SAT and Planning: an Overview

Carlos Camarão and Mateus Galvão and Newton Vieira
Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil

Abstract

This chapter firstly reviews the importance of the Satisfiability Problem (SAT) for a wide range of applications, including applications in Operation Management such as planning. A review of methods nowadays employed by modern SAT-solvers is then presented. We then use Classical Planning as an illustrative example of how a significant problem can be translated into SAT. We point out important results and studies concerning reductions of planning into SAT, and explain how to construct a SAT instance which is satisfiable if and only if an instance of a bounded version of the classic blocks-world problem is solvable.

Keywords artificial intelligence, propositional logic, the satisfiability problem, modern satisfiability solving, satisfiability algorithms, classical planning, planning algorithm, planning compilation into SAT

Author's biographies

Carlos Camarão de Figueiredo has a PhD in Computer Science by the Victoria University of Manchester, England (1994), where he developed a proof system for the formal verification of correctness of object-based programs. He is an Associate Professor at the Department of Computer Science of the Federal University of Minas Gerais, Brazil, where he has been working since 1996. His research interests include propositional satisfiability and applications, programming languages and functional programming.

Mateus Galvão de Melo is a undergraduate Computer Science student at the Federal University of Minas Gerais, Brazil, where he's been doing research around the satisfiability problem since 2007. His research interests include propositional satisfiability, artifficial intelligence and information recovery.

Newton José Vieira has a DSc degree in Computer Science by the Catholic University of Rio de Janeiro (PUC-Rio), Brazil (1987), where he developed an automated reasoning model for knowledge based systems. He is an Associate Professor at the Department of Computer Science of the Federal University of Minas Gerais, Brazil, where he has been working since 1980. His research interests include automated reasoning and knowledge based systems.

Twenty years ago the use of formal logic as a basis for knowledge representation and processing was subject to severe critics, founded not only on problems of expressivity, but also on problems of running time performance. Since that time the problem of expressivity has been object of intense research, which resulted in extensions and alternative formulations of logics intended to deal with important aspects of "intelligent" systems not dealt with by classical formal logic (the logic underlying purely mathematical reasoning) (Brachman & Levesque, 2004). From the point of view of running time performance, significant advancements have also been achieved (A. Robinson & Voronkov, 2001; Harmelen, Lifschitz, & Porter, 2008), even considering the undecidability of more expressive logics.

Although knowledge expression and processing in the area of artificial intelligence is still subject to intensive research, far from providing the necessary basis for its highest objectives (as the production of intelligent behaviour), the last years have seen a growing use of logic based systems for solving many industrial level applications in several domains.

In this chapter one of the simplest logics from the standpoint of expressivity will be considered: propositional logic. In spite of its simplicity, it is sufficient for the description and solution of an enormous amount of practical problems. Even considering that other logics are more adequate from a notational point of view, in order to have more concise descriptions for certain application classes, in many cases it is possible to produce automatically a translation to propositional logic (Cadoli & Schaerf, 2005; Navarro, 2007).

The justification for using a knowledge processor based on propositional logic, even in some situations where more expressive logics would be apparently more adequate (by having more concise sentences), is that the actual programs for propositional logic processing are so advanced that they allow the solution of many practical problem instances. They are typically not only more efficient than respective programs for other logics, but also than specific programs specially tailored to the application at hand (Gomes, Kautz, Sabharwal, & Selman, 2008).

The idea underlying the use of formal logic for problem solving is to obtain a solution for a problem instance from a given formal specification of the problem expressed in the language of that logic. There are two basic approachs for achieving this goal. In the approach based on *proof theory*, finding a solution consists of finding a proof for a formula α from a set of formulas Σ. For example, in the problems of software or hardware verification, Σ would be a formal description of an algorithm or a digital circuit and α a property to be verified. Another example: to verify the equivalence of two circuit descriptions given by two formulas α and β it is sufficient to establish the validity of the formula $\alpha \leftrightarrow \beta$ or, in other words, to prove that $\alpha \leftrightarrow \beta$ follows from the empty set of hypotheses. A huge number of other kinds of problems can be formulated as that of obtaining a proof of a formula (theorem) α from a set of formulas (axioms or hypoteses) Σ (Larry, 1984; Lifschitz, Morgenstern, & Plaisted, 2008).

The second approach is based on *model theory*. Here the goal is to find a model that satisfies a set of formulas Γ. In propositional logic, that corresponds to finding an assignment of truth values to the variables occurring in the formulas of Γ that makes all formulas true under the assignment. The problem of finding whether such assignment exists is what is called the *satisfiability problem* (SAT). Usually the algorithms for the satisfiability problem search, in a space of partial variable assignments, for a total assigment that satisfies the given set of formulas. An algorithm for SAT, as much as a proof procedure, can be used to find if a formula α is a logical consequence of a set of formulas Σ (which must be done, for example, to verify if a property follows from a set of specifications), as it is known that α follows from Σ if and only if $\Sigma \cup \{\neg\alpha\}$ *is not* satisfiable (is unsatisfiable). Consequently, to show that $\alpha \leftrightarrow \beta$ is valid (true for all possible assignments), it is sufficent to show that $\neg(a \leftrightarrow \beta)$ is unsatisfiable. It is interesting to note that if the algorithm finds that $\Sigma \cup \{\neg\alpha\}$ (in the first case) or $\neg(\alpha \leftrightarrow \beta)$ (in the second case) is satisfiable, then the generated satisfying assignment gives a counter-example that helps at diagnosing software/circuit faults.

