Solving Hard Computational Problems using Propositional Logic

I. Lynce

Technical Report 12/2005
August 2005
Abstract

The synergy between logic and computational complexity has gained importance and vigor in recent years, cutting across many areas. Several machine-independent approaches to computational complexity have been developed, which characterize complexity classes by conceptual measures borrowed primarily from mathematical logic.

Propositional SATisfiability (SAT) is well-known for its theoretical importance. SAT was the first problem to be proved to be NP-complete. Although SAT looks easy on its formulation, SAT is hard on solving. Although propositional logic is a logic with very limited expressiveness, finding a solution or proving unsatisfiability has an increasing complexity as the size of the problem increases.

The area of propositional satisfiability has been the subject of intensive research in recent years, with significant theoretical and practical contributions. From a practical perspective, a large number of very effective SAT solvers has recently been proposed, most of which based on a backtrack search algorithm. In addition, these recent algorithms utilize advanced conflict analysis procedures, recording the cause of failure and therefore allowing the search to backtrack non-chronologically. The new solvers are capable of solving very large, very hard real-world problem instances, which more traditional SAT solvers are totally incapable of.

The progress in SAT solving has attracted the attention of researchers that usually use other technologies to solve their problems. With this work we propose to convert into SAT extended forms of propositional logic (also known as SAT extensions). These logics are frequently more adequate to represent most of the problems. Whilst many problems are traditionally represented using SAT extensions, we argue that converting these representations into SAT and solving the resulting problem instances using a state-of-the-art SAT solver is a promising approach.

1 Introduction

Logic has found application in virtually all aspects of computing, from software engineering and hardware to programming and artificial intelligence. As a result of the interaction between logic and computing the following areas of interest have emerged: logical systems, logical issues in logic programming, knowledge-based systems and automated reasoning, logic and semantic of programming, and specification and verification of programs and systems, among others.

A computation is an operation that begins with some initial conditions and gives an output which follows from a definite set of rules. The most common example are computations performed by computers, in which the fixed set of rules may be the functions provided by a particular programming language. A set of rules used to carry out a computation is known as an algorithm.

The complexity of an algorithm is a measure of how difficult it is to perform. The study of the complexity of algorithms is known as complexity theory. Complexity theory is part of the theory of computation dealing with the resources required during computation to solve a given problem. One of the most common resources measured is time. The time complexity of a problem is the number of steps that it takes to solve an instance of the problem as a function of the size of the input, using
the most efficient algorithm. Other resources can also be considered, such as space, i.e. how much memory a computation takes.

In computational complexity theory, a complexity class is a set of problems of related complexity. For example, a problem is assigned to the NP-problem (nondeterministic polynomial time) class if it permits a nondeterministic solution and the number of steps to verify the solution is bounded by some power of the problem’s size.

Propositional Satisfiability (SAT) was the first problem shown to be NP-complete [4]. SAT is both a cornerstone of computational complexity theory and commercially important since thousands of practical combinatorial problems currently benefit from a highly efficient SAT solver. SAT has many different applications: automatic test pattern generation, combinational equivalence checking, bounded model checking, planning, graph coloring, software verification, etc.

Propositional logic has been seen in the past as a very restrictive logic, due to its limited expressiveness. However, a recent wave of improvements in SAT algorithms makes SAT a competitive approach for solving hard computational problems. Hence, we claim that converting into SAT problems which are traditionally encoded using more sophisticated logics and afterwards solving them with a SAT solver is indeed a competitive approach.

This document has to main goals: (1) to show the importance of SAT is in the area of “Logic and Computational Complexity” and (2) to motivate reasearchers to further use SAT technology to solve hard computational problems. We believe we have an important contribution to give to this specific topic, due to our knowledge and experience in SAT research.

This document is organized as follows. The next two sections are an introduction to Logic and Computational Complexity, respectively. Afterwards, we provide a background on propositional satisfiability, namely the basic definitions and the most often used algorithm. Section 5 illustrates how to encode hard computational problems into SAT. Finally, we present our proposal for future research work.

2 Logic

"I know what you’re thinking about,” said Tweedledum; ”but it isn’t so, nohow.”

"Contrariwise,” continued Tweedledee, ”if it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.”

