CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 17: Logic

Harvard SEAS - Fall 2025 *2025-10-30*

1 Announcements

- Salil's in-person OH this week moved to FRIDAY 4-4:45pm.
- SRE on Tuesday.
- PS5 feedback
  - Repeat questions for Zoom: I'm trying to do so; sorry when I forget!
  - Filled-in lecture notes = Hesterberg–Vadhan textbook! (See Maxwell's FAQ.)
  - Median time: 6hrs College, 7.5hrs DCE; 75th percentile: 10hrs College; 10hrs DCE.
  - "I wish he would explain notation more!": please ask (either raising your hand or on Ed)!
- Midterm feedback
  - Some felt it met expectations, others felt it was harder.
  - Wish for more practice problems. Section notes have lots of problems. We'll see if we can distribute an extra practice exam for the final.
- Reflection question about struggle
  - "I'm in such a rush to get work done that I don't accept sometimes that its okay to sit down and take time to struggle through a problem for hours."
  - "However, looking back at what I was struggling with before, and seeing how I overcame those previous challenges brings me hope that I can continue doing better if I just keep trying..."

Recommended Reading:

- Hesterberg–Vadhan 17
- Lewis–Zax 9–10
- Roughgarden IV, Sec. 21.5, Ch. 24

2 Propositional Logic

Recap. Examples:

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

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

We now turn to two important special cases of boolean formulas.

Definition 2.1 (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.

Q: What truth value should an empty term (an AND of no literals) have?

Q: What truth value should an empty clause have?

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

Lemma 2.2. *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.

□

Example 2.3. The majority function on 3 bits can be written in CNF as follows:

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

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 simpler CNF formula

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 $\varphi(\alpha) = 1$ (if one exists)

Computational Problem Satisfiability

Input: A CNF formula φ on n variables

Output: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$ (if one exists)

Computational Problem CNF-Satisfiability

Input: A DNF formula φ on n variables

Output: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$ (if one exists)

Computational Problem DNF-Satisfiability

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

4 Modelling using Satisfiability

One reason that SAT is important is its richness in encoding other problems. Thus any improvements to algorithms for (CNF-)SAT (aka "SAT Solvers") can be easily be applied to many other problems we want to solve.

Theorem 4.1. Graph Coloring *with k colors 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 formula ϕ_G that captures the constraints of the Graph Coloring problem. The variables of ϕ_G are *indicator variables* $x_{v,i}$ for each $v \in V$ and $i \in [k]$, which intuitively are meant to correspond to vertex v being assigned to color i.

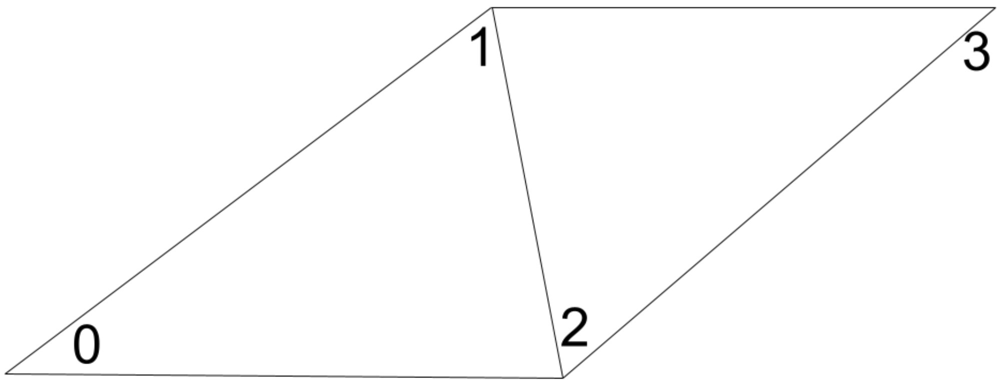
We then have a few types of clauses:

1.

2.

3.

For instance, if G is the graph below and $k = 3$,



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, then ϕ_G is satisfiable.*

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

Both of these claims are worth checking. Note that f_α is *well-defined* because α satisfies clauses of type 1 (so the definition of f_α doesn't try to take the minimum of an empty set) and is *proper* due to clauses of type 2. □

Unfortunately, the fastest known algorithms for SAT 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 instances arising in the 2016 US Federal Communications Commission (FCC) auction to reallocate wireless spectrum.

As we will see repeatedly in the rest of the course, SAT can encode a very wide variety of problems of interest, with the next example being Longest Path in the Sender–Receiver Exercise next week.

Thus motivated, in the next classwe will turn to algorithms for SAT, to get a taste of some of the ideas that go into SAT Solvers.

5 The Basic Resolution Algorithm

SAT Solvers are algorithms to solve CNF-Satisfiability. Although they have worst-case exponential running time, on many "real-world" instances, they terminate surprisingly 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 gives 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).

Our deduction rule will make use of a clause simplification procedure `Simplify`, which takes a clause C and simplifies it as follows:

1.

2.

3.

