University of California, Los Angeles
CS 281 Computability and Complexity

Instructor: Alexander Sherstov
Scribe: Boran Erol
Date: January 24, 2024

LECTURE
5

NP-Completeness and Cook-Levin Theorem

Outside of mathematics and theoretical computer science, almost all references to complexity theory are to NP-Completeness. This isn't without reason. Whether P equals NP is widely regarded as the most important open problem in theoretical computer science and NP-Completeness is a crucial notion in that was developed in humanity's quest to answer this question. In this lecture, we introduce NP-Completeness and prove the Cook-Levin Theorem, a cornerstone of complexity theory. Without a full mathematical understanding of this theorem, one can never hope to understand many important results in complexity theory.

5.1 NP-Completeness

For every language class, we have countably many languages. This can be hard to visualize. The notion of completeness hopes to solve this problem by finding a reference problem that abstracts away the complexity class. This way, we can focus on a single concrete problem.

Definition 5.1. Let $L, L' \subseteq \{0,1\}^*$ be two languages. L is **polynomial time reducible** to L' if there is a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L$ if and only if $f(x) \in L'$. We denote this by $L \leq_P L'$.

Notice that this is a relation on L. Moreover, $L \leq_P L$ for every language since we can just use the identity mapping. One can think of $L \leq_P L'$ as saying that L' is harder than L in the following sense: If you find a polynomial time algorithm A for L', this immediately produces a polynomial time algorithm for L by applying the reduction and applying A.

Reductions are famously counterintuitive. Even people that have worked on reductions for years often get the direction of the reduction wrong. One reason behind this might be that we're showing that a problem is hard by assuming a solution exists for that problem, which creates some psychological tension.

With this definition in our hands, we can define NP-Completeness. In a sense, NP-Complete problems are the hardest problems in NP. Any algorithm for an NP-Complete problem induces an algorithm for every NP problem with a polynomial overhead.

Definition 5.2. A language L is NP-Hard if for all $L' \in NP : L' \leq_P L$.

Definition 5.3. A language L is NP-Complete if L is in NP and is NP-Hard.

Note that a problem can be NP-Hard but not be in NP. One challenge problem asks us to prove that the halting problem is NP-Hard. However, it's not even decidable! Let's now prove a few simple propositions about NP-Hard and NP-Complete problems to familiarize ourselves.

Proposition 5.4. *The polynomial-time reducibility relation is transitive. In other words, for all languages $L, L', L'' : L \leq_P L' \wedge L' \leq_P L'' \implies L \leq_P L''$.*

Proof. Follows by composing the two mappings induced by the reductions. ∎

Proposition 5.5. *Let $L \subseteq \{0,1\}^*$ be an NP-Hard language. Then, $L \in P \implies P = NP$.*

Proof. Assume L is in NP. Let L' be in NP. Then, there exists some polynomial time computable f such that $x \in L' \iff f(x) \in L$. Given $x \in L$, we can compute $f(x)$ and decide whether $f(x) \in L$ in polynomial time. This implies that L' is in P. Since L' was arbitrary, $NP \subseteq P$ and we're done. ∎

Proposition 5.6. *Let $L \subseteq \{0,1\}^*$ be an NP-Complete language.*
Then, $L \in P$ if and only if $P = NP$.

Proof. The forward direction follows from the previous proposition. The converse is immediate since if L is NP-Complete, L is in NP. ∎

Like any definition in mathematics, one needs show that NP-Complete languages exist for the notion to be meaningful. Sometimes, people think that the importance of the Cook-Levin Theorem is that it produced an NP-Complete language. However, as the next theorem shows, producing NP-Complete languages isn't hard. The importance of the Cook-Levin Theorem is that it proves a natural problem that we encounter in real life is NP-Complete. This is significant, because we can then reduce SAT to other problems in order to prove that they're NP-Complete as well. In fact, Cook-Levin was a way to obtain PhDs for generations of students by reducing SAT to another problem. Nowadays, unfortunately for PhD students, people don't care as much.
In the next theorem and for the remainder of these notes, 1^n stands for the unary string where 1 is repeated n times.

Theorem 5.7. *Let $L = \{(M, x, 1^n) : M$ accepts (x, y) in time n for some $y\}$. L is NP-complete.*

Proof. Let's first prove that L is in NP. Let $(M, x, 1^n) \in L$. Then, there exists some y such that $M(x, y)$ accepts in time n. Without loss of generality, this implies that $|y| \leq n$ since the Turing Machine won't have time to read the input past that length. Thus, we can use y as the certificate and M as the verifier. Thus, L is in NP.