*Lewis Carroll, in Through the Looking-Glass, 1872*

If we take logic to be the activity of drawing inferences (conclusions) from a body of knowledge, then no doubt humans have been using logic for as long as they have been thinking. On the other hand, if we take logic to be the analysis of concepts involved in making inferences, and the identification of standards and patterns of correct inference, then logic can be traced only back to the days of Aristotle (384-322 BC), with some parallel development in early Hindu writings. Aristotle was the
first to start writing down the ideas and rules of what constitutes a logical process. However, it is not clear that this increase in logical self-consciousness improved the accuracy of reasoning processes for humankind in general.

The heart of Aristotle’s logic is the syllogism. The classic example of syllogism is as follows: All men are mortal; Socrates is a man; therefore, Socrates is mortal. The core of this definition is the notion of “resulting of necessity” (*ex ananks sumbainein*). This corresponds to a modern notion of logical consequence: X results of necessity from Y and Z if it would be impossible for X to be false when Y and Z are true. Aristotle’s logical work contains the earliest formal study of logic that we have knowledge of. It is therefore a remarkable work that comprises a highly developed logical theory, one that was able to dominate logic for many centuries. Indeed, the syllogistic form of logical argumentation dominated logic for 2,000 years.

Historically, René Descartes (1596-1650) may have been the first mathematician to have had the idea of using algebra, especially its techniques, for solving unknown quantities in equations, as a vehicle for scientific exploration. However, the idea of a calculus of reasoning was cultivated especially by Gottfried Wilhelm Leibniz (1646-1716). Though modern logic in its present form originates with Boole and De Morgan, Leibniz was the first to have a really distinct plan of a broadly applicable system of mathematical logic. However, this information is in Leibniz’s unpublished work, which has only recently been explored.

Logic’s serious mathematical formulation began with the work of George Boole (1815-1864) in the mid-1800s. Boole made significant contributions in several areas of mathematics, but was immortalized for his book *An Investigation of the Laws of Thought* written in 1854, in which he represented logical expressions in a mathematical form now known as Boolean algebra. Boole’s work is so impressive because, with the exception of elementary school and a short time in a commercial school, he was almost completely self-educated. Unfortunately, with the exception of students of philosophy and symbolic logic, Boolean algebra was destined to remain largely unknown and unused for the better part of a century.

In conjunction with Boole, another British mathematician, Augustus De Morgan (1806-1871), formalized a set of logical operations now known as De Morgan laws. However, the rules we now attribute to De Morgan were known in a more primitive form by William of Ockham (also known as William of Occam) in the 14th Century.

Gottlob Frege (1848-1925) in his 1879 *Begriffsschrift* extended formal logic beyond propositional logic to include constructors such as ”all” and ”some”. He showed how to introduce variables and quantifiers to reveal the logical structure of sentences, which may have been obscured by their grammatical structure. For instance, ”All humans are mortal” becomes ”All things x are such that, if x is a human then x is mortal.”

Charles Peirce (1839-1914) introduced the term ”second-order logic” and provided us with most of our modern logical notation, including the symbols ∀ and ∃. Although Peirce published his work some time after the *Begriffsschrift*, Frege’s contribution was not very well known until many years
later. Logicians in the late 19th and early 20th centuries were thus more familiar with Peirce’s system of logic (although Frege is generally recognized today as being the "Father of modern logic").

In 1889 Giuseppe Peano (1858-1932) published the first version of the logical axiomatization of arithmetic. Five of the nine axioms he came up with are now known as the Peano axioms. One of these axioms was a formalized statement of the principle of mathematical induction.

In 1938 a young student called Claude E. Shannon (1916-2001) recognized Boolean algebra’s relevance to digital electronics design. In a paper based on his master’s thesis at MIT, "A Symbolic Analysis of Relay and Switching Circuits" (published in the Transactions of the American Institute of Electrical Engineers, volume 57, pages 713-723), which was widely circulated, Shannon showed how Boole’s concepts of TRUE and FALSE could be used to represent the functions of switches in electronic circuits. It is difficult to convey just how important this concept was; suffice is to say that Shannon had provided electronics engineers with the mathematical tool they needed to design digital electronic circuits, and these techniques remain the cornerstone of digital electronic design to this day.

In 1948, Brattain, Bardeen and Shockley, working at the Bell Telephone Laboratories, published their invention of the transistor. Bardeen, Shockley and Brattain shared the 1956 Physics Nobel Prize for this invention. Later on, in 1959, the first planar transistor was produced. The revolution in integrated circuits has accelerated the automation of information technology that we enjoy today.

