RISTO HELINKO

SAT SOLVERS

CONTENTS

1. BOOLEAN SATISFIABILITY 1
2. SAT SOLVERS 2
    2.1 The DPLL algorithm 2
    2.2 Other algorithms 4
3. PRACTICAL APPLICATIONS OF SAT SOLVERS 5

REFERENCES 6

1. BOOLEAN SATISFIABILITY

A formula in propositional logic contains a sequence of variables and operators. Values of variables can be either true of false (also known as 1 or 0). For an assignment of variables the formula will yield either true or false. That assignment is called an *interpretation*, and if the value of formula becomes true by it, it is also a *model*. The question of satisfiability asks whether there exists a model for a formula. [1, p.29]

The Boolean satisfiability problem (SAT for short) was the first problem to be classified as NP-complete [2]. NP stands for nondeterministic polynomial time, which is a *time complexity class* – a set of problems classified by the time it takes to solve them. Informally speaking, a problem is in NP if a solution for it can be *verified* efficiently, that is, in polynomial time.

Class P on the other hand stands for (deterministic) polynomial time. A problem in P can be *solved* in polynomial time. P is included in NP, but equality between the two is not known [3, p.1064]. It is probably the most famous question in computer science.

As mentioned before, SAT belongs to the set of NP-complete problems, which is also a subset of NP. It consists of problems that are as hard as any in NP [3, p.1050]. Put more precisely, any problem in NP is *reducible* (in polynomial time) to an NP-complete problem [3, p.1067]. That means an NP-problem can be transformed into any of the thousands NP-complete problems.

This leads to an interesting property. If an efficient algorithm was found for any NP-complete problem, it could be used for any problem in NP. Even though there are thousands of NP-complete problems [3, p.1105] and the subject has been studied since the 1970s, such algorithm has not been found. That suggests $P \neq NP$ is the more likely scenario, and so think most researchers [3, p.1065].

So there is no efficient way of solving satisfiability, and it is unlikely that one would emerge in the future. But there is always the inefficient solution, which will be discussed next. It also turns out that it can be enhanced quite a bit, just not for all cases.

2. SAT SOLVERS

Despite a few preceding studies in closely related subjects [4, p.16], the first algorithm to solve SAT is usually attributed to Davis and Putnam in 1960 [5]. It had a tendency to cause memory explosion, so another algorithm was proposed two years later by Davis, Logemann and Loveland [6]. Its memory consumption is predictable, and therefore computing time is the only issue with it [4, p.18]. The two algorithms are referred to as DP and DPLL (or sometimes just DLL), respectively. This section will mainly discuss DPLL and its optimized versions.

2.1 The DPLL algorithm

As input the DPLL algorithm takes the propositional formula in CNF, clausal (or conjunctive) normal form. As output it gives whether the formula is SAT or UNSAT, and also gives the partial satisfying interpretation for the formula (unlike DP) [1, s.116].

The algorithm can be described by a recursive function (in pseudo-code) as follows:

```
DLL(formula, assignment)
{
      necessary = deduction(formula, assignment);
      new_asgnmnt = union(necessary, assignment);
      if (is_satisfied(formula, new_asgnmnt))
            return SATISFIABLE;
      else {
            if (is_conflicting(formula, new_asgnmnt))
                  return CONFLICT;
      }
      branching_var = choose_free_variable(formula, new_asgnmnt);
      asgn1 = union(new_asgnmnt, assign(branching_var, 1));
      result1 = DLL(formula, asgn1);
      if (result1 == SATISFIABLE)
            return SATISFIABLE;
      asgn2 = union (new_asgnmnt, assign(branching_var, 0));
      return DLL(formula, asgn2);
}
```

Program 1. *Recursive description of the DPLL algorithm [4, p.21]*

First, *deduction* adds new assignments to the interpretation, if possible. This is done by *unit propagation.* The function finds clauses that contain only one literal, called *unit clauses*. That literal is assigned the value true, which makes all clauses containing it true. Unit propagation refers to the act of repeating this until there is no more unit clauses without truth values [1, p. 115].

After unit propagation we check satisfiability, as it might be that the partial interpretation now satisfies the formula. It might also lead to a conflict, if complements of the assigned literals make a clause false.

Most likely this deduction is not enough to determine SAT. Next, a variable is chosen and a truth value is assigned to it. A recursive call is made with the interpretation that now includes a new assigned variable. If that branch leads to a conflict, the call is made with the opposite value. If both lead to a conflict, the formula is UNSAT.

Depending on the chosen variable the search will take a certain path. That is the biggest factor in the performance of DPLL – deciding which branch to pursue. For a formula to be proven SAT, one needs to find just one branch that leads to true – for the converse, it has to be shown that no branch does so. In the latter case the order of going through the branches doesn't seem to matter that much, but if the formula is SAT, one definitely should pick the branch that will most likely lead to 'true', as that would immediately terminate the algorithm.

The techniques used for the decision are *heuristics*: rules-of-thumb that may not be accurate or universal, but empirically they are shown to be useful. As the algorithm is executed by a computer, the heuristics have to rely on some measurable aspect of the formula.

