjump(M,F,C_j)
    $(M,F) \leftarrow$Learn(M,F,C_j)
  else if *CanApply(*FAIL*,M,F)* **then return** *UNSAT* ;
  else if *M is a complete assignment* **then return** *(SAT,M)* ;
  else if *CanApply(*UNITPROPAGATE*,M,F)* **then** $(M,F) \leftarrow$UnitPropagate(M,F) ;
  else $(M,F) \leftarrow$Decide(M,F) ;

ity of the rule. The auxiliary functions *Decide*, *UnitPropagate*, *Backtrack*, *Backjump* and *Learn* perform one application of the corresponding rule and return the resulting state.

The simplest algorithm is BACKTRACKING, described in Algorithm 1, and can be defined in terms of the rules DECIDE, BACKTRACK and FAIL. The DPLL algorithm, described in Algorithm 2, is obtained by adding to BACKTRACKING the UNITPROPAGATE rule. Finally, the CDCL algorithm, described in Algorithm 3, replaces the BACKTRACK rule with the BACKJUMP rule, and adds the LEARN rule. Both rules are parametric to a clause C_j that we provide as an argument. In order to obtain C_j, we define the *ConflictAnalysis* function that requires as input a state $M \| F$ where we can apply the

Algorithm 4: Conflict Analysis

Input: F, M: A state where F contains some clause falsified by M.
Output: C: A new clause derived with resolution from a conflict until the 1UIP.

```
C ← a clause in F such that M ⊨ ¬C
while C contains more than one literal of the last decision level do
    l^i ← the rightmost literal of M s.t. ¬l ∈ C
    C ← Resolution(C, C_i)
return C
```

BACKJUMP rule, and therefore where there exists a clause $C_i \in F$ which is falsified by the current assignment M. The conflict analysis procedure is described in Algorithm 4. Starting from an unsatisfied clause C, and by means of the resolution rule, the conflict analysis transforms C until it contains only one literal of the last decision level. Such literal is usually referred to as the First Unique Implication Point (1UIP) [10]. After conflict analysis, the returned clause C_j is learnt and the 1UIP is unit-propagated due to clause C_j in the BACKJUMP rule.

5. SAT-IT GUI and Functionalities

SAT-IT is publicly available[3]. This tool allows the user to work with CNFs in DIMACS format using any of the three given algorithms. We find an example execution of CDCL in Figure 2, and of DPLL and BACKTRACKING in Figure 3. These figures consider the same example input formula:

$$C_0 : x_3 \vee x_4 \vee \neg x_1 \vee x_5 \quad C_2 : x_3 \vee \neg x_4 \vee \neg x_1 \quad C_4 : x_1 \vee \neg x_2 \quad C_6 : \neg x_3 \vee \neg x_4 \vee x_5$$
$$C_1 : \neg x_3 \vee x_4 \vee x_5 \quad C_3 : x_1 \vee x_2 \quad C_5 : \neg x_1 \vee \neg x_5$$

Moreover, when a decision is done, the provided examples always choose the first unassigned variable in the order $1, 2, \ldots, 5$, and always the positive literal.

SAT-IT illustrates the solving process and lets the user control the evolution of the process using the views and functionalities that we describe now. The main window of SAT-IT can be seen in Figure 2. We start by explaining the four main views of the main window of SAT-IT, that we refer as to *trail's history*, *clause list*, *event viewer* and *control buttons*. After this, we describe further functionalities.

Trail's history: The trail's history is shown in the enclosed area at the top-left part in Figure 2. This view shows the evolution of the trail (i.e. of M) as the algorithm progresses. In this window, a tagged literal x_i^t is denoted i^t, and a literal $\neg x_i^t$ is denoted $-i^t$. This representation corresponds to the standard DIMACS format. At the beginning of the execution, it starts with an empty trail. The new literals added to the trail with the rules DECIDE and UNITPROPAGATE are appended to the end of the current trail. Whenever a rule that removes literals from the trail is applied, namely BACKTRACK or BACKJUMP, the current trail is left in its form before the rule application, and the trail resulting of the rule application is added in a new line. See for instance that in Figure 2 there are 3 lines,

[3] https://imae.udg.edu/Recerca/LAI/

since we applied 2 times the BACKJUMP rule. In order to visually illustrate the backtrack point, the part of the trail that is kept after BACKTRACK or BACKJUMP is blurred out. The application of a BACKTRACK or BACKJUMP, which is written at the right of the current trail, is always preceded by the message *CONFLICT i*, meaning that C_i is the conflicting clause that triggered the rule. Moreover, the BACKJUMP rule is also followed by the message *LEARNED j*, meaning that clause C_j is learnt and is the clause that establish the backjump point (recall Algorithm 3). Finally, the last trail will either finish with *SAT* or *UNSAT*. The former case means that the formula is satisfiable and the last trail is a model, while the latter means that unsatisfiability is proven. Note that the content of this history once the solving process has finished shows in full detail what has been the process that the algorithm has followed to find a solution.