3 Computational Complexity

Many of the games and puzzles people play are interesting because of their difficulty: it requires cleverness to solve them. Often this difficulty can be measured mathematically, in the form of complexity.

The complexity of a process or algorithm is a measure of how difficult it is to perform. The study of the complexity of algorithms, also known as complexity theory, deals with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem). Other resources can also be considered, such as how many parallel processors are needed to solve a problem in parallel.

The time complexity of a problem is the number of steps that it takes to solve an instance of the problem, as a function of the size of the input, using the most efficient algorithm. Table 1 has the CPU time required for solving different functions, thus giving a picture of time complexity.

To further understand time complexity intuitively, consider the example of an $n$ size instance that can be solved in $n^2$ steps. For this example we say that the problem has a time complexity of $n^2$. Of course, the exact number of computer instructions will depend on what machine or language is being used. To avoid this dependency problem, we generally use the Big O notation. If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$ on most
other computers, so this notation allows us a generalization away from the details of a particular computer. In this case, we say that this problem has polynomial time complexity. This is also true for all problems having $O(n^x)$ time complexity, with $x$ being a constant.

The space complexity of a problem defines how much memory does it take to solve a problem. For example, consider an $n$ size instance that can be solved using $2^n$ memory units. Hence, this problem has $O(n)$ space complexity and therefore linear complexity.

Much of complexity theory deals with decision problems. A decision problem is a problem where the answer is always YES/NO. For example, the problem IS-PRIME is: given an integer, return whether it is a prime number or not. Decision problems are often considered because an arbitrary problem can always be reduced to a decision problem.

Decision problems fall into sets of comparable complexity, called complexity classes. P and NP are the most well-known complexity classes, meaning Polynomial time and Nondeterministic Polynomial time, respectively.

The complexity class P is the set of decision problems that can be solved by a deterministic machine with a number of steps bounded by a power of the problem’s size. This class corresponds to an intuitive idea of problems which can be effectively solved even in the worst cases.

The complexity class NP is the set of decision problems with a nondeterministic solution and with the number of steps to verify the solution being bounded by a power of the problem’s size. In other words, all the problems in this class have the property that their solutions can be checked effectively in polynomial time. The complexity class Co-NP is the set of decision problems where the NO instances can be checked effectively in polynomial time. The Co in the name stands for complement.

The class of P-problems is a subset of the class of NP-problems. The question of whether P is the same set as NP is the most important open question in theoretical computer science. There is even a $1,000,000 prize for solving it (see http://www.claymath.org/millennium/). Observe that if P and NP are not equivalent, then finding a solution for NP-problems requires an exhaustive search in the worst case.
The question of whether \( P = NP \) motivates the concepts of \textit{hard} and \textit{complete}.

A set of problems \( X \) is \textit{hard} for a set of problems \( Y \) if every problem instance in \( Y \) can be transformed easily (i.e. in polynomial time) into some problem instance in \( X \) with the same answer. The most important hard set is NP-hard. A problem is said to be NP-hard if an efficient algorithm for solving it can be translated into one for solving any other NP-problem. In general, is easier to show that a problem is NP than to show that it is NP-hard.

Set \( X \) is \textit{complete} for \( Y \) if it is hard for \( Y \), and is also a subset of \( Y \). The most important complete set is NP-complete. A NP-complete problem is both NP (verifiable in nondeterministic polynomial time) and NP-hard (any other NP-problem can be translated into this problem). SAT is an example of an NP-complete problem \([4]\).

4 \ Propositional Satisfiability (SAT)

Propositional SATisfiability (SAT) can be simply characterized with a couple of words: propositional logic and computational complexity. SAT looks easy but is hard. Easy on its formulation: SAT problems are encoded in propositional logic, a logic with very limited expressiveness. Hard on solving: finding a solution or proving unsatisfiability has an increasing complexity as the size of the problem increases.