The SAT problem was the first to be proved NP-complete (Cook, 1971). The strength of this class of

problems in general, and of SAT in particular, derives from the existence of many important practical problems that are NP-complete. As these problems are polynomial-time reducible to SAT, SAT algorithms with a good average-time performance are serious candidates to be applied at solving them. In fact, actually more and more problems are being tackled by means of modern SAT solvers, such as software (Chaki et al., 2003) and hardware (Velev & Bryant, 2001) verification, planning (Kautz & Selman, 1996), scheduling (Hoos, 2002) and many others. Actually, any NP-hard problem can eventually be solved through reduction or reformulation and use of some SAT solver sufficiently efficient to solve problem instances that really occur in practice. As already highlighted, it is remarkable that last years' advancements made it possible for some kinds of problems to be solved with performance comparable or even superior to that of solvers specially designed for them. For introductions to the theory of NP-completeness (and definitions of NP-complete and NP-hard problems), the reader may consult e.g. (Garey & Johnson, 1979; Sipser, 1997; Cormen, Leiserson, Rivest, & Stein, 2001).

The results obtained using SAT solvers have been so (surprisingly) good that some authors are foreseeing repercussions on the actual practice of algorithm design and analysis: the current excessive emphasis on the worst-case complexity analysis, that suggests the strict avoidance of non-polinomial perfomance, would not be appropriate any more (Gomes et al., 2008). Actually, such good results could be sensed early from the paper (D. G. Mitchell, Selman, & Levesque, 1992), which has shown that randomically generated difficult instances come from a relatively narrow zone of the space of possibilities. It was to be expected that for some classes of instances a specific approach to SAT solving could have a reasonable running time behaviour, with a worst-case exponential behaviour rarely or even never occurring.

What could have caused the excellent perfomance of modern SAT solvers to the point of opening the possibility of their use in the solution of very big instances that occur in industrial applications? Mitchell argues that the answer comes from three factors: better algorithms, improved implementation techniques and increased computer capacity (time and storage) (D. Mitchell, 2005). In particular, some algorithmic improvements (as, for example, heuristics and intelligent backtracking) are effective only with the computational resources recently available, which allow for efficient access to tens or hundreds of megabytes.

Modern SAT solving

Despite the worst-case exponential runtime of known SAT algorithms and despite the fact that SAT encodings generally substantially increase the size of the representation of a problem, modern SAT-solvers have been used effectively to solve very large problems, involving over a million variables. The approach of using SAT encodings and SAT solvers to solve large problems, from diverse areas, can be traced back to work on SAT published in the early nineties, but its use has increased significantly in the past few years, due to improvements in the performance of SAT solvers.

The work of (D. Mitchell, Selman, & Levesque, 1992) was pioneering in suggesting that, despite their worst-case exponential running time, SAT solvers could be used to solve quickly a large proportion of randomly generated SAT instances. Their experiments indicated that testing satisfiability (with a particular procedure, namely DPLL, see below) becomes less efficient, for any ratio of the number of clauses per number of variables, as the percentage of all formulas that are satisfiable approximates 50% (away from this "hard" area, satisfiability testing could be done efficiently).

In (Selman, Levesque, & Mitchell, 1992) a simple, incomplete, search method, called GSAT, was proposed, that solved instances in the hardest region effectively. GSAT's algorithm searches for a satisfying assignment of values to variables, starting from a random assignment and repeatedly changing the value assigned to a variable. A variable that leads to the largest increase in the number of satisfied clauses is chosen, until either a satisfying assignment is found or a fixed maximum number *m*

of changes is reached. In the latter case the method is incomplete, since it can fail to find a satisfying assignment when *m* is reached.

Since 2002, competitions of SAT solvers are organized yearly as a satellite event at the international conference on Theory and Applications of Satisfiability Testing (cf. http://www.satcompetition.org/, the international SAT Competitions web page). These have contributed to the improvement of implementations of many SAT solvers (e.g. (Silva & Sakallah, 1996; Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001; Gold- berg & Novikov, 2007; Huang, 2007; Selman et al., 1992)) and to the availability of large sets of SAT benchmarks (e.g. (Hoos & Stützle, 2000)).

This chapter presents the main techniques used in modern SAT solvers and also a general idea of the performance of modern SAT-solvers, in particular with respect to the running time these solvers spend for solving SAT models of Operation Management problems cited in the previous chapter, considering inputs of various sizes.

Techniques for solving SAT

Methods for SAT solving can be complete but also incomplete, according to whether the implemented function $sat : F \rightarrow Bool$ is a total or partial function. In other words: incomplete methods may fail to give an answer to some (hopefully few) formulas (problem instances). Complete methods are presented first, then incomplete ones. The motivation for the use of incomplete methods is that there are cases in which complete ones are not as efficient.

We use in the sequel common terms (like formula, clause, literal etc.) with their usual meaning (see e.g. (Garey & Johnson, 1979; Cormen et al., 2001)). In particular, a formula is a set of clauses and a clause is a set of literals. A literal is a variable or its negation. The negation of a literal α is denoted by $\bar{a}$. We use also the following functions:

$$tickle : (F , Set\ Lit) \rightarrow F$$
$$unitProp : F \rightarrow F$$

tickle (p,s) denotes the simplified formula obtained by removing from *p* all clauses with literals in *s* and deleting from all remainining clauses literals whose negations occur in *s*, and *unitProp(p)* is repeated tickling, until an empty clause is obtained or there are no more unit clauses.

$$tickle(p,s) = \left\{ x - \{\bar{k} \mid k \in s\} \mid x \in p \text{ and } x \cap s = \varnothing \right\}$$

$$unitProp(p) = \begin{cases} p & \text{if } \varnothing \in p \text{ or } s_1 = \varnothing \\ unitProp(tickle(p, s_1)) & \text{otherwise} \end{cases}$$
$$\text{where } s_1 = units(p)$$

$$units(p) = \left\{ k \mid \{k\} \in p \right\}$$

Complete Methods

Modern complete methods are still based, fundamentally, on the simple DPLL algorithm introduced in the early 1960's, which is based on selection and backtracking.

