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

Lecture 17: Resolution and Church-Turing thesis

Harvard SEAS - Fall 2024 *Oct. 31, 2024*

1 Announcements

- Happy Halloween
- Pset7 out.
- Anurag's in person OH today Thur 11AM - 12 PM.

2 Resolution

Definition 2.1 (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 $\text{Simplify}((C - \{\ell\}) \vee (D - \{\neg\ell\})) = 1$, 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

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

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

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

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

Theorem 2.3 (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 $C \diamond D \in \mathcal{C}$. Then:*

1. *$\emptyset \in \mathcal{C}$ iff*
2. *If $\emptyset \notin \mathcal{C}$, then there is an algorithm* `ExtractAssignment(`$\mathcal{C}$`)` *that finds*

The `ExtractAssignment(`$\mathcal{C}$`)` algorithm is described in Section 3. The algorithm for solving the CNF-Satisfiability is informally described as follows. We start with a set $\mathcal{C}$ of clauses from a CNF formula and keep adding resolvents until we cannot add any new ones. At this point, we know that the new set of clauses is closed under resolution. Then Theorem 2.3 tells us that if $\emptyset$ is a clause, then the formula is unsatisfiable. If $\emptyset$ is not in the set of clauses, then `ExtractAssignment()` algorithm gives us a way to find a satisfying assignment.

A more formal version is 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 $\emptyset$ 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.

In pseudo-code, the algorithm can be written as follows.

```
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, after simplifying each clause;
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 ∉ {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 ExtractAssignment((C_0, C_1, ... C_{g-1}))
```

Algorithm 1: Resolution algorithm

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

.

3 Assignment extraction, runtime and correctness

The runtime and correctness of the resolution algorithm is based on Theorem 2.3. Thus, we give a proof sketch this theorem and also describe the `ExtractAssignment()` algorithm. The missing details are about the `ExtractAssignment()` algorithm and appear in Section 6.

Proof sketch of Theorem 2.3. First, suppose $\mathcal{C}$ is such that $\emptyset \in \mathcal{C}$. Since $\emptyset$ is trivially a false clause, no assignment can satisfy it. Hence $\mathcal{C}$ is unsatisfiable.

Next, suppose $\mathcal{C}$ is such that $\emptyset \notin \mathcal{C}$. We generate our satisfying assignment based on the following principles, for each variable v:

1. If $\mathcal{C}$ contains a singleton clause (v), then
2. If it contains $(\neg v)$ 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. 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.

The following algorithm formalizes this.

```
1 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
2 foreach i = 0, ..., n − 1 do
3 |   if (x_i) ∈ C then α_i = 1;
4 |   else α_i = 0;
5 |   C = C|_{x_i = α_i};
6 return α
```

Algorithm 2: Assignment extraction algorithm

□

Example of assignment extraction: Consider applying Algorithm 2 to the set of clauses derived from the formula in the Example 2 above:

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

Now, we consider the runtime and correctness of the resolution algorithm. Let $\mathcal{C}_{fin}$ be the final set of clauses produced in Algorithm 1. Let k_{fin} be the maximum *width* (number of literals) among the clauses in $\mathcal{C}_{fin}$.

Runtime: Before analysing the runtime of the resolution algorithm, lets understand why the resolution algorithm terminates.

We can now give a finer estimate of the runtime of the resolution algorithm.

Correctness:

3.1 Efficient algorithm for 2-SAT

Input	**:** A CNF formula φ on n variables in which each clause has width at most 2 (i.e. contains at most 2 literals)
Output	**:** An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem 2-SAT

Runtime of the resolution algorithm for 2-SAT:

3.2 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 `ExtractAssignment()` algorithm 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 $\mathcal{C}$ with x_0 set to α_0, they may discover many new clauses (by resolution) and these can be translated to resolvents of $\mathcal{C}$. 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.