SAT is well-know for its remarkable improvements in the last decade. SAT solvers are now capable of efficiently solving instances with hundreds of thousands of variables and millions of clauses. State-of-the-art SAT solvers can now very easily solve problem instances that more traditional SAT solvers are known to be totally incapable of. For example, Table 2 gives a picture of how SAT has evolved in the last decade, showing how problem instances have been effectively solved as new solvers appeared.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Posit’ 94</th>
<th>Grasp’ 96</th>
<th>Sato’ 98</th>
<th>Chaff’ 01</th>
<th>Siege’03</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssa2670-136</td>
<td>40.66 s</td>
<td>1.2 s</td>
<td>0.95 s</td>
<td>0.02 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>bf1355-638</td>
<td>1805.21 s</td>
<td>0.11 s</td>
<td>0.04 s</td>
<td>0.01 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>pret150,25</td>
<td>&gt;7200 s</td>
<td>0.21 s</td>
<td>0.09 s</td>
<td>0.01 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>dubois100</td>
<td>&gt;7200 s</td>
<td>11.85 s</td>
<td>0.08 s</td>
<td>0.01 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>aim200-2,0-no-1</td>
<td>&gt;7200 s</td>
<td>0.01 s</td>
<td>0 s</td>
<td>0 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>2dlx_cc mc_ex_bp_f2_bug005</td>
<td>&gt;7200 s</td>
<td>&gt;7200 s</td>
<td>&gt;7200 s</td>
<td>2.9 s</td>
<td>0.22 s</td>
</tr>
<tr>
<td>c6288</td>
<td>&gt;7200 s</td>
<td>&gt;7200 s</td>
<td>&gt;7200 s</td>
<td>&gt;7200 s</td>
<td>6676.17 s</td>
</tr>
</tbody>
</table>
4.1 Definitions

A conjunctive normal form (CNF) formula \( \varphi \) on \( n \) binary variables \( X = \{x_1, \ldots, x_n\} \) is the conjunction of \( m \) clauses \( \Omega = \{\omega_1, \ldots, \omega_m\} \) each of which is the disjunction of one or more literals, where a literal is the occurrence of a variable \( x \) or its complement \( \neg x \). A clause having a variable and its complement is called a tautology and is always satisfied regardless the given assignment. A clause with no literals is an empty clause and is always unsatisfied. A clause with one literal is called unit, with two literals is called binary and with three literals is called ternary. Moreover, a literal is pure if its complement does not occur in the formula. For a CNF formula \( \varphi \) and for each clause \( \omega \) we can also use set notation. Hence, \( \omega \in \varphi \) means that clause \( \omega \) is a clause of formula \( \varphi \), and \( l \in \omega \) means that \( l \) is a literal of clause \( \omega \). Often, variables are also denoted by a sequence of alphabetically ordered letters, e.g. \( a, b, c, \ldots \) or \( x, y, z \).

A formula \( \varphi \) denotes a unique \( n \)-variable Boolean function \( f(x_1, \ldots, x_n) \) and each of its clauses corresponds to an implicate of \( f \). Clearly, a function \( f \) can be represented by many equivalent formulas. A binary variable can be assigned a truth value \( v \) which may assume value 0 (or false/F) or 1 (or true/T). Also, clauses and formulas may assume values depending on the values of the corresponding literals and clauses, respectively. The SAT problem is concerned with finding an assignment to the arguments of \( f(x_1, \ldots, x_n) \) that makes the function equal to 1 or proving that the function is equal to the constant 0.

A truth assignment \( A_X : X' \subseteq X \rightarrow \{true, false\} \) for a formula \( \varphi \) is a subset of assigned variables \( X' \) and their corresponding binary values.

An assignment \( A_X \) partitions the clauses of \( \varphi \) into three sets: satisfied clauses having at least one literal assigned value 1; unsatisfied clauses having all literals assigned value 0; and unresolved otherwise. The unassigned literals of a clause are referred to as its free literals. In a search context, a clause is said to be unit if the number of its free literals is one. Similarly, a clause with two free literals is said to be binary and a clause with three free literals is said to be ternary.

Evaluating a formula \( \varphi \) for a given truth assignment \( A_X \) yields three possible outcomes: all the clauses are satisfied and we say that \( \varphi \) is satisfied and refer to \( A_X \) as a satisfying assignment; at least one clause is unsatisfied in which case \( \varphi \) is unsatisfied and \( A_X \) is referred to as an unsatisfying or conflicting assignment; and the value of \( \varphi \) is undefined, i.e. cannot be resolved by the assignment.

