Boolean Satisfiability
Lecture 1: Introduction

Edward A. Hirsch*

January 1, 2024

Welcome!

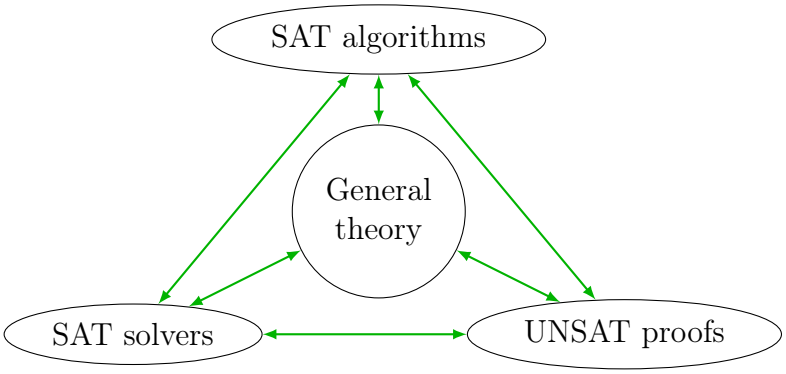
Welcome to the course! For the course logistics and workflow, see the slides from the first lecture.

Contents

*Ariel University, http://edwardahirsch.github.io/edwardahirsch

1 Introduction

In this course we will be interested in various aspects of **SAT**, the Boolean satisfiability problem, which is the problem of finding solutions to propositional (or Boolean) formulas. This problem is not just central to the complexity theory, it is extremely important for practical applications, as many computational problems are very naturally reduced to **SAT**. Our focus will be on the topics described in the following picture:



2 Background

This section is intended to remind definitions and facts learned in earlier courses, and to unify the notation.

2.1 Terminology and Notation

2.1.1 Formulas

- Boolean (propositional) **variables** $x_1, x_2, \ldots, x_n$ can take (truth) values `True` or `False` (aka 1 or 0). We will typically use n to denote the number of variables.
- **Literals** are variables x_i ("positive literals") or the negations of variables $\overline{x_i}$ ("negative literals").
- **Clauses** are sets of literals interpreted as disjunctions $\ell_1 \vee \ldots \vee \ell_k$. Normally, we assume that x and $\overline{x}$ do not occur together.
- A formula in conjunctive normal form (**CNF**) is a conjunction of clauses (maxterms): $F = C_1 \wedge \ldots \wedge C_m$ for clauses $C_1, \ldots, C_m$, for example,

$$(x \vee y) \wedge (\overline{x} \vee \overline{y}) \wedge (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee y).$$

- A formula in disjunctive normal form (DNF) is a disjunction of conjunctions (minterms), for example,

$$(\overline{x} \wedge \overline{y}) \vee (x \wedge y) \vee (\overline{x} \wedge y \wedge \overline{z}) \vee (x \wedge \overline{y}).$$

If we negate a DNF tautology, we obtain an unsatisfiable CNF formula. We will frequently call such two formulas by the same name (for example, the propositional pigeonhole principle).

2.1.2 Truth Assignments

- A (complete) (truth) **assignment** maps all variables to truth values.
- A **partial assignment** maps some variables to their values.

We will denote assignments in two ways:

$$\text{assignment } x \leftarrow \texttt{False},\ y \leftarrow \texttt{True},\ z \leftarrow \texttt{True}$$
$$\text{assignment } \{\overline{x}, y, z\}$$

Let F be a formula in CNF, and A be a (partial) assignment.

- **Substitution** is the application of A to F, its result is denoted $F[A]$:
  - we remove all the satisfied clauses,
  - we remove false literals from the other clauses.

For example,
$$\begin{aligned} F &= (x \vee y) \wedge (\overline{x} \vee \overline{y}) \wedge (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee y), \\ F[x \leftarrow \texttt{True}] &= (\cancel{x} \vee \cancel{y}) \wedge (\cancel{\overline{x}} \vee \overline{y}) \wedge (\overline{y} \vee z) \wedge (\cancel{\overline{x}} \vee y). \end{aligned}$$

- A **satisfying assignment** A satisfies all clauses, that is, $F[A] = \emptyset$ $(= \texttt{True})$.
- If a formula has at least one satisfying assignment, it is **satisfiable**; otherwise it is **unsatisfiable**.

