CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 15: Logic

Harvard SEAS - Fall 2024 *2024-10-24*

1 Announcements

- SRE next class (10/29). Remember to prepare and come on time!
- Salil's OH after class today (no OH next week due to conference travel), Anurag Zoom OH tomorrow 1:30-2:30.
- Remember to *acknowledge all sources* on your problem sets, including students not currently in the class and ChatGPT/AI.
- Problem Set 6 distributed, due 10/30.
- Reflection summaries
  1. PS3: most of you found RAM/WordRAM Models useful in concretizing constant-time operations and connecting to real-world computing, though some (10+) didn't see value in covering it.
  2. PS4: Top two topics that you all wanted to study more (pre-midterm) were RAM/Word-RAM and BSTs. Also want better understanding of what constitutes a rigorous proof, how to perform reductions, and intuition for randomness being useful.
  3. SRE3 (connected components): Found to be our hardest SRE so far. Students felt that examples and diagrams would have been helpful. We are hoping that you (senders) will try to come up with them!
  4. SRE4 (coloring): You all seemed to enjoy this one, and benefitted from past experience with SREs. Again visuals very helpful!

Recommended Reading (good for practice problems, too):

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

2 Propositional Logic

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

Definition 2.1 (boolean formulas, 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).

The Lewis–Zax text contains a 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 (DNF and CNF formulas)**.**

- A *literal* is a variable (e.g. x_i) or its negation ($\neg x_i$).
- A *term* is an AND of a sequence of literals.
- A *clause* is an OR of a sequence of literals.
- A boolean formula is in *disjunctive normal form (DNF)* if it is the OR of a sequence of terms.
- A boolean formula is in *conjunctive normal form (CNF)* if it is the AND of a sequence of clauses.

By convention, an empty term is always true and an empty clause is always false. (**Q:** Why is this a sensible convention?)

Q: For exach of the examples above, is it in DNF or CNF?

A:

Simplifying clauses. 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.

One reason that DNF and CNF are used so commonly is that they can express all boolean functions:

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.

$\square$

Example: The majority function on 3 bits can be written in CNF as follows:

$$\psi(x_0, x_1, x_2) =$$

with the clauses corresponding to the 4 non-satisfying assignments $(0,0,0), (1,0,0), (0,1,0), (0,0,1)$. This example shows that the DNF and CNF given by the general construction are not necessarily the smallest ones possible for a given function, as the majority function can also be expressed by the following simpler CNF formula:

$$(x_0 \vee x_1) \wedge (x_0 \vee x_2) \wedge (x_1 \vee x_2).$$

3 Computational Problems in Propositional Logic

Here are three natural computational problems about Boolean formulas:

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

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

Q: 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 in encoding other problems. Thus any effort gone into optimizing algorithms for (CNF-)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 with kn variables and $n + km$ clauses.*

Proof. Given $G = (V, E)$ and $k \in \mathbb{N}$, we will construct a CNF ϕ_G that captures the graph k-coloring problem. In ϕ_G, we introduce *indicator variables* $x_{v,i}$ where $v \in V$ and $i \in [k]$, which intuitively are meant to correspond to vertex v being assigned to color i.

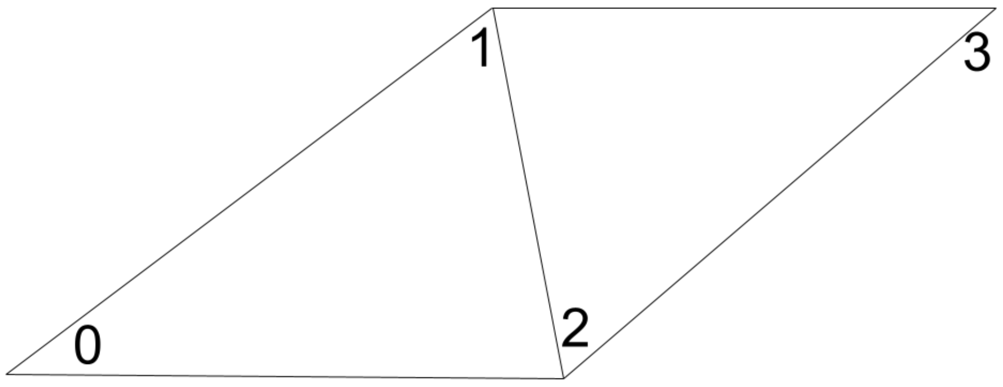
We then have a few types of clauses:

1.

2.

3.

For instance, if G is the graph below,



then we make the following SAT instance ϕ_G:

We then call the SAT oracle on ϕ_G and get an assignment α. If $\alpha = \perp$, we say G is not k-colorable. Otherwise, we construct and output the coloring f_α given by:

$$f_\alpha(v) =$$

The runtime essentially follows from our description.

For correctness, we make two claims:

Claim 4.2. *If G has a valid k coloring, ϕ_G is satisfiable.*

$\implies$ don't incorrectly output $\perp$.

Claim 4.3. *If α satisfies ϕ_G, then f_α is a proper k-coloring of G.*

$\implies$ if we output a coloring, it will be proper.

Both of these claims are worth checking. Note that f_α is *well-defined* because α satisfies clauses of type 1 and is *proper* due to clauses of type 2. $\square$

Unfortunately, the fastest known algorithms for 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.

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

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 best known SAT solvers implicitly use the technique of *resolution.* The idea of resolution 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 thus we have a proof that the original formula is unsatisfiable) or we cannot derive any more clauses (in which case we can efficiently construct a satisfying assignment).

Definition 5.1 (resolution rule)**.** For clauses C and D, define their *resolvent* to be

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

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

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

The validity of the resolution deduction rule is given by the following:

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.*

Given this, we can start with a set $\mathcal{C}$ of clauses from a CNF formula and keep adding resolvents of clauses in $\mathcal{C}$ until we cannot add any new ones. The Resolution Theorem tells us that this suffices to solve Satisfiability.

Theorem 5.4 (Resolution Theorem)**.** *Let $\mathcal{C}$ be a set of clauses over variables $x_0, \ldots, x_{n-1}$. Suppose that $\mathcal{C}$ is* closed under resolution*, meaning that for every $C, D \in \mathcal{C}$, we have $C \diamond D \in \mathcal{C}$. Then:*

1. *$\mathcal{C}$ is unsatisfiable iff $\emptyset \in \mathcal{C}$.*
2. *If $\emptyset \notin \mathcal{C}$, then we can find a satisfying assignment to $\mathcal{C}$ in time $O(n + k \cdot |\mathcal{C}|)$.*

Example: $\phi(x_0, x_1, x_2) = (\neg x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \wedge (x_0 \vee x_1 \vee x_2) \wedge (\neg x_2)$

Example 2: $\psi(x_0, x_1, x_2, x_3) = (\neg x_0 \vee x_3) \wedge (x_0 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3)$