Martin Davis and Hilary Putnam (Davis & Putnam, 1960) defined in 1960 an algorithm, called DP,

that used the resolution rule:

$$\frac{\{a\} \cup x \quad \{\bar{a}\} \cup y}{x \cup y}$$

as the basic rule for satisfiability testing.

In executions of programs based on an unrestricted use of the resolution rule, a large amount of useless consequents of large size can be generated, causing memory consumption to explode. In 1962, an alternative to DP was defined by Martin Davis, George Logemann and Donald Loveland (Davis, Logemann, & Loveland, 1962), usually called DPLL (the name DP is sometimes used to refer to DPLL, confusingly), that used a very simple modification of DP which replaced the use of the resolution rule by a select-and-backtrack (also called *splitting*) rule that avoids the often occurring exponential memory explosion problem of DP.

DPLL is presented in Figure 1 in the form of a function *dpll : F → Bool.*

$$dpll(p) = \begin{cases} \text{true} & \text{if } p = \varnothing \\ \text{false} & \text{if } \varnothing \in p \\ \text{otherwise, selecting any } k \text{ in a minimum size clause of } p\text{:} \\ \quad \begin{cases} \text{true} & \text{if } dpll(p_1) \\ dpll(p_2) & \text{otherwise} \end{cases} \\ \quad \text{where} \quad p_1 = unitProp(p \cup \{k\}) \\ \qquad\qquad\; p_2 = unitProp(p \cup \{\bar{k}\}) \end{cases}$$

Figure 1. DPLL.

The name DPLL is also used for variations of this original scheme that selects any literal occurring in a clause of minimum size, such as the one in which selection involves choosing any unassigned literal. The process of identifying unit clauses and performing unit clause propagation is sometimes referred to *as boolean constraint propagation* (BCP) (Nadel, 2002).

The pure literal rule — which removes clauses with pure literals, where a pure literal is one such that its negation does not occur anywhere —, usually included in DPLL, has been omitted here. Eliminating pure literals may be, however, worthwhile (cf. e.g. (Steedman,1995)).

The main efficiency improvements made in modern SAT solvers, namely clause learning and efficient unit propagation, introduce modifications to this basic DPLL algorithm, described next.

Selection and Backtracking by Clause Learning. The heuristics used for selecting a variable and its value, also referred to as the *decision strategy* or *branching heuristic*, varies significantly among SAT solvers. However, advances in the efficiency of modern SAT solvers can be attributed, to a large extent, to a technique called *clause learning*, that uses information obtained as a result of the selection process in order to guide further selection, by adding clauses to the set of clauses being tested.

The most successful learning scheme, used by modern solvers (e.g. GRASP(Silva & Sakallah, 1996), zChaff(Moskewicz et al., 2001), BerkMin(Goldberg & Novikov, 2007), siege (Huang, 2007)), is based on detecting conflicts (assignments that cause the formula to become un- satisfiable) and on adding a new clause, called a *conflict cause*, to the set of clauses being tested, in order to avoid the conflict to happen again.

A conflict clause is determined by using an *implication graph*, which records implications between variables that arise due to the assignment of values to variables performed during SAT solving.

An implication graph is a labelled directed acyclic graph (DAG) where each node corresponds to a selected or implied literal (implied by unit propagation) and each edge from node *a* to node *b* corresponds to an implication ($a \rightarrow b$), as a consequence of existing clauses and selected assignments (*e.g.* $a \rightarrow b$ is an edge if clause $\{\overline{a}, b, c\}$ exists and $\overline{c}$ has been selected or, else, is implied by unit propagation). More precisely, we have (cf. (Gomes et al., 2008)):

Definition 1 *An implication graph G = (V, E), at a given stage of a SAT algorithm A, is a directed acyclic graph with nodes V and edges E, defined inductively as follows:*

- *initially, let V contain the set of selected literals and E be empty;*
- *for each clause* $c = \{a_1, ..., a_n\}$ *such that* $\overline{a_1}, ..., \overline{a_n}$ *are nodes in G, n ≥ 1, add node a to V and, for* $i = 1, ..., n$ *and add edges* $e_i = (\overline{a_i}, a_1)$ *to G.*

Example 1 *Let* $p = \{x_1, x_2, ..., x_6\}$, *where:*

$$x_1 = \{a_1, a_2\} \qquad x_2 = \{a_1, a_3, a_7\} \quad x_3 = \{\overline{a_2}, \overline{a_3}, a_4\}$$
$$x_4 = \{\overline{a_4}, a_5, a_8\} \quad x_5 = \{\overline{a_4}, a_6, a_9\} \quad x_6 = \{\overline{a_5}, \overline{a_6}\}$$

and consider literals selected in the sequence $\overline{a_9}, \overline{a_8}, \overline{a_7}, \overline{a_1}$. *We have then the following implication graph:*