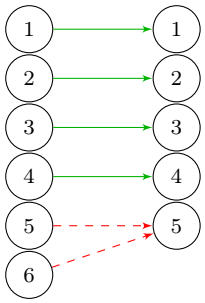
2.2 PHP: The propositional pigeonhole principle

The propositional pigeonhole principle (**PHP**) plays an important role as an example of a "hard" formula: it provides both benchmarks for SAT solvers and candidate formulas for exponential lower bounds on the size of the smallest refutation.

We all know the pigeonhole principle as a basic mathematical fact usually formulated as the absence of an injective mapping of a set of larger cardinality to a set of smaller cardinality.

The propositional formulation of (the negation of) PHP for a specific number of pigeons and holes is an important example of an unsatisfiable formula in CNF that is hard for many popular SAT solvers (and proof systems):

- $x_{i,j} \sim$ pigeon i sits in hole j
  $(1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant m-1)$,
- $\overline{x_{i,j}} \vee \overline{x_{i',j}}$ (for each $i, j, i' \neq i$):
  two pigeons i, i' do not share the same hole j,
- $x_{i,1} \vee \ldots \vee x_{i,m-1}$ (for each i):
  pigeon i is assigned to some hole.



The (unsatisfiable) negation of PHP is in CNF. The corresponding tautology is in DNF.

2.3 Complexity remarks

2.3.1 Search or Decision?

NP problems come in two versions:

SAT, decision version: Given a formula, **check** its satisfiability (return `True`/`False`).

SAT, search version: Given a formula, **find** a satisfying assignment (or say "no").

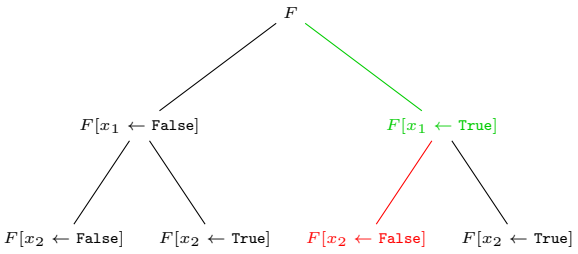
The following procedure shows that for **NP**-complete problems they are polynomially equivalent.

```
Search-to-decision(F):
  if (F = ∅) then print("yes")
  elif (∅ ∈ F) then print("no")
  elif (SAT(F[x ← 0]))                          // x is the first variable
    then { print(x̄); Search-to-decision(F[x ← 0]); }
    else { print(x); Search-to-decision(F[x ← 1]); }
```

Why does it run in a polynomial time?

We always dive in a subtree where a satisfying assignment hides without examining another subtree.

So we only have to process a single path! (See the picture.)



2.3.2 Complexity properties of SAT

- **SAT** is in **NP** as it is easy to verify candidate solutions.
- **SAT** is **NP**-complete (Cook–Levin Theorem), that is, other problems in **NP** reduce to it in polynomial time.
- Moreover, they reduce in a natural way, which is very useful.

- Even some restricted versions of **SAT** are hard:

  k-**CNF**: every clause contains at most k literals; the problem: ***k*-SAT**.
  A clause containing k literals: k-**clause**.

  Even **3-SAT** is already **NP**-complete.

- Sometimes **SAT** is called "**CNF SAT**". One can consider non-CNF SAT (arbitrary formulas using Boolean operations).

Remark 1. **UNSAT** (where one needs to answer 1 if the input formula in CNF is *un*satisfiable) is **co-NP**-complete, that it, it is in **co-NP** and every problem in **co-NP**[1] is polynomial-time many-one (Karp) reducible to it. The same happens for **TAUT**, the problem that asks whether a Boolean formula in DNF is a tautology (think why!).

Exercise 1 (Warmup). Reduce **3-SAT** to its instances containing at most 3 occurrences of every variable.

3 Schaefer's Dichotomy Theorem

Can one understand that a formula is easy from its syntax? Sometimes yes.

In this section we consider several classes of formulas where we can check the satisfiability in a polynomial time. We give polynomial-time algorithms for them and then formulate the result indicating that these are essentially all such classes defined in terms of constraint satisfaction problems.

