CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Lecture 14: Logic

Harvard SEAS - Fall 2022 *2022-10-25*

1 Announcements

Recommended Reading:

- Lewis–Zax Ch. 9–10.
- Roughgarden IV, Sec. 21.5, Ch. 24.

2 Propositional Logic

Motivation: Logic is a fundamental building block of computation (e.g. digital circuits), and is also a very expressive language for encoding computational problems we want to solve.

Definition 2.1 (informal)**.** A *boolean formula* φ is a formula built up from a finite set of variables, say $x_0, \ldots, x_{n-1}$, using the logical operators $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT), and parentheses.

Every boolean formula φ on n variables defines a boolean function, which we'll abuse notation and also denote by $\varphi : \{0,1\}^n \to \{0,1\}$, where we interpret 0 as false and 1 as true, and give $\wedge, \vee, \neg$ their usual semantics (meaning).

See the Lewis–Zax text for formal, inductive definitions of boolean formulas and the corresponding boolean functions.

Examples:

$$\varphi_{maj}(x_0, x_1, x_2) = (x_0 \wedge x_1) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_0)$$

is a boolean formula. It evaluates to 1 if

$$\varphi_{pal}(x_0, x_1, x_2, x_3) = ((x_0 \wedge x_3) \vee (\neg x_0 \wedge \neg x_3)) \wedge ((x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2))$$

is a boolean formula. It evaluates to 1 if

Definition 2.2. A *literal* is a variable (e.g. x_i) or its negation ($\neg x_i$).

A boolean formula is in *disjunctive normal form (DNF)* if it is the OR of a sequence of *terms*, each of which is the AND of a sequence of literals.

A boolean formula is in *conjunctive normal form (CNF)* if it is the AND of a sequence of *clauses*, each of which is the OR of a sequence of literals.

Note terms and clauses may contain duplicate literals, but if a term or clause contains multiple copies of a variable x, it's equivalent to a term or clause with just one copy (since $x \vee x = x$ and $x \wedge x = x$). We can also remove any clause or term with both a variable x and its negation $\neg x$, as that clause or term will be always true (in the case of a clause) or always false (in the case of a term). We define a function Simplify which takes a clause and performs those simplifications: that is, given a clause B, Simplify(B) removes duplicates of literals from clause B, and returns 1 if B contains both a literal and its negation. Also, if we have an order on variables (e.g. x_0, x_1, ...), Simplify(B) also sorts the literals in order of their variables.

Also by convention, an empty term is always true and an empty clause is always false.

Lemma 2.3. *For every boolean function $f : \{0,1\}^n \to \{0,1\}$, there are boolean formulas φ and ψ in DNF and CNF, respectively, such that $f \equiv \varphi$ and $f \equiv \psi$, where we use $\equiv$ to indicate equivalence as functions, i.e. $f \equiv g$ iff $\forall x : f(x) = g(x)$.*

Proof. □

3 Computational Problems in Propositional Logic

Input	**:** A boolean formula φ on n variables
Output	**:** An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem Satisfiability

It is common to restrict the boolean formulas to ones in CNF or DNF.

Input	**:** A CNF formula φ on n variables
Output	**:** An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem CNF-Satisfiability

Input	**:** A DNF formula φ on n variables
Output	**:** An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem DNF-Satisfiability

One of these problems is algorithmically very easy. Which one?

4 Modelling using Satisfiability

One of the reasons for the importance of Satisfiability is its richness for encoding other problems. Thus any effort gone into optimizing algorithms for Satisfiability (aka "SAT Solvers") can be easily be applied to other problems we want to solve.

Theorem 4.1. *Graph k-Coloring on graphs with n nodes and m edges can be reduced in time $O(n + km)$ to CNF-Satisfiability on boolean formulas with kn variables and $n + km$ clauses.*

Proof.

□

Unfortunately, the fastest known algorithms for CNF-Satisfiability have worst-case runtime exponential in n, However, enormous effort has gone into designing heuristics that complete much more quickly on many real-world instances. In particular, SAT Solvers—with many additional optimizations—were used to solve large-scale graph coloring problems arising in the 2016 US Federal Communications Commission (FCC) auction to reallocate wireless spectrum. Roughly, those instances had $k = 23$ (corresponding to UHF channels 14–36), n in the thousands (corresponding to television stations being reassigned to one of the k channels), m in the tens of thousands (corresponding to pairs of stations with overlapping broadcast areas — similarly to how you are viewing interval scheduling on ps7). Over the course of the one-year auction, tens of thousands of coloring instances were produced, and roughly 99% of them were solved within a minute!

Thus motivated, we will now turn to algorithms for Satisfiability, to get a taste of some of the ideas that go into SAT Solvers.

5 Resolution (Preview)

SAT Solvers are algorithms to solve CNF-Satisfiability. Although they have worst-case exponential running time, on many "real-world" instances, they terminate more quickly with either (a) a satisfying assignment, or (b) a "proof" that the input formula is unsatisfiable.

The form of these unsatisfiability proofs is (implicitly) based on *resolution*. The idea is to repeatedly derive new clauses from the original clauses (using a valid deduction rule) until we either derive an empty clause (which is false, and a proof that the original formula is unsatisfiable) or we cannot derive any more clauses.

Definition 5.1 (resolution rule)**.** For clauses C and D, define

$$C \diamond D = \begin{cases} \text{Simplify}((C - \{\ell\}) \vee (D - \{\neg\ell\})) & \text{if } \ell \text{ is a literal s.t. } \ell \in C \text{ and } \neg\ell \in D \\ 1 & \text{if there is no such literal } \ell \end{cases}$$

Here $C - \{\ell\}$ means remove literal ℓ from clause C, 1 represents `true`. As noted last time, if C and D can be resolved with respect to more than one literal ℓ, then for all choices of ℓ we will have $\text{Simplify}((C - \{\ell\}) \vee (D - \{\neg\ell\})) = 1$, so $C \diamond D$ is well-defined.
Examples:

From now on, it will be useful to view a CNF formula as just a set $\mathcal{C}$ of clauses.

Definition 5.2. Let $\mathcal{C}$ be a set of clauses over variables $x_0, \ldots, x_{n-1}$. We say that an assignment $\alpha \in \{0,1\}^n$ *satisfies* $\mathcal{C}$ if α satisfies all of the clauses in $\mathcal{C}$, or equivalently α satisfies the CNF formula

$$\varphi(x_0, \ldots, x_{n-1}) = \bigwedge_{C \in \mathcal{C}} C(x_0, \ldots, x_{n-1}).$$

Lemma 5.3. *Let $\mathcal{C}$ be a set of clauses and let $C, D \in \mathcal{C}$. Then $\mathcal{C}$ and $\mathcal{C} \cup \{C \diamond D\}$ have the same set of satisfying assignments. In particular, if $C \diamond D$ is the empty clause, then $\mathcal{C}$ is unsatisfiable.*

Next time we'll see how repeated application of the resolution rule yields a correct algorithm for deciding satisfiability (and also how to extract a satisfying assignment at the end if the formula is satisfiable).