Clause list: The bottom-left enclosed part lists all the clauses of the formula. Every row, of the form $i : l_1\ l_2\ l_3\ \ldots\ l_n$ shows a different clause, where i is the name given to clause C_i, and the right part is the list of literals of the clause. Again, the minus sign means negated literal, hence following the standard DIMACS format. At the beginning, the list contains the original clauses. Every time a clause C_j is learned, the corresponding message *LEARNED j* is shown after the trail, and a new clause is added to the list. For instance, in Figure 2 clauses 7 and 8 are learned. Note that the ids of learnt clauses are painted in orange. The colouring of the literals during the solving process is modified. The colouring code of the literals, for a current trail (assignment M), is the following: red means falsified, green means satisfied, and black means unassigned.

Event viewer: This is the top-right part of the window and it shows, in chronological order, some events that the solver has processed together with extra information. The events that we consider are: (i) a decision, specifying the decided literal and its decision level; (ii) a maximal chain of consecutive unit propagations; (iii) a backtrack followed by the backtrack resulting literal; (iv) or a backjump followed by the decision level at which it "backjumps". This view provides therefore a summary of the whole solving process. Moreover, if the user double-clicks a backjump entry, a popup will show the chain of resolution rule applications followed in the conflict analysis in order to derive the clause to be learnt (see Figure 2 and recall Algorithm 4).

Control buttons: These buttons allow the user to control the solving process at the desired pace. The *Decision* button performs a literal decision. Depending on the selected configuration the literal is either picked automatically, or chosen by the user. In case of automatic picking, we can configure SAT-IT to pick the literals either based on input order, or following the VSIDS heuristic [11]. The latter case is only enabled if we use CDCL, and in that case the scores of the variables will be shown. The *Unit Prop.* button advances the execution step by step, doing only one propagation each time, or a decision if needed. The *Conflict* button advances the execution until a conflict is found. The *Resolve C.* button is enabled only when a conflict is found and is used to resolve the conflict. The *End* button will trigger the automatic completion of the solving process.

Other Functionalities: SAT-IT uses and illustrates the two-watched literals (2WL) scheme (see [11]) to apply UP. The literals that are added to the trail are processed in the order they are added since the last decision, in order to find unit clauses where to apply UP. That is, we will consider all unit clauses containing the literal at position i of the trail (starting from the last decision), before continuing with the unit clauses containing the

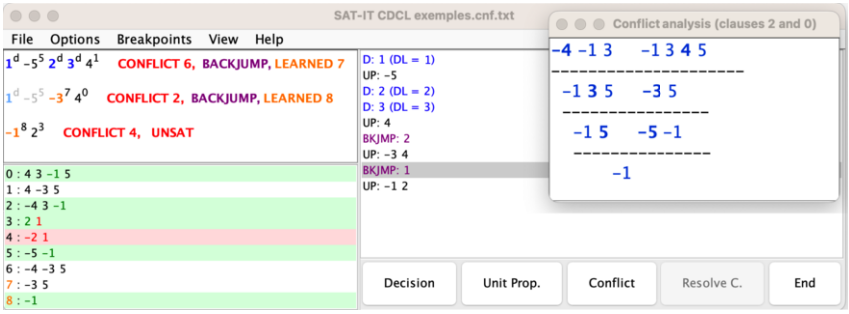

Figure 2. Main view of SAT-IT, with a completed execution of the CDCL algorithm, and the pop-up window showing the conflict analysis for the second backjump.

Figure 3. Left: trail history of a DPLL run. Right: three first and three last lines of the trail history of a BACKTRACKING run (the total number of lines of the example is 16).

literal at position $i+1$. 2WL benefits from the fact that inspecting two unassigned literals in a clause is enough to check the moment when the clause becomes unit. In SAT-IT, each clause in the clause list view is dynamically reordered to put the two watched literals in the two first positions of the list of literals. The user can thus appreciate how a chain of unit propagations updates the watched literals. SAT-IT also implements a *breakpoint system*, that lets the user indicate variables to be tracked. Then, a message appears every time one of these variables is assigned. In order to try the consequences of different decisions, SAT-IT implements *undo and redo* functionality. This allows to undo the execution process decision by decision, and pick another path (by manual decision selection). Finally, SAT-IT incorporates a preprocess of the input formulas. If the input CNF contains unit clauses, UP is applied until fix point. The formula is simplified accordingly, meaning that satisfied clauses are removed and falsified literals are removed from the remaining clauses. The literals propagated by this preprocess are listed in a line at the beginning of the trail history, with the superscript symbol p.