3.1 Polynomial-time tractable classes of formulas

Horn formulas. This is our first polynomial-time tractable class.

> **Definition 1 (Horn formulas, Alfred Horn).** A formula in CNF is called **Horn** if every its clause contains at most one positive literal.
>
> A clause containing a single literal is called **unit** clause. The value of such literal is, of course, evident from the clause.
>
> Notice that if a Horn formula contains no unit (and no empty) clauses, it is satisfiable by the all-0 assignment.
>
> The following simple procedure that eliminates unit clauses (it is also called **unit propagation**) is in fact a very important one not just for **Horn-SAT**. In particular, in practice SAT solvers spend most of their time doing it, so optimizing this procedure is crucial for the success of a SAT solver.

[1] For a complexity class $\mathcal{C}$, the class **co**$-\mathcal{C}$ contains every language (decision problem) L such that its complement $\overline{L} \in \mathcal{C}$. In particular, languages in **co-NP** are those where we can verify solutions for the answer 0. It is a major open problem whether **NP** = **co-NP** (it is implied by, but not necessarily equivalent to, **P** = **NP**).

`Unit-clause-elimination(`F`):`
    `while (there is` $(\ell) \in F$`) do` $F := F[\ell]$.

This procedure

- does not change the satisfiability,
- the formula remains Horn.

To solve **SAT** for a Horn formula F, apply `Unit-clause-elimination` to F.

If afterwards F contains the empty clause, then it is, of course, unsatisfiable. Otherwise it is satisfiable.

Example 1. Here is how `Unit-clause-elimination` works on a Horn formula:

$$\begin{array}{llll} (x) & (y \vee \overline{z}) & (\overline{x} \vee \overline{y}) & (\overline{x} \vee z) \\ \boxed{(x)} & (y \vee \overline{z}) & (\overline{x} \vee \overline{y}) & (\overline{x} \vee z) \\ & (y \vee \overline{z}) & (\overline{y}) & (z) \\ & (y \vee \overline{z}) & (\overline{y}) & \boxed{(z)} \\ & (y) & (\overline{y}) & \\ & \boxed{(y)} & (\overline{y}) & \\ & & () & \end{array}$$

Anti-Horn formulas is our second "easy" class (dual to Horn formulas).

Definition 2. An **anti-Horn** (or **dual Horn**) formula contains at most one negative literal per clause.

Systems of linear (affine) equations modulo two. These are xor-relations of the form $x_{i_1} \oplus \ldots \oplus x_{i_k} = b$, where constant $b \in \{0, 1\}$.

Example 2.

$$\begin{aligned} x_1 \oplus x_2 \oplus x_3 &= 1 \\ x_2 \oplus x_4 \oplus x_5 &= 0 \\ x_1 \oplus x_4 &= 0 \\ x_1 \oplus x_5 &= 1 \end{aligned}$$

As all linear systems, systems modulo two can be solved using the Gaussian elimination algorithm.

Note that every such equation in k variables is expressed by 2^{k-1} clauses, and this representation can be easily recognized. For example, for $k = 2$ this is

$$x \oplus y \qquad \sim \qquad (x \vee y) \wedge (\overline{x} \vee \overline{y})$$

2-SAT. A reminder from the Algorithms-2 course.

We transform a 2-CNF formula into a directed graph with $2n$ vertices:

- We introduce a vertice for every literal.
- A clause $\ell \vee \ell'$ is mapped to the two edges $\overline{\ell} \to \ell'$ and $\overline{\ell'} \to \ell$.

Note that the formula is unsatifiable iff for some variable x, there is a directed cycle containing both x and $\overline{x}$.

Algorithm: Construct strongly connected components and check this.

Warning: the optimization problem **MAX-SAT** asking for an assignment satisfying the maximum possible number of clauses, remains **NP**-hard even for 2-CNFs.

3.2 Constraint Satisfaction Problem

We now put our polynomially tractable classes in the context of the Constraint Satisfaction Problem. In fact, this problem is an umbrella name for many computational problems formulated using a pre-defined set of relations.