The techniques can be classified by when the calculations are done. If the heuristic doesn't take to account the progress of the solver, it is said to be s*tate-independent*. Conversely, if calculations are made before each branching, the heuristic is *state-dependent.* Obviously the latter leads to a bigger demand on computing power, but they have been shown to be more effective, especially in real world applications [4, p. 34].

The algorithm above maintains the same formula throughout the solving process. This, however, is not the case in most state-of-the-art solvers. They have a feature called *learning*, which adds clauses to the formula as calculation progresses. It relies on multiple ways of deducing from the calculation whether a certain branch will definitely lead to a conflict. It is important as it reduces the search space, and therefore computing time, whether the formula is SAT or not.

As mentioned before, the heuristics are not universal, which makes comparison between them complicated. Choosing the "best" one depends on the formula one tries to solve. Different heuristics and learning schemes are introduced for example in Zhang's thesis [4].

2.2 Other algorithms

Even though most of modern solvers are based on DPLL, they are not the only ones available. Other procedures include for example Binary Decision Diagrams (BDD), Stålmarck's algorithm and stochastic algorithms.

Stålmarck's algorithm dates back to early 1980s [4, p.24]. It is a proprietary solver, which is partly why it is not that widely used and researched. The algorithm is actually designed to solve validity of a formula, but it can also be used for SAT solving.

Whereas DPLL takes in the formula in CNF, Stålmarck uses triplets (p, q, r):

$$p \leftrightarrow (q \rightarrow r)$$

where p, q and r are literals. Any function in logic can be represented with such triplet [4, p.25]. The algorithm uses two kinds of rules to simplify the formula: *simple rules* and the *dilemma rule*. The former includes statements like "(0, y, z) means y=1 and z=0." The simple rules alone cannot determine the satisfiability of the formula. Dilemma rule won't be discussed here in detail, but it is described for example in Zhang's thesis [4, p. 26].

Stålmarck's algorithm determines SAT by deriving *terminal triplets*. There are three of them: (1, 1, 0), (0, x, 1) and (0, 0, x). An appearance of such triplet means there is a contradiction in the logic – if the formula was assumed to be false, a contradiction makes it valid, and vice versa.

DP, DPLL and Stålmarck's algorithm are all *complete*, which means given enough time and memory, they will always come up with an answer. *Stochastic (or randomized) algorithms*, on the other hand, are *incomplete* [4, p.27]. They can only find a model, that is, they cannot prove a formula is UNSAT.

That is certainly a drawback in some cases, but not all. Generally speaking a randomized algorithm is useful when most possible answers are acceptable, and only a handful make the solving highly inefficient. So to avoid systematically picking the worst candidates over and over again, the algorithm does not pick systematically, but in random. So if most interpretations satisfy the formula, a randomized approach might be the right choice.

As the stochastic algorithm doesn't rely much on the characteristics of the formula, there is a lot of general research results that can be applied here. Again, examples are discussed in Zhang [4].

3. PRACTICAL APPLICATIONS OF SAT SOLVERS

As mentioned in the introduction, SAT is NP-complete, and therefore any problem in NP can be efficiently transformed to a SAT problem. That means a good SAT solver would be useful to solve any problem in NP. The performance, however, is determined largely by the instance. That suggests that it may not be worth the extra computation to transform the problem, unless the propositional formulation of the problem is known to be easily solved.

Even if SAT solvers are not the go-to tool for all NP problems, they are useful in several fields. As solvers become more efficient in solving a wide range of instances, they are of course becoming more and more useful.

One of the original application areas was artificial intelligence, specifically relating to space exploration [7]. SAT solvers have also been used for quite a while in EDA (Electronic Design Automation), in multiple processes such as automatic test pattern generators and equivalence checkers. More recently they have been applied to software verification and debugging, too.

REFERENCES

[1] M. Ben-Ari, Mathematical Logic for Computer Science, 3rd Edition, Springer-Verlag London, UK, 2009, 346 p.

[2] S. Cook, The Complexity of Theorem-proving Procedures. Originally published in Proceedings of the Third IEEE Symposium on the Foundations of Computer Science, 1971. Available: http://4mhz.de/cook.html

[3] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, The MIT Press, Cambridge, Massachusetts, USA, 2009, 1292 p.

[4] L. Zhang, Searching for Truth: Techniques for Satisfiability of Boolean Formulas, dissertation, Princeton University, 2003, 197 p. Available: http://research.microsoft.com/en-us/people/lintaoz/thesis_lintao_zhang.pdf

[5] M. Davis, H. Putnam, A Computing Procedure for Quantification Theory. Originally published in Journal of the ACM, Vol. 7, Iss. 3, pp.201 – 215, 1960. Available: http://dl.acm.org/citation.cfm?id=321034

[6] M. Davis, G. Logemann, D. Loveland, A Machine Program for Theorem Proving. Originally published in Communications of the ACM, Vol. 5, Iss. 7, pp.394–397, 1962. Available: http://dl.acm.org/citation.cfm?id=368557

[7] S. Malik, L. Zhang, Boolean Satisfiability: From Theoretical Hardness to Practical Success, Communications of the ACM, Vol. 52, Iss. 8. pp. 76-82, 2009. Available: http://cacm.acm.org/magazines/2009/8/34498-boolean-satisfiability-from-theoretical-hardness-to-practical-success/fulltext