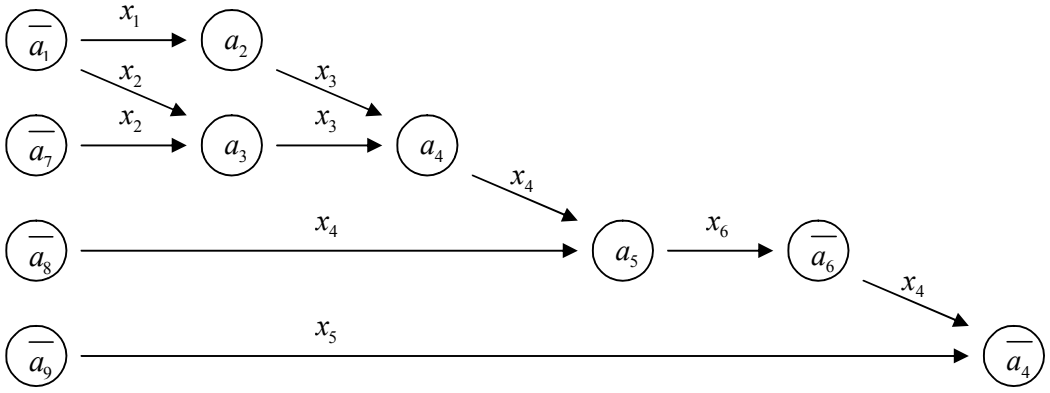

Figure 2. Implication Graph

A conflict occurs when both *a* and $\overline{a}$, called *conflicting literals*, occur in an implication graph (*a* is called a *conflicting variable*). An implication graph *G* contains a conflict if *a* and $\overline{a}$ are nodes of *G*, for some *a*, in which case *G* is said to contain a conflict graph (in other words, a conflict graph is a subgraph of *G* containing conflict literals *a* and $\overline{a}$, for some *a*).

Consider an implication graph *G = (V, E)*. Let a *cut* of a subset *s* of nodes of *G*, *cut (s)*, be the set of all

edges going from nodes in s to nodes not in *s*, i.e. *cut (s)* = $\{e \mid e = (a, b) \in E, b \in s, a \notin s\}$.

Given an edge (a,b), *a* is called a source and *b* a target of *e*.

Any cut $C = cut\ (s)$ of a subset *s* of a conflict graph *H* separates *H* in two parts, called the reason and the conflict sides of *H*, the reason side including nodes from a selected literal to a source of an edge in *C* and the conflict side including nodes from a target of an edge in *C* to a conflicting literal. The sources of edges in *C*, plus conflicting literals, if they are in *s*, are *causes* of the conflict (at the conflict side) given by *C*, denoted by *causes* (*C*). The set (representing the disjunction) of all negations of causes is a *conflict clause of C*, denoted *conflictClause* (*C*).

For example, considering Example 1 we have (where s_{i+1} adds to s_i , for i = 0, 1, 2, the targets of edges in C_i , plus conflicting literals, if they are already targets of an edge in C_i):

s_0	$= \{\overline{a_1}, \overline{a_7}, \overline{a_8}, \overline{a_9}\}$
$C_0 = cut(s_0)$	$= \{x_1, x_2\}$
$causes(C_0)$	$= \{\overline{a_1}, \overline{a_7}\}$
$conflictClause(C_0)$	$= \{a_1, a_7\}$
s_1	$= s_0 \cup \{a_2, a_3\}$
$C_1 = cut(s_1)$	$= \{x_3\}$
$causes(C_0)$	$= \{a_2, a_3\}$
$conflictClause(C_0)$	$= \{\overline{a_2}, \overline{a_3}\}$
s_2	$= s_1 \cup \{a_4\}$
$C_2 = cut(s_2)$	$= \{x_4, x_5\}$
$causes(C_2)$	$= \{a_4, \overline{a_8}, \overline{a_9}\}$
$conflictClause(C_0)$	$= \{\overline{a_4}, a_8, a_9\}$
s_3	$= s_2 \cup \{a_5, a_6\}$
$C_2 = cut(s_2)$	$= \{x_6\}$
$causes(C_2)$	$= \{a_5, a_6\}$
$conflictClause(C_0)$	$= \{\overline{a_5}, \overline{a_6}\}$

In order to choose a conflict clause, in a clause learning scheme, each selected literal is usually associated with a number, from 1 upwards, called the *level* of the selected literal. The level of an implied literal is the maximum of the levels of all selected literals that are used in its implication. The level of a node of an implication graph is the level of its literal. Level zero is used to characterize unsatisfiability, when a conflict that does not depend on any selection is detected.

The backtracking scheme used in modern SAT solvers consists of changing the value assigned to one or more variables occurring in a conflict clause. A SAT solver uses some way — called a *clause learning scheme* — of obtaining a conflict clause. For example, Grasp (Silva & Sakallah, 1996) and zChaff (Moskewicz et al., 2001) use *FirstUIP*, which is a scheme that identifies a node in the implication graph that is closest to the conflict variable and is a unique implication point (UIP). Letting *k* and l_k be respectively the currently selected literal and its level, in a conflict graph *H* (*k* is the last selected literal and l_k is the highest level in *H*), a UIP of *H* is a node (literal) *k'* with level l_k such that every path from *k* to a conflicting literal must pass through *k'* (*k* itself is always a UIP) (Silva & Sakallah, 1996). Note that a conflict clause, the singleton $\{\overline{k}\}$ ($\{a_1\}$ in Example 1) thus always exists

in a conflict graph. a_4 in our previous example is a *FirstUIP*.

A learning clause scheme enables distinct heuristics for performing and extending backtracking, in particular:

- *Backjumping* (also called *non-chronological backtracking*) is a term that generally refers to some way of performing backtracking by going directly to a previous point (level) in order to avoid irrelevant ones (Dechter & Frost, 1999). Several ways of performing *backjumping* have been used in SAT solvers.