4.2 Backtrack Search SAT Algorithm

Over the years a large number of algorithms has been proposed for SAT, from the original Davis-Putnam (DP) procedure [6], followed by the Davis-Logemann-Loveland (DLL) procedure [5], to recent backtrack search algorithms [10, 2, 9, 16, 11, 7, 12] and to local search algorithms [14, 13], among many others. Local search algorithms can solve extremely large satisfiable instances of SAT. These algorithms have also been shown to be very efficient on randomly generated instances of SAT. On the other hand, several improvements to the DLL backtrack search algorithm have been
introduced. These improvements have been shown to be crucial for solving large instances of SAT derived from real-world applications, and in particular for those where local search cannot be applied, i.e. for unsatisfiable instances. Indeed, proving unsatisfiability is often the objective in a large number of significant real-world applications.

Typically, the backtrack search algorithm is implemented by a search process that implicitly enumerates the space of $2^n$ possible binary assignments to the $n$ problem variables. Starting from an empty truth assignment, a backtrack search algorithm enumerates the space of truth assignments implicitly and organizes the search to find a satisfying assignment by searching a decision tree. Each node in the decision tree specifies an elective assignment to an unassigned variable; such assignments are referred to as decision assignments. A decision level is associated with each decision assignment to denote its depth in the decision tree; the first decision assignment at the root of the tree is at decision level 1. Assignments made before the first decision, i.e. during preprocessing, are assigned at decision level 0. For each new decision assignment, the decision level is incremented by 1. After each decision assignment, the unit clause rule [6] is applied iteratively, i.e. BCP is applied. Also, each implied assignment is associated with an explanation, i.e. with the unit clause that implied the assignment.

Algorithm 1 gives the pseudo-code for a DLL-based backtrack search algorithm. Given a SAT problem, formulated as a CNF formula $\varphi$, the algorithm conducts a search through the space of all possible assignments to the $n$ problem variables. At each stage of the search, a variable assignment is selected with the Decide function. A decision level $d$ is then associated with each selection of an assignment. Implied assignments are identified with the Deduce function, which in most cases corresponds to the straightforward derivation of implications by applying BCP. Whenever a clause becomes unsatisfied, the Deduce function returns a conflict indication which is then analyzed using

```plaintext
Algorithm 1 DLL-based backtrack search algorithm

SAT(\varphi)
(1)  d = 0
(2)  while Decide(\varphi, d) == DECISION
(3)    if Deduce(\varphi, d) == CONFLICT
(4)      \beta = Diagnose(\varphi, d)
(5)      if \beta == -1
(6)        return UNSATISFIABLE
(7)      else
(8)        BACKTRACK(\varphi, d, \beta)
(9)      else
(10)     d = \beta
(11)    else
(12)   d = d + 1
(13) return SATISFIABLE
```

...
the **Diagnose** function. The diagnosis of a given conflict returns a backtracking decision level \( \beta \), which denotes the decision level to which the search process is required to backtrack to. Afterwards, the **Backtrack** function clears all assignments (both decision and implied assignments) from the current decision level \( d \) through the backtrack decision level \( \beta \). Furthermore, considering that the search process should resume at the backtrack level, the current decision level \( d \) becomes \( \beta \). Finally, the current decision level \( d \) is incremented. This process is interrupted whenever the formula is found to be satisfiable or unsatisfiable. The formula is satisfied when all variables are assigned (meaning \( \text{Decide}(\varphi, d) \neq \text{DECISION} \)) and therefore all clauses must be satisfied. The formula is unsatisfied when the empty clause is derived, which is implicit when the **Diagnose** function returns \(-1\) as the backtrack level.

Given the above description, it is usual to decompose the backtrack search algorithm into three main engines:

1. The decision engine **Decide**, which selects a decision assignment at each stage of the search. (\( \text{DECISION} \) is returned unless all variables are assigned or all clauses are satisfied.) Observe that selected variable assignments have no explanation. This engine is the basic mechanism for exploring new regions of the search space.

2. The deduction engine **Deduce**, which identifies assignments that are deemed necessary, usually called implied assignments. Whenever a clause becomes unsatisfied, implying that the current assignment is not a satisfying assignment, we have a conflict, and the associated unsatisfying assignment is called a conflicting assignment. The **Deduce** function then returns a conflict indication which is then analyzed using the **Diagnose** function.

