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

Lecture 16: Resolution

Harvard SEAS - Fall 2022 *Oct. 27, 2022*

1 Announcements

- PS7 posted, due 11/9 (week after next)
- Sender-Receiver Exercise Tue 11/1: Senders and Receivers should all prepare.
- Adam & Salil OH after class 11:15-12:15, SEC 2.122
- For more individualized help in OH, come to the underloaded ones: the earliest ones on Mon, Tue, Thu, Sun.
- Participation & learning portfolio highlights 1 due Sat.

2 Recap

- A *literal* is a variable (e.g. x_i) or its negation ($\neg x_i$). We define $\neg(\neg x_i) = x_i$.
- 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.
- It will be convenient to also allow 1 (true) to be a clause. 0 (false) is already a clause: the empty clause is always false

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 (SAT)

Motivation for SAT (and logic problems in general): can encode many other problems of interest

- Graph Coloring (last time)
- Longest Path (SRE 4)
- Independent Set (section)
- Programming Team (ps7)
- Program Analysis (lec24)
- and much more (lec19)

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. Roughly, those instances had $k = 23$ colors (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). 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!

3 Resolution

Definition 3.1 (clause simplification)**.** Given a clause B, Simplify(B) returns 1 if B contains both a literal and its negation, and otherwise removes duplicates of literals from B and sorts the variables in B according to a fixed ordering on the variables (e.g. x_0, x_1, ...).

Examples:
Duplicate removal:

$$\text{Simplify}(x_3 \vee \neg x_2 \vee x_0 \vee x_3) = (x_0 \vee \neg x_2 \vee x_3)$$

Clause with both literal and its negation:

$$\text{Simplify}(x_1 \vee x_4 \vee x_2 \vee \neg x_4) = 1$$

Definition 3.2 (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 removing all occurrences of literal ℓ from clause C.

Examples:
Below, we perform the resolution rule by resolving on x_1:

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

For singleton clauses:

$$(x_0) \diamond (\neg x_0) = \text{empty clause} = 0.$$

We could also have a clause that appears to be resolvable in two ways, either resolving on x_0 or x_4:

$$(x_0 \vee x_1 \vee \neg x_4) \diamond (\neg x_0 \vee x_2 \vee x_4) = \text{Simplify}(x_1 \vee \neg x_4 \vee x_2 \vee x_4) \text{ OR } = \text{Simplify}(x_0 \vee x_1 \vee \neg x_0 \vee x_2)$$

but both of these equal 1.

In general, if C and D can be resolved with respect to more than one literal ℓ, then for all choices of ℓ we will have Simplify$((C - \ell) \vee (D - \neg\ell)) = 1$, so $C \diamond D$ is well-defined.

The motivation for this definition is the following property:

Lemma 3.3. *If an assignment α satisfies clauses C and D, then α satisfies $C \diamond D$.*

Proof.
Case 1: $C \diamond D = 1$. Then, the assignment α is vacuously true, since any assignment satisfies $C \diamond D$.
Case 2: There exists some literal l s.t.

$$C \diamond D = \text{Simplify}((C - \ell) \vee (D - \neg\ell))$$

Let α be any assignment that satisfies C and D.

Case 2a: α satisfies ℓ.

Since α satisfies D and α does not satisfy $\neg\ell$, the assignment α must satisfy the remainder of the clause $D - \neg\ell$. Thus, α satisfies $(C - \ell) \vee (D - \neg\ell)$ and α satisfies $\text{Simplify}((C - \ell) \vee (D - \neg\ell))$.

Case 2b: α satisfies $\neg\ell$.

This proceeds similarly to **Case 2a**, with the logic applied to $C - \ell$ instead.

□

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

Definition 3.4 (Satisfiability)**.** 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

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

A corollary of Lemma 3.3 is the following:

Lemma 3.5. *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.*

This gives rise to the *resolution meta-algorithm* for deciding satisfiability of a CNF formula φ. We keep adding resolvents until we find the empty clause (in which case we know φ is unsatisfiable by Lemma 3.5) or cannot generate any more new clauses. There are many variants of resolution, based on different ways of choosing the order in which to resolve clauses. We give a particular version below, where starting with $\varphi = C_0 \wedge C_1 \wedge \cdots \wedge C_{m-1}$, we 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 $C_m, C_{m+1}, \ldots$
2. Resolve C_1 with each of $C_2, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Step 1, again adding any new clauses.
3. Resolve C_2 with each of $C_3, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Steps 1 and 2, again adding any new clauses.
4. etc.