4 Introduction to Limits of Computation

Thus far in CS 1200, we've focused on what algorithms can do, or what they can do efficiently. In the remainder of the course, we'll talk about what algorithms can't do, or can't do efficiently. In particular, recall Lecture 4's lemma about reductions:

Lemma 4.1. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

1. *If there exists an algorithm solving Γ, then there exists an algorithm solving Π.*
2. *If there does not exist an algorithm solving Π, then there does not exist an algorithm solving Γ.*
3. *If there exists an algorithm solving Γ with runtime $R(n)$, and $\Pi \leq_{T,q\times h} \Gamma$, then there exists an algorithm solving Π with runtime $O(T(n) + q(n) \cdot R(h(n)))$.*
4. *If there does not exists an algorithm solving Π with runtime $O(T(n) + q(n) \cdot R(h(n)))$, and $\Pi \leq_{T,q\times h} \Gamma$, then there does not exist an algorithm solving Γ with runtime $R(n)$.*

In the last unit of the course, we'll use the item 2: we'll find a problem Π which we can prove is not solved by any Word-RAM algorithm, then reduce Π to other problems Γ to prove that no Word-RAM algorithm solves them.

Similarly, in the upcoming second-last unit of the course, we'll use item 4: we'll assume that the problem $\Pi = SAT$ is not solved quickly by any Word-RAM algorithm, then reduce SAT to other problems Γ to prove that no Word-RAM algorithm solves them quickly.

Before we do so, let's consider how fundamental Word-RAM is to the statements above. That is, if we prove limitations of Word-RAM programs, are those limits specific to Word-RAM or are they more general/independent of technology? Could find substantially faster algorithms by choosing a different model of computation than Word RAM, like Python or Minecraft? The answer is conjectured to be "no".

To explain why, we'll first recall our simulation arguments which state that the same problems are solvable by Word-RAM programs, Python programs, and so on.

5 The Church–Turing Thesis

Theorem 5.1 (Turing-equivalent models)**.** *If a computational problem* Π *is solvable in one of the following models of computation, then it is solvable in all of them:*

Moreover, there is an algorithm (e.g. a RAM program) that can transform a program in any of these models of computation into an equivalent program in any of the others.

The Church–Turing Thesis: The equivalence of many disparate models of computation leads to the Church–Turing Thesis, which has (at least) two different variants:

1.

2.

This is not a precise mathematical claim, and thus cannot be formally proven, but it has stood the test of time very well, even in the face of novel technologies like quantum computers (which have

yet to be built in a scalable fashion); every problem that can be solved by a quantum algorithm can also be solved by a RAM program, albeit much more slowly.

Proof idea:

Simple and elegant models:

Input encodings:

5.1 The Strong (or Extended) Church–Turing Thesis

The Church–Turing hypothesis only concerns problems solvable at all by these models of computation (Word-RAM programs, etc.). We haven't even seen any problems that are *not* solvable by Word-RAM programs—that will be a topic for the end of the course. There is, however, a stronger version of the Church–Turing hypothesis that also covers the efficiency with which we can solve problems.

Extended Church–Turing Thesis v1:

The Strong Church–Turing Thesis is not a precise mathematical claim, and thus cannot be formally proven. In fact, randomized algorithms, massively parallel computers, and quantum computers all could potentially provide an exponential savings in runtime. (For randomized algorithms, however, it is conjectured that they provide only a polynomial savings, as discussed in Lecture 8.)

If we modify the statement with some qualifiers, then these challenges no longer apply:

Extended Church–Turing Thesis v2:

"Deterministic" rules out both randomized and quantum computation, as both are inherently probabilistic. "Sequential" rules out parallel computation. This form of the Extended Church–Turing Thesis has stood the test of time for the approximately fifty years since it was formulated, even as computing technology has changed tremendously in that time.

Considering computational efficiency when comparing models of computation will be the subject of the next few lectures.

6 Formalizing Assignment Extraction

This section is optional reading, to give you more precision and proof details about the assignment extraction algorithm.