Consider for example a situation in which n is any large positive number and the ordering of variables used for selecting values in variable assignment is:

$$a_1, b_1, b_2, ..., b_n, a_2, a_3$$

and consider clauses that include:

$$\{\ \{a_1, a_2, a_3\},\ \{a_1, \overline{a_2}, a_3\},\ \{b_1, b_2, ..., b_n\},$$
$$\{a_1, a_2, \overline{a_3}\},\ \{a_1, \overline{a_2}, a_3\},\ \{\overline{b_1}, \overline{b_2}, ..., \overline{b_n}\}\ \}$$

Any assignment ρ for which $\rho\ (a_1) = 0$ will lead to a contradiction, since any assignment of values to a2 and a3 will generate in this case a contradiction. If the selection scheme uses the variable ordering $[a_1, b_1, ..., b_n, a_2, a_3]$ and selection starts by assigning 0 to a_1, all 2^n assignments to b variables could be performed by the algorithm, all of them generating a contradiction. A naive backtracking strategy can take thus a very long time trying all such assignments before changing the assignment of a_1 to 1. To avoid this, modern SAT solvers use backjumping. Backjumping identifies, at a failed choice point (i.e. when both assignments, of *true* and *false*, to a variable, in our example a_2, lead to a contradiction), the causes of the conflicts (consisting of previously selected or implied variables, in our example the singleton $\{a_1\}$). In this case, backtracking changes the assignment of the variable with the greatest decision level (i.e. most recently selected). For example, Grasp and zchaff's backjumping heuristic allow backtracking to a lower decision level d when one of the branches that leads to a conflicting literal involves only literals at level d or lower with the exception of the literal with the current level (Gomes et al., 2008).

- SAT solvers can spend a lot of time in unproductive parts of a search tree. Many SAT solvers then randomly restart the search from scratch, with an empty assignment, taking into account the fact that searches will not be repeated due to a randomness that occurs in the selection process (Gomes, Selman, & Kautz, 1998).

Proposals to guarantee completeness in the presence of restarts, discussed in (Nadel, 2002), include: preserving all learnt clauses, repeatedly increasing the number of conflicts between consecutive restarts, preserving an increasing number of learnt clauses after each restart, and including a learnt clause with the negations of all decision variables that participated in the conflict in order to avoid the search path that led to the conflict to be taken again.

- As the set of learnt clauses increases, the ability to prune the search space increases but also does the cost of unit propagation. Heuristics that try to achieve a balance on the size of the set of learnt clauses include associating a measure of activity for each variable, that is increased each time a variable (and also a clause, as in (En & Sorensson, 2003)) is involved in a conflict (Moskewicz et al., 2001). Periodically removing inactive clauses and the more common use of variable activity to guide the selection process are heuristics used in e.g. (En & Sorensson, 2003).

<u>*Unit propagation with Watched Literals*</u>. Efficient unit propagation is a key to current fast SAT solvers. It has been observed by profiling that around 90% of SAT solvers runtime is spent in unit

propagation (Zhang, Madigan, Moskewicz, & Malik, 2001).

Almost all modern SAT solvers use an algorithm for unit propagation based on a simple technique of (so-called) *watched literals*, firstly used in the Chaff SAT solver (Moskewicz et al., 2001). The idea is to use, for each *active* (i.e. not yet satisfied) clause c, two literals k_1 and k_2 (called *watches* of c) such that $\overline{k_1}$ and $\overline{k_2}$ are not yet selected or implied (in other words, neither k_1 nor k_2 are *false*), and update information about watched literals whenever one of them, say k_2, is assigned to *false* (as a result of selection or implication, by unit propagation, of k_2), as follows:

1. k_1 is not implied; then there exists another literal k_3 in c that is not *false*, and k_1 and k_3 become the watched literals of c;
2. k_1 is implied (assigned to true); then $\overline{k_1}$ is assigned to *false* and unit propagation is done for k_1;

The watched literals technique is beneficial particularly because, upon backtracking, nothing has to be done or undone, since watched literals can stay the same.

The implementation of the watched literal scheme typically uses a *watch-list* for each literal k, that contains a list of clause references (numbers) for which k is a watch, and, for each clause reference (number), its pair of *watches*. When k is assigned to *false*, k's watch list is scanned and, for each encountered clause x, x's watches are updated.

Incomplete Methods

In principle, unsatisfiability could be searched but, in practice, most incomplete methods typically run with pre-set limits until a satisfying assignment is found or failure is reported, unsatisfiability being never reported. On problems of various domains, they run much faster than complete methods. Incomplete methods have been the subject of a lot of work since the early 1990's. References can be found in chapter 6 of the Handbook of Satisfiability, by Henry Kautz, Ashish Sabharwal and Bart Selman ((Kautz, Sabharwal, & Selman, 2008)), of which the material in this section is a condensed and adapted version.

The original work with incomplete local search methods was incentivated by experiments done with GSAT (Selman et al., 1992), which showed that global minima could be reached with local search strategies, in many cases much faster than with complete search methods. GSAT uses a very simple technique of choosing random assignments for all variables of a propositional formula p in cnf (conjunctive normal form, see e.g. (Prestwich, 2008)) and changing this assignment for the variable that leads to the greatest decrease in the number of unsatisfied clauses of p. GSAT is written in Figure 2 as a function $gsat : F \rightarrow Bool$, overloaded for clarity and for parametrization on the number (t) of tries (number of ran- dom assignments chosen), with maximum value *maxTries*, and on the number (n) of flips (i.e. variable assignment changes), with maximum value *maxFlips* (*fVs* below abbreviates, say, *flippedVars*).

Experiments have demonstrated that GSAT usually proceeds quickly towards a truth assignment but in many cases it spends a lot of time flipping variables without changing the number of unsatisfied clauses, before finding another assignment in which this number does decrease. One of the most successful strategies to speed up this process has been the one used in Walksat (Selman, Kautz, & Cohen, 1996), which tries to select a variable to flip from an unsatisfied clause, chosen at random and, if that leads current satisfied clauses to become unsatisfied, either i) a random variable in C is flipped or ii) a variable in C with lowest *break-down* (the number of satisfied clauses that becomes unsatisfied, according to the current assignment) is chosen. The choice between i) and ii) is controlled by a so-called *noise* parameter ρ, which is a value between 0 and 1 that controls how often i) is chosen, instead of ii).