Let's now prove that L is NP-Hard. Let $A \in NP$ and $M(x, y)$ be the verifier for A that runs in time n^c. Notice that $x \in A \iff (M, x, 1^{n^c}) \in L$. Then, the function f defined by $f(x) = (M, x, 1^{n^c})$ produces the desired reduction. Clearly, f is polynomial time computable. ∎

5.2 CNF Formulas

Before we prove the Cook-Levin Theorem, we need to recall some definitions from logic.

Definition 5.8. $x_1, x_2, ..., x_n \in \{0, 1\}$ are called **Boolean variables.**

Definition 5.9. A **literal** is a Boolean variable x_i or its complement $\bar{x_i}$.

Literals are assignments to Boolean variables. If we have n Boolean variables $x_1, x_2, ..., x_n$, we get $2n$ literals $x_1, ..., x_n, \bar{x_1}, ..., \bar{x_n}$.

Definition 5.10. A **clause** is a disjunction of literals.

Definition 5.11. A **term** is a conjunction of literals.

Definition 5.12. A Boolean formula is said to be in **conjunctive normal form (CNF)** if it's a conjunction of clauses.

Definition 5.13. A Boolean formula is said to be in **disjunctive normal form (DNF)** if it's a disjunction of terms.

Notice that a conjunction of terms is just another term and a disjunction of clauses is just another clause. CNFs can be thought of as a list of necessary conditions (where each condition is a clause) and DNFs can be thought of as a list of sufficient conditions (where each condition is a term, or a partial assignment!). We also make a couple of definitions that'll be useful later in the lecture.

Definition 5.14. A Boolean formula is said to be a **k-CNF formula** if it's a CNF and if every clause has at most k literals.

Definition 5.15. A Boolean formula is said to be a **k-DNF formula** if it's a DNF and if every term has at most k literals.

To get some familiarity with these forms, let's try to write the majority function in CNF and DNF form. In a page or so, we're going to prove that every function has a CNF and DNF representation.

Let's first make the following observation:

Lemma 5.16. *The negation of a CNF formula with m clauses is a DNF formula with m terms. Similarly, the negation of a DNF formula with m terms is a CNF formula with n clauses.*

Proof. The proof follows immediately from generalized De Morgan's Laws. □

This observation is useful because we can prove a statement for DNFs and convert the statement to an equivalent statement about CNFs by considering the negation. We can now express the majority function in CNF form. Intuitively, this formula returns 1 when every subset of the variables that constitute a majority has at least one literal that is set to true. Notice that that this statement is true if and only if the majority of variables is set to true. Thus, we have the following expression:

$$MAJ(x_1, x_2, ..., x_n) = \bigwedge_{\substack{S \subseteq \{1,2,...,n\} \\ |S| \geq \frac{n}{2}}} \bigvee_{i \in S} x_i$$

The intuition behind the DNF expression is simpler. We simply check every subset that constitutes a majority and see if they're all set to true. This gives us the following expression:

$$MAJ(x_1, x_2, ..., x_n) = \bigvee_{\substack{S \subseteq \{1,2,...,n\} \\ |S| \geq \frac{n}{2}}} \bigwedge_{i \in S} x_i$$

The beautiful symmetry between the two expressions doesn't hold in general. The following theorem proves that every Boolean function can be represented in CNF and DNF form.

Theorem 5.17. *Every $f : \{0,1\}^n \to \{0,1\}$ can be represented in CNF form with less than 2^n clauses and DNF form with less than 2^n terms.*

Proof. Let $x \in \{0,1\}^n$ such that $f(x) = 1$. Notice that there are at most 2^n such x. For every such x, create a term by adding the literalx_i if $x_i = 1$ and $\bar{x_i}$ otherwise. Notice that the disjunction of all such terms produces a DNF. Since there are at most 2^n such x's, this DNF has at most 2^n terms. Now, notice that we can produce a DNF for the negation of f with 2^m terms. Negating this DNF produces a CNF for f with less than 2^n clauses, so we're done. ∎

5.3 Cook-Levin Theorem

We can finally prove the Cook-Levin Theorem. Let's first define the language SAT. In words, SAT contains all CNF formulas that have at least one satisfying assigment of variables. Formally:

$$SAT = \{\phi : \phi \text{ is a CNF formula such that } \phi \not\equiv 0\}$$

Clearly, $SAT \in NP$ since given a satisfying assignment, we can check its validity in polynomial time. The hard part of the proof is proving that SAT is NP-Hard.

In order to prove that SAT is NP-Hard, we're going to peek under the hood of the Turing Machine and capture the inner workings of the Turing Machine by a CNF formula. The Cook-Levin Theorem is a theorem born from computer engineering. The idea behind is not mathematical and the theorem doesn't use fancy mathematical tools.

The key idea of the following proof is the notion of a *tableau* of M on input (x, y). This tableau is a *complete* record of M's computation. Imagine someone pointed a camera under the hood of M as it was computing and took a snapshot at every step of the computation. We'll examine this tableau and try to extract a CNF formula from it in the reduction.