3. The diagnosis engine **Diagnose**, which identifies the causes of a given conflicting partial assignment. The diagnosis of a given conflict returns a backtracking decision level, which corresponds to the decision level to which the search must backtrack. This backtracking process is the basic mechanism for retreating from regions of the search space where satisfying assignments do not exist.

## 5 Encoding Hard Computational Problems into SAT

When given a hard computational problem, one has two major tasks: to model the problem and to solve the selected model. The progress in SAT solving has attracted the attention of researchers that usually use other technologies to solve their problems. Although more sophisticated logics are frequently more adequate to represent most of the problems, encoding problems in CNF format and solving them with SAT solvers is indeed a competitive approach. SAT has the advantage of being very easy in its formulation. Nonetheless, the simplicity of the CNF format makes its use very restrictive. Consequently, encoding such problems as CNF formulas may require a significant effort. Hopefully, this effort will be counterbalanced by the performance of SAT solvers.
To encode a problem into SAT one must define a set of variables and a set of constraints on the variables. The set of variables may be defined based on different criteria: the most intuitive variables set, the set with minimum cardinality, the set that will require the smallest number of clauses, etc. Choosing the most adequate variables is more an art than a science. Moreover, the definition of the set of constraints may require the definition of additional auxiliary variables. In some cases, these variables are really essential; in other cases, we prefer to have more variables rather than more clauses.

Combinational electronic circuits are used as the most paradigmatic example of encoding into SAT. For example, given the combinational circuit in Figure 1, it is clear that it can be easily encoded in propositional logic as follows:

\[ (a \lor b) \land (\neg a \lor \neg b) \]

In what follows we give different examples of problems that can be easily encoded as SAT problems and further effectively solved by state-of-the-art SAT solvers.

5.1 Combinational Equivalence Checking

The combinational equivalence problem consists in determining whether two given digital circuits implement the same Boolean function. This problem arises in a significant number of computer-aided design (CAD) applications, for example when checking the correctness of incremental design changes (performed either manually or by a design automation tool).

Equivalence checking is a Co-NP complete problem. Equivalence checking can be solved using SAT by identifying a counterexample. SAT-based equivalence checking builds upon a miter circuit. A miter circuit consists of two circuits \( C_1 \) and \( C_2 \) and also a set of XOR gates and an OR gate. Consider that the outputs of \( C_1 \) are \( O_{11}, \ldots, O_{1m} \) and the outputs of \( C_2 \) are \( O_{21}, \ldots, O_{2m} \). Hence, the miter circuit has \( m \) XOR gates, and the input of an XOR\( _i \) gate is \( O_{1i} \) and \( O_{2i} \), where \( i = 1, \ldots, m \). Finally, an OR gate links all the outputs of the XOR gates. A generic miter circuit is given in Figure 2. The output of the miter is 1 if the two circuits represent different Boolean functions. Hence, adding the objective \( O = 1 \) to the CNF encoding makes the SAT instance unsatisfiable iff \( C_1 \) and \( C_2 \) are equivalent.

Figure 3 gives an example of a miter including the circuit from Figure 1, which is supposed to be equivalent to an XOR gate. Clearly, for this miter circuit there is no need to add an OR gate.
To encode the circuit given above, one has first to consider the encoding of the combinational gates AND, OR and XOR in the CNF format given in Figure 4. Considering those encodings, the miter circuit may be encoded in a CNF formula by defining a set of clauses for each gate as follows [15]:

1. \((a \lor -b \lor -c)(-a \lor b \lor c)(a \lor -b \lor c)(a \lor b \lor -c)\)
2. \((a \lor b)(-b \lor d)(a \lor b \lor -d)\)
3. \((a \lor -e)(b \lor e)(-a \lor -b \lor -e)\)
4. \((d \lor -f)(e \lor -f)(-d \lor -e \lor f)\)
5. \((c \lor -f \lor -g)(-c \lor f \lor g)(c \lor -f \lor g)(c \lor f \lor -g)\)
6. \((g)\)

Observe that the number given for each set of clauses corresponds to a number in a gate (see Figure 3). This CNF formula has no solution, meaning that the two circuits are equivalent.
The mathematician Leonhard Euler introduced Latin squares in 1783 as a new kind of magic squares. A Latin square of order \( n \) is an \( n \times n \) array of \( n \) symbols in which every symbol occurs exactly once in each row and column of the array. Here are two examples:

Latin square of order 2
\[
\begin{array}{cc}
a & b \\
b & a \\
\end{array}
\]

Latin square of order 3
\[
\begin{array}{ccc}
x & y & z \\
z & x & y \\
y & z & x \\
\end{array}
\]

You can get many more Latin squares by permuting rows, columns, and/or symbols in any combination.

