Laboratory: *SAT Solvers*[*]

Objectives

1. Using a SAT solver for deciding the SAT of various problems

1 Preliminaries

A **SAT solver** solves the *Boolean satisfiabiliy problem.*

The SAT problem can be determined using truth table algorithm:

1. *Case 1*: To check whether F is satisfiable, compute the truth table for F. If there is a row in which $\mathbb{T}$ appears as the value for F, then F is satisfiable. Otherwise, F is unsatisfiable.

2. *Case 2*: To check whether $F_1, ..., F_n \models G$ [1], check the satisfiability of $F_1 \wedge ... \wedge F_n \wedge \neg G$. If it is unsatisfiable, then $F_1, ..., F_n \models G$, otherwise $F_1, ..., F_n \not\models G$.

What is the complexity of such an algorithm? Can we do better than 2^n (n is the number of propositional symbols)?

SAT was the first problem shown to be NP-complete [Coo71]: all of the problems in the class NP can be solved by translating them (in polynomial time) into SAT. Hence, if we could somehow build a fast solver for SAT, it could be used to solve lots of other problems. In theory, this seems dubious, as problems in NP are known to take exponential time in the worst case. Remarkably, modern SAT solvers are very fast most of the time!

A formula must be in conjunctive normal form (CNF) in order to be handled by a SAT solver.

Example 1. *Consider the following formula which is in CNF:*

$$
\begin{aligned}
&(\neg A \vee \neg B \vee E) \wedge (\neg E \vee A) \wedge (\neg E \vee B) \wedge \\
&(\neg C \vee F) \wedge (\neg F \vee C) \wedge \\
&(\neg D \vee \neg E \vee G) \wedge (\neg G \vee D) \wedge (\neg G \vee E) \wedge \\
&(\neg E \vee \neg F \vee H) \wedge (\neg H \vee E) \wedge (\neg H \vee F) \wedge \\
&(G \vee H \vee \neg I) \wedge (\neg H \vee E) \wedge (\neg H \vee F) \wedge \\
&I
\end{aligned}
$$

[*]Based on: (Lecture notes SAT: Theory and Practice by Clark Barrett from VSTA 2008 Summer School)

[1] $A \models B$ means that for all $I \in \mathcal{I}$, if A is true in I, then also B is true in I

Alternative notation is as follows:

1. Alternative 1:

   Example 2. *For example above, we have:*

   $$(A' + B' + E)\,(E' + A)\,(E' + B)$$
   $$(C' + F)(F' + C)$$
   $$(D' + E' + G)(G' + D)(G' + E)$$
   $$(E' + F' + H)(H' + E)(H' + F)$$
   $$(G + H + I')(H' + E)(H' + F)$$
   $$I$$

2. Alternative 2: DIMACS standard: Each variable is represented by a positive integer. A negative integer refers to the negation of the variable. Clauses are given as sequences of integers separated by spaces. A 0 (zero) terminates the clause.

   Example 3. *For example above, we have:*

   $$-1\ -2\ 5\ 0 \quad -5\ 1\ 0 \quad -5\ 2\ 0$$
   $$-3\ 6\ 0 \quad -6\ 3\ 0$$
   $$-4\ -5\ 7\ 0 \quad -7\ 4\ 0 \quad -7\ 5\ 0$$
   $$-5\ -6\ 8\ 0 \quad -8\ 5\ 0 \quad -8\ 6\ 0$$
   $$7\ 8\ -9\ 0 \quad -8\ 5\ 0 \quad -8\ 6\ 0$$
   $$9\ 0$$

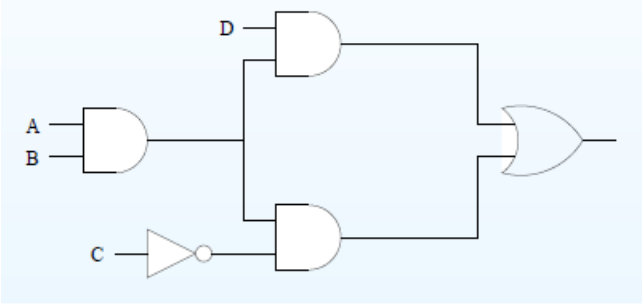
The first SAT solvers were based on the Davis-Putnam method (see lectures). Nowadays, SAT solvers, which are extremely fast being able to solve formulae with more than $1000K$ clauses and tens of thousands of variables, are based on the (1) Conflict-Driven Clause Learning algorithm, which can be viewed as a modern variant of the DPLL algorithm (e.g. the SAT solver Chaff) and (2) stochastic local search algorithms (e. g. SAT solver WalkSAT).

We will use an online SAT solver, e.g. `http://dai.fmph.uniba.sk/~simko/satsolver/`, `https://www.comp.nus.edu.sg/~gregory/sat/`, `https://www.msoos.org/2013/09/minisat-in-your-browser/`. Experimenting with more advanced SAT solvers (see `http://www.satcompetition.org/` for the most competitive SAT solvers) is strongly encouraged.

SAT solvers could help in solving the long-standing problem in theoretical computer science P = NP? More details here:
`http://www.se-radio.net/2017/07/se-radio-episode-298-moshe-vardi-on-p-versus-np/`.

2 Exercises

1. The *circuit satisfiability problem* (CSP) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true. Consider the CSP for following circuit using a SAT solver:



2. Consider Homework 1, in particular the exercises asking you to decide the consistency (SAT) of formulae and logical consequences. Check your initial answer using a SAT solver. You should have at least 5 examples, 3 checking for consistency (SAT) and 2 for logical consequence.

3. Search on the internet relevant problems with significantly more variables (at least 100) and clauses (at least 1000) and use a SAT solver to check for their SAT/UNSAT.

References

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.