Theorem 5.18 (Cook-Levin'71). *SAT is NP-Complete.*

Proof. Let L be in NP. We will reduce L to SAT. Let $M(x, y)$ be the verifier for M. Without loss of generality, we can assume that M has a single tape. We now formally define the tableau of M on input (x, y).

The tableau is an $|x|^c$ by $|x|^c$ matrix. Row i of the tableau contains the contents of the tape at step i. Moreover, the cell containing the tape head is marked with a special symbol that also indicates the state of M at step i. Clearly, the first row of the tableau is going to have the initial state and the tape head at the first cell. The triangles at the leftmost cell indicate the left end of the tape.

Notice that we can produce "bogus" tableaus that don't correspond to an actual computation of M on x. For example, we can have two tape heads in a single row or make the tape head move 2 cells between two adjacent rows. Therefore, we'll call a tableau **valid** if it corresponds to an actual computation of M where M accepts. In other words, if every cell "follows" from the previous row according to M's specification, the tableau is valid. Notice that the fact that M halts and accepts 1 can be enforced by forcing the last row of the tableau to have the first cell contain a 1 and the HALTED state, as shown in the figure below.

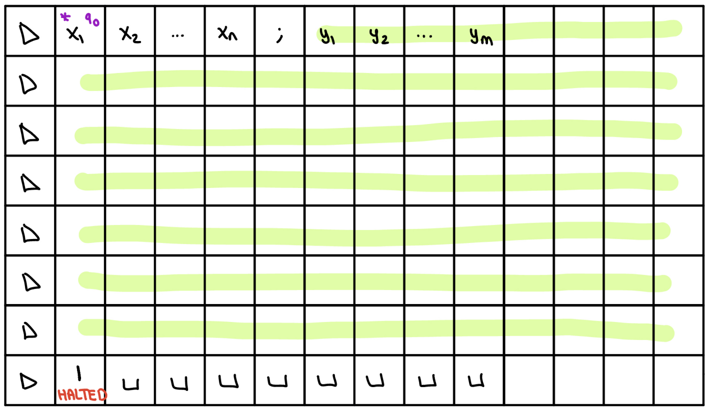

FIGURE 5.1: Tableau with the green cells highlighted.

Now, we make a seemingly innocent but powerful observation: x is in L if and only if there is an *assignment* to the green cells that make the tableau valid.

Notice that every cell of the tableau can be expressed using a constant number of bits. This is because M is fixed: we can encode the tape alphabet, states and the tape head position tracker into a constant binary string. Let's call this constant **k**.

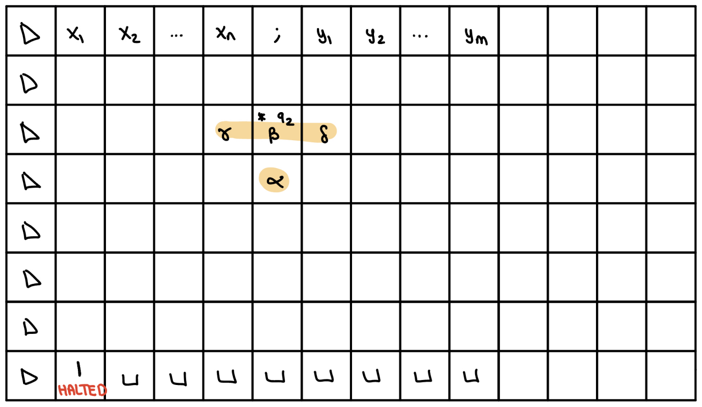

FIGURE 5.2: A cell is completely determined by the 3 cells in the previous row.

We now make a second powerful observation: the validity of cell T_{ij} only depends on the 3 cells above in the previous row. This is because the contents of a cell can only change if the tape head is currently on that cell, and the tape head can only move to a cell if it's in one of the adjacent cells or already on the cell.

Then, the formula for the validity of each cell depends on 4 cells (the 3 cells above and the cell itself). This can be expressed using a CNF formula of constant size less than 2^{4k} since the formula involves at most $4k$ bits. In order to check if the last row has the HALTED state with a 1 on the tape, we need a CNF with $O(|x|^c)$ clauses that check the content of each cell. Therefore, we can construct a CNF that checks the validity of a computation that has a polynomial size! We can reduce any string x to a formula ϕ_x that is satisfiable if and only if $x \in L$. We conclude the proof.

$\square$

While proving the Cook-Levin theorem we proved the following stronger result:

THEOREM 5.19. *Let $M(x, y)$ be a polynomial time Turing Machine. Then, there exists a polynomial time map from 1^n to a CNF formula ϕ such that*

$$\exists y : M(x, y) = 1 \iff \exists z : \phi(x, z) = 1$$

In order to appreciate interplay with the quantifiers, consider an M that only takes x as input. (To see that the theorem below holds, use the above theorem and take a Turing Machine M that ignores y.)