Latin squares were originally mathematical curiosities, but statistical applications were found early in the 20th century, e.g. experimental designs. The classic example is the use of a Latin square configuration to place different grain varieties in test patches. Having multiple patches for each variety helps to minimize localized soil effects.

Similar statements can be made about medical treatments. Suppose that we want to test three drugs A, B, and C for their effect in alleviating the symptoms of a chronic disease. Three patients are available for a trial, and each will be available for three weeks. Testing a single drug requires a week. Each patient is expected to try all the drugs and each drug is supposed to be tried exactly in one patient per week. The structure of the experimental units is a rectangular grid (which happens to be square in this case); there is no structure on the set of treatments. We can use the Latin square to allocate treatments. The rows of the square represent patients (P1, P2, P3) and the columns are weeks (W1, W2, W3). For example the second patient (P2), in the third week of the trial (W3), will be given drug B. Each patient receives all three drugs, and in each week all three drugs are tested.

<table>
<thead>
<tr>
<th></th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>P3</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

An incomplete or partial Latin square is a partially filled Latin square such that no symbol occurs more than once in a row or a column. The Latin square completion problem is the problem...
of determining whether the remaining entries of the array can be filled in such a way that we obtain a complete Latin square. The Latin square completion problem is NP-complete [3] and exhibits a phase-transition behavior with an easy-hard-easy pattern as a function of the fraction of the symbols already assigned [1].

Latin square completion problems are naturally represented as a constraint satisfaction problem (CSP), even though efficient SAT-based formulations have also been tried in the past [8]. SAT-based encodings for the Latin square completion problem can be distinguished between the minimal and the extended encoding. The extended encoding adds some clauses to the minimal encoding. In both SAT encodings, for each Latin square of order $n$ consider $n^3$ Boolean variables $x_{ijk}$, meaning that a symbol $k$ is assigned to cell $i,j$, where $i,j,k = 1,2,\ldots,n$.

The minimal encoding includes clauses that represent the following constraints:

1. Some symbol must be assigned to each entry:
   \[ \forall ij \forall_{k=1}^n x_{ijk} \]

2. No symbol is repeated in the same row:
   \[ \forall ijk \wedge_{l=j+1}^n (\neg x_{ijk} \lor \neg x_{ilk}) \]

3. No symbol is repeated in the same column:
   \[ \forall ijk \wedge_{l=i+1}^n (\neg x_{ijk} \lor \neg x_{ljk}) \]

The total number of clauses of the minimal encoding is $O(n^4)$.

The extended encoding explicitly considers each entry in the array has exactly one symbol, by also including the following constraints:

1. Each symbol must appear at least once in each row:
   \[ \forall ik \forall_{j=1}^n x_{ijk} \]

2. Each symbol must appear at least once in each column:
   \[ \forall jk \forall_{i=1}^n x_{ijk} \]

3. No two symbols are assigned to the same entry:
   \[ \forall ijk \wedge_{l=k+1}^n (\neg x_{ijk} \lor \neg x_{ijl}) \]

Similarly to the minimal encoding, the size of the extended encoding is $O(n^4)$.

Experimental results on both encodings reveal that SAT solvers are competitive on solving this problems, as long as the size of a problem instance is manageable. Moreover, SAT solvers perform better on the extended encoding.

6 Expected Contributions

In the previous sections we have given an introduction to SAT, mainly describing the basic backtrack search algorithm and explaining how to encode a problem in CNF. It is clear that even
though combinational circuits are the most paradigmatic example were SAT can be used, many other
problems may also be encoded in CNF, as it has been done in the past.

While SAT technology was being improved in the last decade, extended forms of satisﬁability
(also known as SAT extensions) have emerged. In SAT extensions, both variables and constraints
may be either Boolean variables and clauses, respectively, or variables with a ﬁnite set of possible
values and constraints in a speciﬁc theory.