$$gsat(p) = gsat(p,1)$$

$$gsat(p,t) = \begin{cases} fail & \text{if } t > maxTries \\ gsat(p,\sigma,t,1,\varnothing) & \text{otherwise, } \sigma \text{ random assignment for } p \end{cases}$$

$$gsat(p,\sigma,t,n,fVs) = \begin{cases} true & \text{if } \sigma \text{ makes } p \text{ satisfiable} \\ gsat(p,t+1) & \text{if } n > maxFlips \\ gsat(p,\sigma',t,n+1,fVs \cup \{v\}) & \text{otherwise} \\ \quad \text{where } v = flip(\sigma, p, fVs) & \\ \quad\quad \sigma' \text{ is } \sigma \text{ but with } \sigma(v) \text{ negated} & \end{cases}$$

Figure 3. gsat

walksat can be defined as *gsat* with only *flip* changed:

$$flip(\sigma, p, vs) =$$
let x be an unsatisfied clause of p, chosen at random, in
v if there exists v in x with break-down 0
otherwise, where i/0 is chosen with probability $\rho \in [0,1]$,
either i) a variable from x, chosen at random, or
ii) a variable from x with smallest break-down

Figure 4. walksat

Several other approaches for improving the performance of local search methods have been proposed, for which we refer interested readers to (Kautz et al., 2008).

Planning as SAT

Due to significant improvements in the efficiency of satisfiability solvers in recent years, many scientific and industrial problems have been modeled as a translation into SAT (Chaki et al., 2003; Velev & Bryant, 2001; Kautz & Selman, 1996; Hoos, 2002). Since the 1990s, software for solving Planning and Scheduling problems of Operations Management, in particular, have experienced great advances with this approach (Andrew & Baker, 1994; Kautz & Selman, 1996). This section gives a brief overview of the planning-as-SAT approach and explain the main issues involved in this approach.

Classical planning is a well-known problem in Artificial Intelligence and Operations Research. Planning is the problem of finding a sequence of actions that can be used for going from a initial to a final state, from given specifications of initial and final states and a set of action definitions, that specify changes of the state. Classical planning is its most basic form, where actions are deterministic.

Despite its importance in real world tasks, until the 1990s planners were not capable of solving real world instances, partly due to computer technology but mainly because of the algorithms used. During the 1990s dramatic advances occurred, with the introduction of three new methods: the GraphPlan algorithm (Blum & Furst, 1995), the compilation-to-SAT approach (first proposed by Kautz and

Selman in (Kautz & Selman, 1992), extended by the same authors with improved encodings and test results in (Kautz & Selman, 1996) and extended in many ways in (Kautz, Mcallester, & Selman, 1996; Ernst, Millstein, & Weld, 1997)) and by new heuristic search based techniques (described in e.g. (Bonet & Geffner, 2001; Hoffmann & Nebel, 2001)).

The basic idea behind using SAT for solving problems of planning is to restrict planning to practically interesting cases where the sequence of actions to reach a final state from the initial state is bounded, by some length n. The problem can then be encoded so that a propositional formula for a given n is satisfiable if and only if there exists a plan of such length n; a different value of n must be searched for in the case of unsatisfiability.

To date, lots of different encoding schemes for compilation into SAT have been proposed (e.g. (Ernst et al., 1997; Xing, Chen, & Zhang, 2006; N. Robinson, Gretton, Pham, & Sattar, 2008)), but the basic idea behind them is to construct a SAT instance along the lines described by means of the classic "blocks world" example presented in Figure 5 (cf. e.g. (Kautz & Selman, 1992; Kautz, McAllester, & Selman, 1996)).

$$
\begin{aligned}
&\mathit{init}: \quad \mathit{clear}(b_1) \wedge \mathit{clear}(b_2) \wedge \mathit{clear}(b_3) \wedge \\
&\qquad\quad \mathit{on}(b_1,t) \wedge \mathit{on}(b_2,t) \wedge \mathit{on}(b_3,t) \\
&\mathit{goal}: \quad \mathit{on}(b_1,b_2) \wedge \mathit{on}(b_2,b_3) \\
&\mathit{action}\ (\mathit{move}(b,x,y), \\
&\quad \mathit{pre}: \quad \mathit{on}(b,x) \wedge \mathit{clear}(b) \wedge \mathit{clear}(y) \\
&\quad \mathit{effect}: \quad \mathit{on}(b,y) \wedge \neg\mathit{on}(b,x) \wedge \mathit{clear}(x) \wedge \neg\mathit{clear}(y)) \\
&\mathit{action}\ (\mathit{moveToTable}(b,x), \\
&\quad \mathit{pre}: \quad \mathit{on}(b,x) \wedge \mathit{clear}(b) \\
&\quad \mathit{effect}: \quad \mathit{on}(b,t) \wedge \neg\mathit{on}(b,x) \wedge \mathit{clear}(x))
\end{aligned}
$$

Figure 5. Instance of the classic blocks-world planning problem

The blocks world problem consists of placing blocks, initially on a table (in Figure 5, blocks b_1, b_2, b_3 and table t), so that b_i becomes on top of b_{i+1} (i = 1, 2). Symbols ¬, ∧, ∨, → and ↔ are used in Figure 5 and in the sequel as the logical connectives of negation, conjunction, disjunction, implication and equivalence, respectively.

In the rather self-explanatory problem specification above, *init* marks the beginning of the specification of the initial state, *goal* of the final state and action of an *action* definition. Initial and final states are defined by propositions. In our example, a proposition consists of conjunctions of *predicate* applications (in general a proposition may use other logical connectives). Predicates are functions which return a boolean value, given elements in the domain of the problem instance (in our example, the domain of the problem instance consists of blocks b_i, i = 1, 2, 3, and table t). An action definition defines a predicate by means of:

1. a precondition: a proposition that must hold for the action to be taken. The definition of a precondition begins with the keyword *pre*.
2. an effect: a proposition that holds after the action is taken. The definition of an effect begins with the keyword *effect*.