THEOREM 5.20. *Let $M(x)$ be a polynomial time Turing Machine. Then, there exists a polynomial time map from 1^n to a CNF formula ϕ such that*

$$M(x) = 1 \iff \exists z : \phi(x, z) = 1$$

This theorem states that a very powerful model of computation (Turing Machines) is equivalent to a simple and seemingly less powerful model of computation modulo a quantifier.

The extra computational power gained by adding certain logical quantifiers is a crucial question in computational complexity. Here's another version of these theorems:

THEOREM 5.21. *Let $M(x, y)$ be a polynomial time Turing Machine. Then, there exists a polynomial time map from 1^n to a DNF formula ψ such that*

$$\forall y : M(x, y) = 1 \iff \forall z : \psi(x, z) = 1$$

Proof. Use Theorem 5.19 in order to create ϕ such that

$$\exists y : (\neg M)(x, y) = 1 \iff \exists z : \phi(x, z) = 1$$

Negating both sides of the statement produces:

$$\forall y : M(x, y) = 1 \iff \forall z : \phi(x, z) = 0$$

Let $\psi = \neg\phi$. Since ϕ is a CNF of polynomial size, ψ is a DNF of polynomial size. □

5.4 NP-Completeness of 3-SAT

3-SAT is a further restriction of SAT where we restrict ϕ to a 3-CNF:

DEFINITION 5.22.

$$\text{3-SAT} = \{\phi : \phi \text{ is a 3-CNF formula such that } \phi \not\equiv 0\}$$

In fact, we can extend this definition as follows:

DEFINITION 5.23.

$$\text{k-SAT} = \{\phi : \phi \text{ is a k-CNF formula such that } \phi \not\equiv 0\}$$

Notice that a 1-SAT formula is just a term and thus 1-SAT $\in P$. Whether or not 2-SAT is in P is left as a challenge problem.

DEFINITION 5.24. A **Boolean formula** is a tree where internal nodes are labeled with $\wedge, \vee$ and the leaves are labeled with literals.

Here is an example of a Boolean formula:

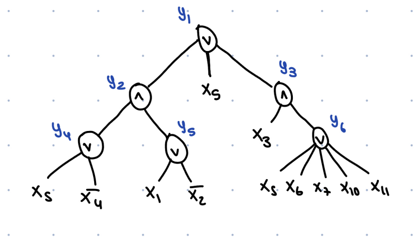

FIGURE 5.3: An example Boolean formula.

In order to prove that 3-SAT is NP Complete, we're going to reduce SAT to 3-SAT. We're going to use the following theorem in the proof:

Theorem 5.25. *Let ϕ be a Boolean formula. Then, there is a polynomial time computable 3-CNF Φ such that*

$$\phi(x) = 1 \iff \exists y : \Phi(x, y) = 1$$

Proof. Let ϕ be a Boolean formula. Without loss of generality, we can assume that ϕ has a fan-out of 2 since we can add extra internal nodes into ϕ and split up nodes with fan-out greater than two. Now, for every internal node i in ϕ, add an intermediate variable y_i as in Figure 5.3. (Notice that y_1 and y_6 have to be further split up in the figure first.) Having done this, we can now add the formula that represent the relationship of y_i to its children. For example, for y_2 in Figure 5.3, this would be $y_2 \iff y_4 \wedge y_5$. Since this is a Boolean formula with 3 variables, it has a CNF with at most 8 clauses. We can repeat this for every intermediate node and create a CNF formula. Since we want the CNF to output 1, we also add the clause y_1. Therefore, for a Boolean formula with m internal nodes, we get a CNF with $8m + 1$ clauses. $\square$

Theorem 5.26. *3-SAT is NP-Complete.*

Proof. 3-SAT is clearly in NP since we can verify a satisfying assignment in polynomial time. Thus, it suffices to reduce SAT to 3-SAT.

Let ϕ be a CNF formula. By the previous theorem, there exists a polynomial time computable map that takes ϕ to a 3-CNF Φ such that

$$\phi(x) = 1 \iff \exists y : \Phi(x, y) = 1$$

Quantifying both sides with $\exists x$, we get:

$$\exists x : \phi(x) = 1 \iff \exists x : \exists y : \Phi(x, y) = 1$$

In other words, ϕ is satisfiable if and only if ψ is satisfiable, so we're done. $\square$

5.5 Challenge Problems

1. Decide and prove whether 2-SAT is in P.
2. Without using Razborov-Smolensky, prove that PARITY requires an exponential size DNF/CNF formula.
3. Construct a polynomial size DNF f that requires an exponential size CNF.
4. Construct a polynomial size CNF f that requires an exponential size DNF.