Note that this process will resolve every pair of clauses, except for resolving C_i with resolvents of the form $C_i \diamond C_j$ for $j > i$. Omitting the latter is harmless by the following lemma:

Lemma 3.6. *For all clauses C and D, $C \diamond (C \diamond D) = 1$*

Proof. Exercise. □

Examples:

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

We write out the clauses explicitly:

$$C_0 = (\neg x_0 \vee x_1)$$
$$C_1 = (\neg x_1 \vee x_2)$$
$$C_2 = (x_0 \vee x_1 \vee x_2)$$
$$C_3 = (\neg x_2)$$

We can then begin to resolve clauses:

$$C_4 = C_0 \diamond C_1 = (\neg x_0 \vee x_2)$$
$$C_5 = C_0 \diamond C_2 = (x_1 \vee x_2)$$
$C_6 = C_0 \diamond C_3 = 1$ (since there is no common literal to resolve on)
$$C_7 = C_1 \diamond C_2 = (x_0 \vee x_2)$$
$$C_8 = C_1 \diamond C_3 = (\neg x_1)$$
~~$C_1 \diamond C_4 = 1$~~ (since we already have the clause 1)
$$C_9 = C_1 \diamond C_5 = (x_2)$$
$$C_{10} = C_2 \diamond C_3 = (x_0 \vee x_1)$$
$$C_{11} = C_3 \diamond C_4 = (\neg x_0)$$
$$C_{12} = C_3 \diamond C_5 = (x_1)$$
$$C_{13} = C_3 \diamond C_7 = (x_0)$$
$$C_{14} = C_3 \diamond C_9 = 0$$

Therefore, $\phi(x_0, x_1, x_2)$ is `unsatisfiable`.

For a second example:

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

Note that the first four clauses correspond to the palindrome formula. When we apply resolution to the above formula, we derive $(\neg x_0) = (\neg x_0 \vee x_3) \diamond (\neg x_3)$, leaving us with the following set of clauses:

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

Then we get stuck and cannot derive any new clauses. Then the Resolution Algorithm says that ψ is `satisfiable`.

In pseudocode:

```
ResolutionInOrder(φ)
  Input  : A CNF formula φ(x_0, ..., x_{n-1})
  Output : Whether φ is satisfiable or unsatisfiable
Let C_0, C_1, ..., C_{m-1} be the clauses in φ, after simplifying each clause;
i = 0 ;                  /* clause to resolve with others in current iteration */
f = m ;     /* start of 'frontier' - new resolvents from current iteration */
g = m ;                                                  /* end of frontier */
while f > i + 1 do
    foreach j = i + 1 to f − 1 do
        R = C_i ⋄ C_j;
        if R = 0 then return unsatisfiable;
        else if R ∉ {C_0, C_1, ..., C_{g-1}} then
            C_g = R;
            g = g + 1;
    f = g;
    i = i + 1
return satisfiable
```

Algorithm 15 raises two questions:

1. (Termination) Why does resolution always terminate? And what is its runtime?
2. (Correctness) Is Algorithm 15 correct? If it ever derives the empty clause $R = 0$, we know that φ is unsatisfiable (why?) but if never generates the empty clause, can we be sure that φ is satisfiable?

4 Termination and Efficiency

Q: Why does resolution terminate?

A: For Question 1, note that resolution always terminates because there are only finitely many clauses that can be generated on n variables, namely at most $3^n + 1$. (The base is 3 since for each variable we can either include it, include its negation, or not include it at all.) The $+1$ accounts for the "clause" (1).
Q: What is the runtime of resolution?

A: Letting $\mathcal{C}_{fin}$ be the final set of clauses when we stop running, we have runtime $O(k_{fin} \cdot |\mathcal{C}_{fin}|^2)$, where k_{fin} is the maximum *width* (number of literals) among the clauses in $\mathcal{C}_{fin}$ Using $k_{fin} \leq n$ and $|\mathcal{C}_{fin}| \leq 3^n$, we have a worst-case runtime $O(n \cdot 9^n)$, which is worse than exhaustive search over the 2^n satisfying assignments.

However, in many cases, there is a *short* proof of unsatisfiability that resolution will find. One case is for the 2-SAT problem, defined as follows:

Input	: A CNF formula φ on n variables in which each clause has width at most k (i.e. contains at most k literals)
Output	: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem k-SAT

Q: What is the runtime of Resolution for 2-SAT?

A: Note that we will never create a clause of size larger than 2 (this is not true in general for larger initial clauses - why is it true for 2?). By removing 1 literal from each clause and concatenating the remainder of each clause (which has at most 1 literal), the new clause also has at most 2 literals.