A plan that solves the problem above in the smallest number of steps is the sequence of actions: [*move* (b_2 ,t ,b_3), *move* (b_1,t ,b_2)].

In order to translate a bounded version of the planning problem in terms of SAT, we have first to

choose a limit n to the length of a plan to reach the final state. Algorithms for choosing values of n in order to improve the efficiency of the process of finding a solution to the planning problem (if it exists) — which is a critical step of the planning-as-SAT approach — are presented in e.g. (Rintanen, 2009). If no solution exists, a different value of n may be tried; if there is no n such that the planning problem is solvable with a plan of length n, this process iterates forever. The approach is thus only useful for problems for which a solution exists. For problems for which it is not known whether a solution exists and, furthermore, for problems for which the planning problem requires a formula that grows exponentially with the size of the problem instance, another approach should in general be used (see e.g. (Stockmeyer & Meyer, 1973)). However, in practice most planning problems have a solution — a plan — whose length grows polynomially with the size of the problem.

Once n is chosen, a propositional formula A is constructed that consists of a conjunction of "instantiations" of each action definition a, for each stage $k = 0,...,n-1$ and for each possible combination of arguments for a. An instantiation of an action a with precondition p and effect e is an equivalence $a(x_1,...,x_m,k) \leftrightarrow p_k^l \rightarrow e_{k+1}^l \wedge F_{a_k}$, where $x_1,...,x_m$ is sequence (possible combination of arguments of a, pre_k^l is an instantiation of the precondition for a for these arguments and state k, $effect_{k+1}^l$ is an instantiation of the effects of a for these arguments and stage $k + 1$, and F_{a_k} are so-called "frame conditions", which are implications that state that predicate instantiations that are not altered remain in the stage as they were in the previous state.

The propositional formula that corresponds to the bounded planning problem is a conjunction $S_0 \wedge G \wedge A \wedge E$, where:

- S_0 represents the initial state,
- G the final goal state,
- A is as described above, and
- E asserts that at most one action occurs at each stage.

In our example, for the chosen value of n = 3 these proposition are as follows (where c abbreviates clear):

$$
\begin{aligned}
S_0 \;=\;& c(b_1,0) \wedge c(b_2,0) \wedge c(b_3,0) \wedge \\
& on(b_1,t,0) \wedge on(b_2,t,0) \wedge on(b_3,t,0) \\
G \;=\;& on(b_1,b_2,3) \wedge on(b_2,b_3,3) \\
A \;=\;& \text{combinations of } move(x,y,z,i) \text{ and } moveToTable(x,y,i), \\
& \text{for } i = 0,...,n-1 \text{ and arguments } x,y,z, \text{ including e.g.} \\
& move(b_2,t,b_3,0) \leftrightarrow \\
& \quad (on(b_1,t,0) \wedge c(b_2,0) \wedge c(b_3,0) \rightarrow \\
& \quad\; (on(b_2,b_3,1) \wedge \neg on(b_2,t,1) \wedge \\
& \quad\quad c(b_2,1) \wedge \neg c(b_3,1) \wedge (c(b_1,0) \leftrightarrow c(b_1,1)))) \\
E \;=\;& move(x,y,z,i) \vee moveToTable(x,y,i) \vee \\
& ((c(x,i) \leftrightarrow c(x,i+1)) \wedge (on(x,y,i) \leftrightarrow c(x,y,i+1))) \\
& \text{for combinations of arguments of } move, moveToTable, on, c \\
& \text{and for } i = 0,...,n\text{-}1
\end{aligned}
$$

The presence of action definitions with 3 arguments (like *move* in the example above) or more leads in general to the generation of a proposition that is too big to be processed efficiently, which would make

the approach impractical. However, there are techniques that may be used for reducing the number of parameters of action definitions and other techniques for reducing the number of variables and clauses of the generated proposition (cf. e.g. (Kautz, McAllester, & Selman, 1996)).

Conclusion

We have presented a brief overview of the problem of satisfiability of propositional formulas (SAT), which has many practical implications and is at the heart of the most important problem in theoretical computer science (the P versus NP question). There is a lot of research related to SAT, spanning from complexity analysis, heuristics, encodings of different problems into SAT, quantified boolean formulae, predicate logic and other kinds of logics, constraint programming etc. We have also given a gentle introduction to the approach of translating classical planning problems into SAT. The material aims at giving the reader a motivation for further studies in this vast and exciting field of research.

References

Andrew, J. C., & Baker, A. B. (1994). Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of the 12th National Conference on Artificial Intelligence* (pp. 1092–1097).

Blum, A. L., & Furst, M. L. (1995). Fast planning through planning graph analysis. *Artificial Intelligence*, *90* , 1636–1642.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*, 5–33.

Brachman, R. J., & Levesque, H. J. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann.

Cadoli, M., & Schaerf, A. (2005). Compiling problem specifications into SAT. *Artificial Intelligence*, *162*(1-2), 89–120.

Chaki, S., et al. (2003). Modular verification of software components in C. In *Proc. of the 25th ACM/IEEE International Conference on Software Engineering* (pp. 385–395).

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proc. of the 3rd Annual ACM Symposium on the Theory of Computing* (pp. 151–158).

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to Algorithms* (Second ed.). MIT Press.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM , 5* (7), 394–397.

Davis, M., & Putnam, H. (1960). A Computing Procedure for Quantification Theory. *J. ACM , 7* (3), 201–215.

Dechter, R., & Frost, D. (1999). *Backtracking algorithms for constraint satisfaction problems* (Tech. Rep.). Univ. of California, Irvine.

Ernst, M., Millstein, T., & Weld, D. (1997). Automatic SAT-Compilation of Planning Problems.In *Proc. of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)* (pp.1169–1176).

