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

Lecture 16: Resolution

Harvard SEAS - Fall 2024 *Oct. 29, 2024*

1 Announcements

- SRE 5 today
- Midterm regrade requests due tonight.
- Anurag's in person OH moved to Thur 11AM - 12 PM.
- A reminder that both the psets and lectures are part of the learning material.
- Pset5 reflection responses:
  - Most common questions students struggled with - Word-RAM to RAM simulations, Asymptotic runtime analysis and notation.
  - Students found that struggling through the problem was valuable and led to better understanding and problem solving skills.
  - Students learned how to break down problems and to ask for help, which is great!

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)

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). We define a function Simplify which takes a clause and performs those simplifications: 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.

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

- Graph Coloring (last time)
- Longest Path (SRE 5)
- Independent Set (section)
- Program Analysis (lec24)
- 3D Matching (pset 7)
- 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.

3 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 3.1 (resolution rule)**.** For clauses C and D, define their *resolvent* to be

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

Here $C - \{\ell\}$ means remove literal ℓ from clause C, and 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 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 use our definition from Lecture 15 that empty clause is always false and obtain

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

Intuition: The intuition behind resolution can be seen from the following example. Consider two clauses $C_1 = (\neg x_0 \vee x_1)$ and $C_2 = (\neg x_1 \vee x_2)$. If both C_1, C_2 are required to be true (which is the goal of the CNF-Satisfiability problem), there is an implicit dependence between x_0 and x_2, as follows. We can't set $x_0 = 1, x_2 = 0$ since that would render the first clause as x_1 and the second

clause as $\neg x_1$ and one of them will be unsatisfiable. Thus, the implicit dependence between x_0 and x_2 is captured by requiring that the clause $(\neg x_0 \vee x_2)$ be true.

Following the definition, this is precisely the resolvent of C_1, C_2:

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

Example 2:

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

Example 3: We could also have a clause that appears to be resolvable in two ways:

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

but both of these classes are equivalent, since they are both TRUE. Thus, if there are multiple ways to resolve, all ways of resolving result in TRUE.