6. Conclusions an Future Work

We have presented SAT-IT, a visual and interactive tool to monitor and illustrate the basic search-based exact algorithms for SAT solving in a compact but detailed and very informative way. Whereas the tool can rapidly solve fairly large formulas, it has been designed to deal with formulas of few tens or hundreds of variables, so that the evolution of the solving process can be visually inspected. We believe that this tool can be really helpful to assist the teaching basics on SAT solving for different reasons. First, it is a very intuitive graphical tool that shows a good selection of the most relevant information involved in a SAT solving process. Second, the three included algorithms with increasing

sophistication allow the users to approach gently to the full CDCL algorithm. The used language and the trace of the algorithms are completely deterministic and the notation is also suitable for hand writing resolution of SAT formulas. These facts favours the use of the tool as an assistant to correct practical exercises and assist autonomous study.

The practical utility of the tool goes much beyond the learning of the CDCL algorithm or even teaching. For instance, SAT-IT can be used as an assistant for creating and validating examples for research reports. In particular, in papers on SAT modelling and solving, we can frequently find illustrative examples where an initial CNF formula is provided, and the inferences made by UP on the formula are then analysed. Similarly, SAT-IT can be used to analyse the properties of some SAT formulas or to find counterexamples. As an illustrative example we consider the analysis of the propagation strength of a SAT encoding. We are given a SAT formula consisting of the encoding of the cardinality constraint $x_1+x_2+x_3+x_4\geq 3$, and this encoding is supposed to maintain Generalized Arc Consistency (GAC) by UP [14]. Satisfying the GAC property implies, for instance, that assigning *false* to one of the three variables must make UP assign *true* to the other three. This scenario can be easily verified with SAT-IT by means of a manual decision followed by UP, and the user can inspect which clauses are involved in this process.

We plan to improve SAT-IT by including more features of CDCL SAT solvers such as restarts, learnt clause removal and bounded variable elimination, by providing graphic representation of unsatisfiability proofs, and by supporting MaxSAT solving algorithms.

References

[1] Cook SA. The Complexity of Theorem-Proving Procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. STOC '71; 1971. p. 151-158.
[2] Levin LA. Universal sequential search problems. Problemy peredachi informatsii. 1973;9(3):115-6.
[3] Kaufmann D, Biere A, Kauers M. Verifying Large Multipliers by Combining SAT and Computer Algebra. In: 2019 Formal Methods in Computer Aided Design (FMCAD); 2019. p. 28-36.
[4] Kautz HA, Selman B, et al. Planning as Satisfiability. In: ECAI. vol. 92. Citeseer; 1992. p. 359-63.
[5] Demirovic E, Musliu N, Winter F. Modeling and solving staff scheduling with partial weighted maxSAT. Ann Oper Res. 2019;275(1):79-99.
[6] Bofill M, Coll J, Garcia M, Giráldez-Cru J, Pesant G, Suy J, Villaret M. Constraint Solving Approaches to the Business-to-Business Meeting Scheduling Problem. J Artif Intell Res. 2022;74:263-301.
[7] Nieuwenhuis R, Oliveras A, Tinelli C. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM. 2006;53(6):937-77.
[8] Ben-Ari MM. LearnSAT: a SAT solver for education. In: International Conference on Theory and Applications of Satisfiability Testing. Springer; 2013. p. 403-7.
[9] Elffers J, Giráldez-Cru J, Gocht S, Nordström J, Simon L. Seeking Practical CDCL Insights from Theoretical SAT Benchmarks. In: Lang J, editor. Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI; 2018. p. 1300-8.
[10] Marques-Silva JP, Sakallah KA. GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers. 1999;48(5):506-21.
[11] Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S. Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference; 2001. p. 530-5.
[12] Eén N, Sörensson N. An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers 6. Springer; 2004. p. 502-18.
[13] Biere A, Heule M, van Maaren H, Walsh T, editors. Handbook of Satisfiability - Second Edition. vol. 336 of Frontiers in Artificial Intelligence and Applications. IOS Press; 2021.
[14] Bofill M, Coll J, Nightingale P, Suy J, Ulrich-Oltean F, Villaret M. SAT encodings for Pseudo-Boolean constraints together with at-most-one constraints. Artif Intell. 2022;302:103604.