Definition 3 (CSP). A **Constraint Satisfaction Problem** (**CSP**) is parameterized by a finite domain $D = [1\,..\,n]$ and a (not necessarily finite) set of relations $R_i : D^{a_i} \to \{0, 1\}$.

Input: Constraints $C_1, \ldots, C_m$ of the form $C_j = (r_j, s_j)$, where

- r_j is one of the R_i's,
- $d_j = a_i$,
- s_j is a d_j-size tuple of D-valued variables $\{x_1, \ldots, x_n\}$.

Output: a satisfying assignment $f\colon [1\,..\,n] \to D$ such that for every j,

$$r_j(f(s_{j,1}), \ldots, f(s_{j,d_j})) = \texttt{True}$$

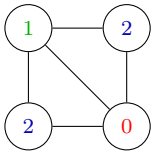
or "no" if it does not exist.

Remark 2. One can consider different domains for different variables. Moreover, one can consider even infinite domains, but it is another story.

CSP thus asks for a satisfying assignment to a conjunction of relation(s)-based conditions. It gives a way to define various computational problems. We consider two examples:

	SAT	**3-COLORING**	**CSP**
domains	{0,1}	{0,1,2}	any
relations	disjunctions of literals	non-equality	some relations

Example 3 (**3-COLORING**). This is a problem asking for vertex coloring of a graph in 3 colors such that adjacent vertices have different colors.



Domain: Three colors $\{0, 1, 2\}$.

Relation: $R(x, y) = (x \neq y)$.

Input: $\{(R, \{x_u, x_v\}) : \{u, v\} \in E\}$ for a graph $G = (V, E)$ and variables x_v for $v \in V$.

Output: $f : V \to \{0, 1, 2\}$ s.t. $f(x) \neq f(y)$ for every edge $\{u, v\} \in E$.

Example of a formula: $R(x_a, x_b) \wedge R(x_b, x_c) \wedge R(x_c, x_d) \wedge R(x_d, x_a) \wedge R(x_a, x_c)$.

Example 4 (**2-SAT**).

Domain: Boolean $\{\texttt{False}, \texttt{True}\}$.

Relations:

$$\begin{aligned} R_1(x, y) &= x \vee y & \qquad R_5(x) &= x \\ R_2(x, y) &= x \vee \overline{y} & \qquad R_6(x) &= \overline{x} \\ R_3(x, y) &= \overline{x} \vee y \\ R_4(x, y) &= \overline{x} \vee \overline{y} \end{aligned}$$

Example of a formula: $R_1(x, y) \wedge R_2(x, z) \wedge R_6(x)$.

3.3 Schaefer's dichotomy theorem

As we know, CSP over $\{0, 1\}$ for clauses (disjunctions where some arguments are negated) is called **SAT**. Schaefer's theorem tells us that the following types of Boolean CSPs are the only types that admit polynomial-time **SAT** algorithms unless **P** = **NP**.

1. CSP for 2-clauses and 1-clauses (and their conjunctions), that is, **2-SAT**.
2. CSP for xor-relations (and their conjunctions), that is, **XOR-SAT**.
3. CSP for Horn clauses (and their conjunctions), that is, **Horn-SAT**.
4. CSP for anti-Horn clauses (and their conjunctions), that is, **anti-Horn-SAT**.
5. CSP for all relations R such that $R(1, \ldots, 1) = 1$ (called 1-valid).
6. CSP for all relations R such that $R(0, \ldots, 0) = 1$ (called 0-valid).

Theorem 1 (Schaefer, 1978)**.** *Consider a set of relations $\mathcal{R}$ over the domain $\{0, 1\}$. If all of them belong to the same single type out of (1)–(6), plus trivially false and trivially true relations, then **CSP** for $\mathcal{R}$ is polynomial-time solvable. Otherwise it is **NP**-hard.*

We have already proved the first part of this theorem, and we will not prove the **NP**-hardness part in this course.

Remark 3. While **2-SAT**, **(Anti)Horn**, **XOR-SAT** are polynomial-time solvable, any mix of them is not necessarily that easy!

4 Boolean circuits

Boolean circuits are an abstraction for computations that have fixed-length inputs. Many good examples of such computations can be found in the circuitry of a CPU, for example, 64-bit arithmetic operations.