En, N., & Sörensson, N. (2003). An Extensible SAT-solver. In *Sat* (Vol. 2919, pp. 502–518). Springer.

Garey, M., & Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman.

Goldberg, E., & Novikov, Y. (2007). Berkmin: A fast and robust sat-solver. *Discrete Appl. Math., 155* (12), 1549–1561.

Gomes, C., Kautz, H., Sabharwal, A., & Selman, B. (2008). Satisfiability Solvers, chapter 2 of the Handbook of Knowledge Representation (Foundations of Artificial Intelligence). In (pp.89–134). Elsevier B.V.

Gomes, C., Selman, B., & Kautz, H. (1998). Boosting combinatorial search through randomization. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)* (pp. 431–437).

Harmelen, F. van, Lifschitz, V., & Porter, B. (Eds.). (2008). *Handbook of Knowledge Representation*. Elsevier.

Hoffmann, J., & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research , 14* , 253–302.

Hoos, H. (2002). An adaptive noise mechanism for WalkSAT. In *Proc. of the 18th National Conference on Artificial Intelligence (AAAI-2002)* (pp. 655–660).

Hoos, H., & Stützle, T. (2000). SATLIB: An Online Resource for Research on SAT. In *Proc. Of SAT 2000* (pp. 283–292). (http:/www.satlib.org)

Huang, J. (2007). A Case for Simple SAT Solvers. In *Proc. 13th international conf. on principles and practice of constraint programming (cp'07), lncs 4741* (pp. 839–846). Springer.

Kautz, H., Mcallester, D., & Selman, B. (1996). Encoding plans in propositional logic. In *Proc. of the 5th International Conference on the Principle of Knowledge Representation and Reasoning (KR'96)* (pp. 374–384).

Kautz, H., McAllester, D., & Selman, B. (1996). Encoding Plans in Propositional Logic. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96)* (p. 374-384).

Kautz, H., Sabharwal, A., & Selman, B. (2008). Incomplete algorithms, chapter 15 of the Handbook of Satisfiability. In (pp. 185–203). IOS Press.

Kautz, H., & Selman, B. (1992). Planning as satisfiability. In *Proc. of the 10th European Conference on Artificial intelligence (ECAI'92)* (pp. 359–363). John Wiley & Sons, Inc.

Kautz, H., & Selman, B. (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of 13th the National Conference on Artificial Intelligence (AAAI'96)* (pp. 1194–1201).

Larry, R. O. (1984). *Automated reasoning: Introduction and applications*. Unknown. Hardcover. Lifschitz, V., Morgenstern, L., & Plaisted, D. (2008). Knowledge Representation and Classical Logic, chapter 1 of the Handbook of Knowledge Representation (Foundations of Artificial Intelligence). In (pp. 3–88). Elsevier B.V.

Mitchell, D. (2005). A SAT Solver Primer. *EATCS Bulletin , 85* , 112–133.

Mitchell, D., Selman, B., & Levesque, H. (1992). Hard and Easy Distributions of SAT Problems. In *Proc. of aaai (association for the advancement of artificial intelligence) 92* (pp. 459–465).

Mitchell, D. G., Selman, B., & Levesque, H. (1992). Hard and easy distributions of SAT solvers. In *Proc. of the 10th National Conference on Artificial Intelligence (AAAI-92)* (pp. 459–465).

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proc. Design Automation Conference* (pp. 530–535).

Nadel, A. (2002). *Backtrack Search Algorithms for Propositional Logic Satisfiability: Review and Innovations*. Unpublished master's thesis, Univ. of Tel-Aviv.

Navarro, J. A. (2007). *Encoding and Solving Problems in Effectively Propositional Logic*. Unpublished doctoral dissertation, University of Manchester.

Prestwich, S. (2008). CNF Encodings, chapter 2 of the Handbook of Satisfiability. In (pp. 75–97). IOS Press.

Rintanen, J. (2009). Planning as SAT, chapter 15 of the Handbook of Satisfiability (Volume 185, Frontiers in Artificial Intelligence and Applications). In (pp. 483–504). IOS Press.

Robinson, A., & Voronkov, A. (Eds.). (2001). *Handbook of Automated Reasoning (vols. I & II)*. The MIT Press.

Robinson, N., Gretton, C., Pham, D. N., & Sattar, A. (2008). A compact and efficient sat encoding for planning. In *Icaps* (p. 296-303).

Selman, B., Kautz, H., & Cohen, B. (1996). Local Search Strategies for Satisfiability Testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (Vol. 26, pp. 521–532).

Selman, B., Levesque, H., & Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proc. of aaai 92* (pp. 440–446).

Silva, J. P. M., & Sakallah, K. (1996). Grasp — a new search algorithm for satisfiability. In *ICCAD '96: Proc. of the 1996 IEEE/ACM international conference on Computer-aided design* (pp. 220–227).

Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.

Steedman, M. (1995). *Improvements to Propositional Satisfiability Search Algorithms*. Unpublished doctoral dissertation, Univ. of Pennsylvania.

Stockmeyer, L. J., & Meyer, A. R. (1973). Word problems requiring exponential time. In *Proc. of the 5th Annual ACM Symposium on the Theory of Computing* (pp. 1–9).

Velev, M. N., & Bryant, R. E. (2001). Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proc. of the 38th Conference on Design Automation* (pp. 226–231).

Xing, Z., Chen, Y., & Zhang, W. (2006). MaxPlan: Optimal planning by decomposed satisfiability and backward reduction. In *Proc. of the international planning competition (ipc)* (pp. 53–56).

Zhang, L., Madigan, C., Moskewicz, M., & Malik, S. (2001). Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. IEEE/ACM International Conference on Computer- Aided*

Design (pp. 279–285).