Thus, in this case we have $k_{fin} \leq 2$ and $|\mathcal{C}_{fin}| = O(n^2)$, since there are only $O(n^2)$ clauses of size at most 2. So resolution runs in time $O(2 \cdot (n^2)^2) = O(n^4)$ for 2-SAT. An additional factor of n can be saved by only trying to resolve each clause with the $O(n)$ other clauses that share a variable (with opposite sign), yielding a runtime of $O(n^3)$. In CS124, it is shown how to obtain runtime $O(n+m)$ for 2-SAT, where m is the number of clauses, by reduction to finding strongly connected components of directed graphs. Unfortunately, just like with coloring, once we switch from $k = 2$ to $k = 3$, the best known algorithms still have exponential ($O(c^n)$) worst-case runtimes.

5 Correctness and Assignment Extraction

We will prove:

Theorem 5.1. *Algorithm 15 is a correct algorithm for deciding SAT, and when it outputs* `satisfiable`*, a satisfying assignment can be extracted from the final set $\mathcal{C}_{fin}$ of clauses it produces in time $O(n + k_{fin} \cdot |\mathcal{C}_{fin}|)$, where k_{fin} is the maximum size among the clauses in $\mathcal{C}_{fin}$.*

Corollary 5.2. *2-SAT can be solved in time $O(n^3)$.*

To prove Theorem 5.1, we rely on the key property of the set of clauses generated by Resolution:

Definition 5.3. Let $\mathcal{C}$ be a set of clauses over variables $x_0, \ldots, x_{n-1}$. We say that $\mathcal{C}$ is *closed* if for every $C, D \in \mathcal{C}$, we have $C \diamond D \in \mathcal{C}$.

Observe that if $\mathcal{C}$ is a closed, nonempty set of clauses, then $1 \in \mathcal{C}$, since $C \diamond C = 1$ for any $C \in \mathcal{C}$.

Lemma 5.4. *Let $\mathcal{C}_{fin}$ be the final set of clauses in any execution of Algorithm 15 that outputs* `satisfiable`*. Then $\mathcal{C}_{fin} \cup \{1\}$ is closed.*

Proof idea.
We can show this by contradiction: otherwise, we would not be at the final set of clauses in the algorithm. □

It turns out that closed sets of clauses that don't contain the empty clause are always satisfiable:

Lemma 5.5. *Let $\mathcal{C}$ be a closed set of clauses on n variables, each of width at most k, such that $0 \notin \mathcal{C}$. Then an assignment that satisfies $\mathcal{C}$ exists and can be found in time $O(n + k \cdot |\mathcal{C}|)$.*

Proof idea. We generate our satisfying assignment one variable v at a time:

1. If $\mathcal{C}$ contains a singleton clause (v), then we assign $v = 1$.
2. If it contains $(\neg v)$ then assign $v = 0$.
3. If it contains neither (v) nor $(\neg v)$, then assign v arbitrarily.
4. $\mathcal{C}$ cannot contain both (v) and $(\neg v)$, because $\mathcal{C}$ is closed and does not contain 0.

Once we have assigned a variable to a value, we set that variable's value in every clause and simplify. Crucially, we argue that even after assigning the variable, the set of clauses (a) does not contain 0, and (b) remains closed. (a) holds because of how we set v. Intuitively, (b) holds because assigning v and then resolving two resulting clauses C' and D' is equivalent to first resolving the original clauses C and D and then assigning v. We know that $C \diamond D \in \mathcal{C}$ by closure of $\mathcal{C}$, so we have $C' \diamond D'$ after assigning v. □

Example: Consider applying this procedure to the set of clauses derived from the formula ψ above:

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

Going through the variables in order, we set $x_0 = 0$ because we are forced to by the clause $(\neg x_0)$. After that, the clauses become:

$$(\neg 0 \vee x_3), (0 \vee \neg x_3), (\neg x_1 \vee x_2), (\neg x_2 \vee x_1), (\neg x_3), (\neg 0)$$

which simplifies to

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

These clauses don't include (x_1) or $(\neg x_1)$, so we can set x_1 as either 0 or 1. Arbitrarily choosing $x_1 = 1$, the clauses become:

$$(\neg x_3), (\neg 1 \vee x_2), (\neg x_2 \vee 1), (\neg x_3),$$

which simplifies to

$$(\neg x_3), (x_2).$$

Then we set $x_2 = 1$, and finally set $x_3 = 0$, yielding the satisfying assignment $(0, 1, 1, 0)$.

Lemmas 5.4 and 5.5 imply Theorem 5.1.