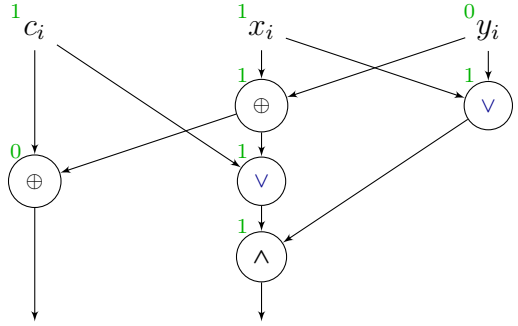
The verification of such computations naturally leads to checking the satisfiability of Boolean formulas, thus it is a popular source of benchmarks for SAT solvers.

4.1 The basics

Definition 4. A **Boolean circuit** is a directed acyclic graph. Its sources (nodes of in-degree zero) are called **inputs** and are labelled by Boolean variables. Its internal nodes (called **gates**) are labelled by binary[2] Boolean operations. Some of the nodes are designated as **outputs**.

The computation of a circuit is performed in a natural way: the inputs get values, then one computes the values of the gates that already have values on the incoming edges, etc.

Example 5 (Full Adder). The following circuit is a building block for integer addition. (Green numbers show the values propagating from the top to the bottom.)

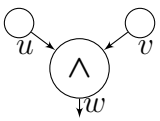


Circuit-SAT is a generalization of **SAT**, and it is obviously in **NP**, so it is, of course, polynomially equivalent to **SAT**.

The reduction is very natural. To reduce the satisfiability problem for circuits (are there input values such that the circuit evaluates to 1) to **3-SAT** one can use **Tseitin's translation**.

The resulting 3-CNFs will have both old variables (present in the circuit itself) and new auxiliary variables, one for every internal gate.

Every auxiliary variable comes with a bunch of clauses expressing that its value is computed correctly. For example, if a gate is labelled with $\wedge$, one adds the following three clauses:



$$(\overline{w} \vee u) \quad \wedge$$
$$(\overline{w} \vee v) \quad \wedge$$
$$(\overline{u} \vee \overline{v} \vee w)$$

[2]The negations are usually embedded to the adjacent nodes.

Exercise 2 (***k*-SAT→ 3-SAT**). Reduce ***k*-SAT** to **3-SAT** introducing fewer new variables than in Tseitin's transformation. Then show how to restore the satisfying assignment.

Remark 4. One popular way to prove the Cook–Levin theorem is to replace a non-deterministic polynomial-time Turing machine by a Boolean circuit, thus showing the **NP**-completeness of **Circuit-SAT**. The inputs of the constructed circuit are the bits of the witness (solution), and the gates are arranged in levels, level i describing the full configuration of the machine at step i. The number of steps is defined by the polynomial that bounds the running time of the machine applied to the actual length of its input. Level 0 describes the starting configuration formed using the input of the machine (constants hardwired to the circuit), the witness, and other necessary attributes of the machine. Levels i and $i+1$ are connected using subcircuits performing the correct computation of the next configuration. The output checks whether the machine comes to an accepting configuration.

4.2 Equivalence checking

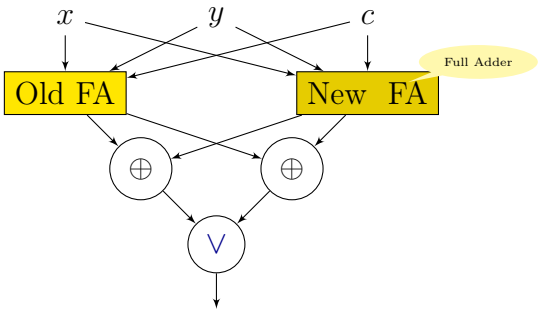
Equivalence checking is a major problem in circuit design and verification. It is a very important application of **SAT** and a very popular source of benchmarks for SAT solvers.

Equivalence checking is asking whether two Boolean circuits compute the same function. Imagine that you have two versions of some circuit, say, 64-bit integer multiplication: a good old one and a new experimental highly optimized one. How can you make sure that the new circuit does the same thing as the old one?

Definition 5 (Equivalence checking). For two circuits f and g, check whether there are $x_1, \ldots, x_n \in \{0,1\}$ such that $f(x_1 \cdots x_n) \neq g(x_1 \cdots x_n)$.