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

$$C \diamond D = \begin{cases} \rule{8cm}{0.4pt} & \text{if } \ell \text{ is a literal s.t. } \ell \in C \text{ and } \neg\ell \in D \\ 1 & \text{if there is no such literal } \ell \end{cases}$$

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

In the special case where $C = \ell, D = \neg\ell$, we obtain the empty clause $(\ )$, which is always false:

$$(\ell) \diamond (\neg\ell) = (\ ) = \text{FALSE}.$$

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

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

Intuition. The intuition behind resolution can be seen from the following example. Consider a CNF-Satisfiability instance that includes two clauses $C = (\neg x_0 \vee x_1)$ and $D = (\neg x_1 \vee x_2)$, along with other clauses. Since a satisfying assignment must make both C_1 and C_2 true, there is an implicit dependence between x_0 and x_2:

Following the definition, this is precisely the resolvent:

$$C \diamond D = (\neg x_0 \vee x_1) \diamond (\neg x_1 \vee x_2) =$$

Example 5.3.

$$(x_0 \vee \neg x_1 \vee x_3 \vee \neg x_5) \diamond (x_1 \vee \neg x_4 \vee \neg x_5) =$$

Example 5.4. We could also have a clause that appears to be resolvable in two ways:

$$(x_0 \vee x_1 \vee \neg x_4) \diamond (\neg x_0 \vee x_2 \vee x_4) =$$

The following theorem gives us a criterion to decide if a set of clauses is satisfiable. Note that resolution plays a crucial rule here.

Theorem 5.5 (Resolution Theorem). *Let $\mathcal{C}$ be a set of clauses over n 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 $C \diamond D \in \mathcal{C} \cup \{1\}$. Then:*

1. *$(\ ) \in \mathcal{C}$ iff*
2. *If $(\ ) \notin \mathcal{C}$, then* `ExtractAssignment`$(\mathcal{C})$ *finds where* `ExtractAssignment` *is an algorithm described in Section 6.*

Thus we can obtain an algorithm for solving CNF-SATISFIABILITY as follows. We start with the set of clauses $C_0, C_1, \cdots, C_{m-1}$ that appear in the CNF φ, simplify all the clauses in φ and then:

1. Resolve C_0 with each of $C_1, \ldots, C_{m-1}$, adding any new clauses obtained from the resolution $C_m, C_{m+1}, \ldots$. If the empty clause ( ) is found, return `unsatisfiable`.

2. Resolve C_1 with each of $C_2, \ldots, C_{m-1}$ as well as with

3. Resolve C_2 with each of $C_3, \ldots, C_{m-1}$ as well as with

4. etc.

5. Run `ExtractAssignment` on the set of all clauses and return the satisfying assignment.

Pseudocode is given as Algorithm 5.2.

```
   ResolutionInOrder($\varphi$):
   Input      : A CNF formula $\varphi(x_0, \ldots, x_{n-1})$
   Output     : Whether $\varphi$ is satisfiable or unsatisfiable
0  Let $C_0, C_1, \ldots, C_{m-1}$ be the clauses in $\varphi$, after simplifying each clause;
1  $i = 0$ ;  /* clause to resolve with others in current iteration */
2  $f = m$ ;     /* start of 'frontier' - new resolvents from current
    iteration */
3  $g = m$ ;                                 /* end of frontier */
4  while $f > i + 1$ do
5      foreach $j = i + 1$ to $f - 1$ do
6          $R = C_i \diamond C_j$;
7          if $R = 0$ then return unsatisfiable;
8          else if $R \notin \{1, C_0, C_1, \ldots, C_{g-1}\}$ then
9              $C_g = R$;
10             $g = g + 1$;
11     $f = g$;
12     $i = i + 1$
13 return ExtractAssignment(($C_0, C_1, \ldots C_{g-1}$))
```

Algorithm 5.2: A Resolution-based SAT Algorithm

Example 5.6. Let's apply Algorithm 5.2 to the following formula:

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

Example 5.7. $\psi(x_0, x_1, x_2, x_3) = (\neg x_0 \vee x_3) \wedge (x_0 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3)$

.

6 Assignment Extraction

We begin by describing the `ExtractAssignment` algorithm for finding a satisfying assignment to a set $\mathcal{C}$ of clauses that are closed under resolution and don't contain $(\ )$. Specifically, we generate our satisfying assignment one variable v at a time in the following manner:

1. If $\mathcal{C}$ contains a singleton clause (v), then
2. If it contains a singleton clause $(\neg v)$ then
3. If it contains neither (v) nor $(\neg v)$, then
4. $\mathcal{C}$ cannot contain both (v) and $(\neg v)$, because

Once we have assigned a variable to a value, we set that variable's value in every clause and simplify.

Formally, the algorithm works as follows:

```
  ExtractAssignment($\mathcal{C}$):
  Input  : A closed and simplified set $\mathcal{C}$ of clauses over variables
           $x_0, \ldots, x_{n-1}$ such that $(\ ) \notin \mathcal{C}$
  Output : An assignment $\alpha \in \{0,1\}^n$ that satisfies all of the clauses
           in $\mathcal{C}$
0 foreach $i = 0, \ldots, n-1$ do
1     if $(x_i) \in \mathcal{C}$ then $\alpha_i = 1$;
2     else $\alpha_i = 0$;
3     $\mathcal{C} = \mathcal{C}|_{x_i = \alpha_i}$, meaning that we set $x_i = \alpha_i$ and then simplify all clauses
        (see Sec. ??)
4 return $\alpha$
```

Algorithm 6.2: Assignment extraction algorithm

Example 6.1. Consider applying Algorithm 6.2 to the set of clauses derived from the formula in Example 5.7 above:

$$(\neg x_0 \vee x_3), (x_0 \vee \neg x_3), (\neg x_1 \vee x_2), (\neg x_2 \vee x_1), (\neg x_3), (\neg x_0)$$