6 SAT Solvers

Enormous effort has gone into designing SAT Solvers that perform well on many real-world satisfiability instances, often but not always avoiding the worst-case exponential complexity. These methods are very related to resolution. In some sense, they can be viewed as interleaving the assignment extraction procedure and resolution steps, in the hope of quickly finding either a satisfying assignment or a proof of unsatisfiability. For example, they start by assigning a variable (say x_0) to a value $\alpha_0 = 0$. Recursing, they may discover that setting $x_0 = 0$ makes the formula unsatisfiable, in which case they backtrack and try $x_0 = 1$. But in the process of discovering the unsatisfiability of $\mathcal{C}|_{x_0=\alpha_0}$, they may discover many new clauses (by resolution) and these can the translated to resolvents of $\mathcal{C}$ (in a manner similar to Lemma 7.3 below). These new "learned clauses" then can help improve the rest of the search. Many other heuristics are used, such as always setting a variable v as soon as a unit clause (v) or $(\neg v)$ is derived, and carefully selecting which variables and clauses to process next.

7 Formalizing Assignment Extraction

This section is optional reading, to give you more precision and proof details about the assignment extraction procedure. We introduce the following notation:

Definition 7.1. For a (simplified) clause C, a variable v, and an assignment $a \in \{0,1\}$, we write $C|_{v=a}$ to be the simplification of clause C with v set to a. That is,

1. if neither v nor $\neg v$ appears in C, then $C|_{v=a} = C$,
2. if v appears in C and $a = 0$, $C|_{v=a}$ equals C with v removed,
3. if $\neg v$ appears in C and $a = 1$, $C|_{v=a}$ equals C with $\neg v$ removed,
4. if v appears in C and $a = 1$ or if $\neg v$ appears in C and $a = 0$, $C|_{v=a} = 1$.

(We do not need to address the case that both v and $\neg v$ appear in C, since we assume that all clauses are simplified.)

Definition 7.2. For a set $\mathcal{C}$ of clauses, a variable v, and an assignment $a \in \{0,1\}$, we write

$$\mathcal{C}|_{v=a} = \{C|_{v=a} : C \in \mathcal{C}\}.$$

Observe that the satisfying assignments of $\mathcal{C}|_{v=a}$ are exactly the satisfying assignments of $\mathcal{C}$ in which v is assigned a.

Here is pseudocode for assignment extraction algorithm:

```
ExtractAssignment(C)
  Input  : A closed and simplified set C of clauses over variables x_0, ..., x_{n-1} such that
           0 ∉ C
  Output : An assignment α ∈ {0,1}^n that satisfies all of the clauses in C
foreach i = 0, ..., n − 1 do
    if (x_i) ∈ C then α_i = 1;
    else α_i = 0;
    C = C|_{x_i=α_i};
return α
```

To analyze the correctness of this algorithm, we prove the following:

Lemma 7.3. *Let $\mathcal{C}$ be a set of clauses, v a variable, and $a \in \{0,1\}$ an assignment to v. If $\mathcal{C}$ is closed, then so is $\mathcal{C}|_{v=a}$.*

Proof. Let $C|_{v=a}$ and $D|_{v=a}$ be any two clauses in $\mathcal{C}|_{v=a}$, where $C \in \mathcal{C}$ and $D \in \mathcal{C}$. We need to show that $C|_{v=a} \diamond D|_{v=a} \in \mathcal{C}|_{v=a}$. By definition,

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

In the former case (where we resolve on ℓ and $\neg\ell$), we have

$$C|_{v=a} \diamond D|_{v=a} = \text{Simplify}((C|v = a - \ell\}) \vee (D|_{v=a} - \neg\ell)) = \text{Simplify}((C - \ell) \vee (D - \neg\ell))|_{v=a} = (C \diamond D)|_{v=a}.$$

That is, we could have resolved on literal ℓ first, then set $v = a$. Since $\mathcal{C}$ is closed, $C \diamond D \in \mathcal{C}$, and hence $(C \diamond D)|_{v=a} \in \mathcal{C}|_{v=a}$. In the latter case (there is no such literal ℓ), we have

$$C|_{v=a} \diamond D|_{v=a} = 1 = 1|_{v=a} \in \mathcal{C}|_{v=a}.$$

□

Proof of Lemma 5.5. Lemma 7.3 implies the correctness of `ExtractAssignment(`$\mathcal{C}$`)`. It ensures (by induction) that as we assign $x_0 = \alpha_0, x_1 = \alpha_1, \ldots$, the set $\mathcal{C}$ of variables remains closed. This also implies (by induction) that we never derive the empty clause: since $\mathcal{C}$ is closed and does not contain the empty clause, it cannot contain both (x_i) and $(\neg x_i)$, so our choice of α_i ensures that $\mathcal{C}|_{x_i=\alpha_i}$ does not contain the empty clause. Lemma 5.5 now follows by observing that Algorithm 6 can be implemented in time $O(n + k \cdot |\mathcal{C}|)$. □