CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 18: Resolution

Harvard SEAS - Fall 2021 *Nov. 4, 2021*

1 Announcements

- PS8 posted
- Active learning today!
- Salil OH after class in person, M1:30-2:30 on zoom

2 Resolution

Definition 2.1 (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`, and Simplify(B) removes duplicates of literals from clause B, and returns 1 if B contains both a literal and its negation. 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 Simplify$((C - \{\ell\}) \vee (D - \{\neg\ell\})) = 1$, so $C \diamond D$ is well-defined.

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

Definition 2.2. 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 2.3. *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.*

Proof.
Omitted.

□

This gives rise to the *resolution 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 2.3) 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:

1. Resolve C_0 with $C_1, \ldots, C_{m-1}$,
2. Resolve C_1 with $C_2, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Step 1.
3. Resolve C_2 with $C_3, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Steps 1 and 2.
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 because $C_i \diamond C_j$ cannot contain the negation of any literal in C_i (without simplifying to 1).

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:

$$C_4 = C_0 \diamond C_1 = (\neg x_0 \vee x_2)$$
$$C_5 = C_0 \diamond C_2 = (x_1 \vee x_2)$$
$$\cancel{C_0 \diamond C_3 = 1}$$
$$C_6 = C_1 \diamond C_2 = (x_0 \vee x_2)$$
$$C_7 = C_1 \diamond C_3 = (\neg x_1)$$
$$\cancel{C_1 \diamond C_4 = 1}$$
$$C_8 = C_1 \diamond C_5 = (x_2)$$
$$C_9 = C_3 \diamond C_4 = (\neg x_0)$$
$$C_{10} = C_3 \diamond C_5 = (x_1)$$
$$C_{11} = C_3 \diamond C_6 = (x_0)$$
$$C_{12} = C_3 \diamond C_8 = \emptyset = \text{FALSE}$$

Therefore, $\phi(x_1, x_2, x_3)$ 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:

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

Algorithm 15 raises two questions:

1. Why does resolution always terminate? And what is its runtime?
2. 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?

For Question 1, note that resolution always terminates because there are only finitely many clauses that can be generated on variables, namely at most 3^n. (The base is 3 since for each variable we can either include it, include its negation, or not include it at all.)

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 size (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. But 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:

Note that when we use resolution to solve 2-SAT (i.e. SAT where all clauses are of size at most 2), we will start with clauses of size at most 2 and 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?). 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

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

Computational Problem k-SAT

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 $O(n^3)$. On PS9, you'll show how to reduce the runtime further to $O(nm)$, for 2-SAT instances with n variables and m clauses. 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.

For Question 2, we show how to extract a satisfying assignment from the final set $\mathcal{C} = \{C_0, C_1, \ldots, C_{g-1}\}$ generated by Resolution. We will generate our satisfying assignment one variable v at a time:

1. If $\mathcal{C}$ contains a singleton clause (v), then we know we need to assign $v = 1$.
2. If it contains $(\neg v)$ then we know we need to assign $v = 0$.
3. If it contains neither (v) nor $(\neg v)$, then intuitively we should be able to assign v arbitrarily.
4. $\mathcal{C}$ cannot contain both (v) and $(\neg v)$, since otherwise Resolution would have derived the empty clause 0.

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

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 $(0, 1, 1, 0)$.

In the section below (which we did not cover in class and is optional reading) we will formalize this assignment extraction procedure and use it to prove:

Theorem 2.4. *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 $O(n \cdot |\mathcal{C}_{fin}|)$.

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 x_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 3.5 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.

3 Proof of Correctness (Supplementary Reading)

To make assignment extraction precise, we introduce the following notation:

Definition 3.1. 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 3.2. 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.

To formalize the conditions under which assignment-extraction succeeds, we introduce the following definition:

Definition 3.3. 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 $C \diamond D \in \mathcal{C}$ or $C \diamond D = 1$.

Note that at the end of Resolution, $\mathcal{C}$ is closed. Here is pseudocode for assignment extraction algorithm:

```
ExtractAssignment(C)
  Input  : A closed set C 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
```

Lemma 3.4. *Given a closed set $\mathcal{C}$ of clauses over variables $x_0, \ldots, x_{n-1}$ such that $0 \notin \mathcal{C}$,* `ExtractAssignment(C)` *generates an assignment $\alpha \in \{0,1\}^n$ that satisfies all clauses in $\mathcal{C}$ in time $O(n \cdot |\mathcal{C}|)$. (In particular, every such set $\mathcal{C}$ must be satisfiable.)*

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

Lemma 3.5. *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 $D \in \mathcal{C}$. Then we observe that

$$C|_{v=a} \diamond D|_{v=a} = (C \diamond D)|_{v=a},$$

i.e. the resolvent of $C|_{v=a}$ and $D|_{v=a}$ can be also obtained by first resolving C and D and then setting $v = a$. (This fact requires a proof, which we omit.) Since $\mathcal{C}$ is closed, we have $C \diamond D \in \mathcal{C} \cup \{1\}$, and thus $(C \diamond D)|_{v=a} \in \mathcal{C} \cup \{1\}$.

□

Lemma 3.5 implies the correctness of `ExtractAssignment(C)` (Lemma 3.4). 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 3.4 now follows by observing that Algorithm 6 can be implemented in time $O(n \cdot |\mathcal{C}|)$. This completes our proof of correctness of Resolution (Algorithm 15), yielding Theorem 2.4.