In the last couple of years, mainly due to the increasing commercial use of SAT technology, there
has been an effort to incorporate SAT solvers in solvers dealing with extended forms of satisﬁability.
Although in the past problems encoded as SAT extensions where solved with dedicated solvers, we
are currently seeing a more extensive use of SAT solvers, by an efﬁcient conversion of extended forms
of propositional logic into SAT. Clearly, this does not mean that the whole problem will be converted
into SAT, but rather that a signiﬁcant part of the problem will be converted into SAT.

Observe that recent tools following this approach, namely MiniSat+1 and MathSAT2, are known
for being very efﬁcient and therefore demonstrate that this approach is quite promising.

In what follows we brieﬂy describe three extended forms of satisﬁability where it looks promis-
ing further integrating SAT technology: pseudo Boolean problems, satisﬁable modulo theories and
constraint satisfaction problems. By integrating SAT techniques for solving this problems, we expect
to be able to efﬁciently solve, for the ﬁrst time, a signiﬁcant number of problems.

6.1 Pseudo Boolean Problems

In Pseudo Boolean (PB) problems all variables take Boolean values, i.e. either 0 (False) or 1
(True). Constraints are linear inequalities with integer coefﬁcients, for example $2x + y + z \geq 2$.

MiniSat+ has been one of the most successful participants in the Pseudo Boolean evaluation
20053, which has been run along the SAT 2005 conference. MiniSat+ converts PB constraints to
SAT. The authors felt that this approach, as opposed to generalizations of SAT solvers to PB solvers,
had not yet been properly investigated.

6.2 Satisﬁable Modulo Theories

In Satisﬁable Modulo Theories (SMT), the goal is to decide the satisﬁability of formulas with
respect to decidable background theories.

MathSAT is a SMT solver using a combination of SAT solving and theory-speciﬁc decision pro-
duces. MathSAT is a DLL-based decision procedure for the SMT problem for various theories,
including those of Equality and Uninterpreted Function (EUF), Separation Logic (SEP), Linear
Arithmetic over the Reals (LA(R)) and Linear Arithmetic over the Integers (LA(Z)). MathSAT is

1Available at http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat+.html.
2Available at http://mathsat.itc.it.
3See http://www.cril.univ-artois.fr/PB05.
based on the approach of integrating a state-of-the-art SAT solver with a hierarchy of dedicated solvers for the different theories, and implements several optimization techniques. MathSAT pioneers a lazy and layered approach, where propositional reasoning is tightly integrated with solvers of increasing expressive power, in such a way that more expensive layers are called less frequently.

6.3 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) can be formally defined by a set of variables, each of which has a discrete and finite set of possible values (their domain), and a set of constraints among these variables. The solution to a CSP is to find a value for each variable, from their respective domain, which satisfy all the constraints.

Although SAT is usually seen as a special case of CSP, there is not much work relating both topics. We believe that the non-existence of a standard format to describe a CSP problem is a significant disadvantage. Currently, there are different languages for describing CSP problems, which are chosen depending on the solver to be used. Nonetheless, with the first CSP competition\footnote{See \url{http://cpai.ucc.ie/05/CallForSolvers.html}}, which will be held along the CP 2005 conference, the first attempt to define a standard format has been made. We plan to convert this format to CNF.

The efficiency of the conversion of extended forms of propositional logic into SAT will be measured by comparing the performance of SAT solvers with the performance of dedicated solvers, namely PB, SMT and CSP solvers, on the same problem instances. Clearly, different conversions should be tried, as well as different SAT solvers. It is well known that different SAT solvers behave differently depending on the encodings. Hence, choosing the right encoding for the right solver will be considered in our experimental evaluation.

7 Conclusions

The area of propositional satisfiability has seen remarkable improvements in the most recent years. The new solvers are capable of solving very large, very hard real-world problem instances, which more traditional SAT solvers are totally incapable of.

Recent advances in SAT solving motivate an increasing number of combinatorial problems to be encoded into SAT. In addition, one may envision converting extended forms of propositional logic into SAT. Even though many problems have been naturally encoded using these logics, we believe that converting these encodings into SAT is a promising approach.

Eventhough one should consider the additional overhead from making this conversion, we converting these problems to SAT allows to leverage recent dramatic improvement in SAT solvers. Furthermore, we automatically benefit from future progress in SAT research.
We plan to convert pseudo Boolean problems, satisfiable modulo theories and constraint satisfaction problems, among others, into SAT. We plan to try different conversions, to study its complexity and to perform an extensive experimental evaluation.

References