This problem can be expressed as the satisfiability of a circuit comparing the outputs of f and g: just merge the two circuits sharing the same inputs and connect their outputs by $\vee$ of $\oplus$'s.

Example 6.



Remark 5. If we find a satisfying assignment for this circuit, it means that we have found a bug in the newly suggested hardware.

However, what if a bug is not found? Even if our algorithm is deterministic and worked till it outputted "UNSAT", can we be absolutely confident that the circuits are indeed equivalent? An error can cost billions of dollars, thus companies strongly prefer to have *provably* correct hardware. We will later talk about *proof systems* that can such provide such proofs.

5 Practical SAT solving: An overview

5.1 Solvers and Benchmarks

A **SAT solver** is simply a program that solves **SAT**. While **SAT** is **NP**-complete, we still need to solve it (well, some of it cases) in practice.

How is it possible? SAT solvers use highly heuristic algorithms, and of course they are very slow on some inputs — hopefully, not on those that we will need when we use the solver.

Solvers regularly participate in SAT Competition[3]: a regular race of SAT solvers. During a competition, the solvers are run on sets of benchmarks and are compared on the basis of how many benchmarks they solve within specified time slice (see https://satcompetition.github.io/2023/downloads/satcomp23slides.pdf for an example).

What is a **benchmark**? It is simply a Boolean formula used to compare the performance of SAT solvers on it. SAT Competitions maintain a database of benchmarks:

- industrial (equivalence checking, and more),
- handcrafted (known to be hard: for example, PHP),
- randomly generated.

"Industrial" benchmarks are frequently "easy" (for example, contain lots of 2-clauses). Thus data structures and efficient implementation are very important.

Remark 6. Nowadays SAT solvers are used not only in the industrial applications. Even mathematicians use them to check their combinatorial conjectures. It takes only a few minutes to write down a CNF and run a SAT solver on it!

There are many solvers developed both in the academia and in the industry (an example in Israel is Intel® SAT Solver developed in Haifa by Alexander Nadel[4]).

5.2 Classification of SAT Solvers

Solvers can be classified by several features.

1. Completeness.
   - Complete:
     - outputs a satisfying assignment, if it finds one,
     - says "UNSAT" if there is none,
     - may produce a proof of unsatisfiability.
   - Incomplete:
     - can also return "UNKNOWN".

[3] www.satcompetition.org
[4] drops.dagstuhl.de/opus/volltexte/2022/16682/pdf/LIPIcs-SAT-2022-8.pdf

2. Randomness.
   - Randomized:
     - Errorless (just using randomness to tune the heuristics).
     - Bounded-error (sometimes also called incomplete...).
   - Deterministic.
3. The ability to use multiple processors / threads.
   - Parallel.
   - Sequential.

5.3 Competition Formats: Input and Output

Competition formats using to process CNFs and to output satisfying assignments are self-explanatory.

5.3.1 Input.

- Benchmarks.
  - (Simplified) DIMACS format:
    $(x_1 \vee \overline{x}_5 \vee x_4) \wedge (\overline{x}_1 \vee x_5 \vee x_3 \vee x_4) \wedge (\overline{x}_3 \vee \overline{x}_4)$ is described as

```
c
c start with comments
c here is a CNF with 5 variables and 3 clauses
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

  - RTL format
    - Serves for describing Boolean circuits

5.3.2 Output.

- Decisions and Assignments:
  the positive answer with sat. assignment $x_1, x_3, \overline{x}_4, x_6$ is described as

```
c Comments are still allowed
s SATISFIABLE
v 1 3
v -4 6 0
```

- Proofs of unsatisfiability (if required)
  - DRAT format (too early to explain).
- Format descriptions and other rules:
  https://satcompetition.github.io/2023/

Further reading

As this lecture is introductory, there is no single source for the material. One can check any edition of SAT Handbook (*Handbook of Satisfiability*, edited by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh; published by IOS Press), in particular the exposition of Schaefer's theorem follows the chapter "Worst-Case Upper Bounds". One can also check the SAT Competition web site out of curiosity. However, in fact the subject is covered sufficiently in these lecture notes.