CSCI 341–Fall 2024: Lecture Notes
Set 19: NP-Completeness

Edward Talmage

December 6, 2024

To better understand the relationship between P and NP, consider the hardest (in terms of computational cost) problems in NP.

Theorem 1 (Cook-Levin)**.**

1. *SAT is in NP.*
2. *For any A in NP, $A \leq_p SAT$.*

SAT is the problem of determining whether a Boolean expression has a satisfying variable assignment–can the expression be True for some input. What the Cook-Levin Theorem tells us is that every NP problem has a solution that is only polynomially more expensive than SAT, which leads us to the following corollary:

Corollary 1. *If $SAT \in P$, then $P = NP$.*

Proof.

1. $P \subseteq NP$, since a poly-time decider is a poly-time verifier that uses the empty certificate.
2. Assume that R is a poly-time decider for SAT. Let A be any problem in NP. By the Cook-Levin Theorem, we have that $A \leq_p SAT$. Let T be the reduction machine. Then the following is a poly-time decider for A:

   On input w:
   (a) Decide $R(T(w))$.

□

There are actually many problems which have these same properties as SAT, so we name this subclass of NP:

Definition 1. A language satisfying both conditions of the Cook-Levin theorem is *NP-Complete*. Alternately,

$$NP\text{-}Complete = \{L \mid L \in NP, \forall M \in NP, M \leq_p L\}$$

If a language satisfies only the second condition, we call it *NP-Hard*. Such languages may or may not be in NP.

A polynomial time solution for any NP-Complete (NPC) problem would show that $P = NP$, since it would imply a polynomial time solution for every NP problem via reduction. Conversely, to show that $P \neq NP$, we only need to show that any one NP problem has no polynomial-time solution.

The question now is how to show that a language is NP-Complete, since showing that every NP language, including ones no one has thought of yet, reduces to a particular language is not obvious. There are two ways. The hard way, which we will come back to, and reduction. If L is NPC and M is in NP, then if $L \leq_p M$, M is NPC. This follows by the transitivity of reduction. For any NP language A, we can reduce it to L in polynomial time, then reduce that to M in polynomial time. The composition of two polynomials is polynomial, so any NP language A reduces to M in polynomial time.

We will return to the proof of the Cook-Levin theorem, that SAT is NP-Complete, but first let us look at an example reduction proof. This requires already knowing that a language is NP-Complete, so for now we will assume that SAT is NPC. We will actually go one step further and assume the related problem 3-SAT is NPC.

Definition 2. To define 3-SAT, we recall some definitions from Boolean logic.

- A *Boolean variable* can be True or False.

- An *Assignment* is a choice of value for each variable.
- A *literal* is a Boolean variable or its negation.
- A *clause* is a disjunction (OR) of literals.
- A formula is in *CNF* (Conjunctive Normal Form) if it is a conjunction (AND) of clauses.
- A formula is in *3-CNF* if it is in CNF and every clause contains exactly 3 literals.
- A formula is *satisfiable* if there is a truth assignment that makes the formula True.

$$3\text{-}SAT = \{\langle F\rangle \mid F \text{ is a Boolean formula in 3-CNF and } F \text{ is satisfiable}\}$$

Exercise: Are the following formulae in 3-CNF? Are they satisfiable? If so, give a satisfying assignment. If not, explain why there can be no satisfying assignment.

$$(a \vee b \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \wedge (a \vee \overline{b} \vee c) \wedge (\overline{a} \vee \overline{b} \vee \overline{c})$$

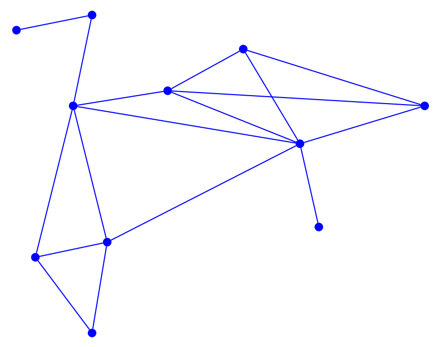
$$(a \vee b \vee c) \wedge (a \vee b \vee d) \wedge (\overline{a} \vee b \vee \overline{c}) \wedge (\overline{b} \vee c \vee d) \wedge (\overline{b} \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee \overline{b} \vee d)$$

Claim 1. *CLIQUE is NP-Complete, where*

$$CLIQUE = \{\langle G, k\rangle \mid G = (V, E) \textit{ is an undirected graph containing a clique of size } k\}$$

and a clique is a complete subgraph, $V' \subseteq V$ *is a clique if* $\forall u, v \in V', (u, v) \in E$.

Exercise: What is the largest clique in the following graph?



Proof. To show that $CLIQUE$ is hard, we show that if we could solve it easily, we could solve a known hard problem easily. To do this, we show that $3\text{-}SAT \leq_p CLIQUE$.

Suppose we have a Boolean expression in CNF, with k clauses: $(a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$. Note that the various a_i's, b_i's, and c_i's may be equal or negations of each other. We convert this expression to a graph s.t. the graph has a k-clique iff the expression is satisfiable.

Construct $G = (V, E)$, where

- $V = \{a_1, b_1, c_1, a_2, b_2, c_2, \ldots, a_k, b_k, c_k\}$
- $E = (V \times V) \setminus \{(u, v) \mid u \text{ and } v \text{ are in the same clause or } u = \overline{v}\}$.

Exercise: Consider the expression $(v \vee y \vee \overline{x}) \wedge (\overline{x} \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y} \vee \overline{z})$ Draw the corresponding graph this construction specifies, and look for triangles. (We are looking for triangles because this expression has three clauses, and a triangle is a 3-clique)

Claim 2. *A Boolean expression has a satisfying assignment iff there is a k-clique in the graph we construct.*

Proof. (⇒) Since there is a satisfying assignment, every clause must be True, and a clause is only True when at least one of its literals is True. Let $d_1, d_2, \ldots, d_k$ be True literals, with d_i from clause i. The vertices $\{d_1, \ldots, d_k\}$ form a k-clique, since no two are in the same clause or are each other's negation (or they could not both be True).

(⇐) If we have a k-clique in the graph, set the literals corresponding to the vertices of the clique to be True. This is a satisfying assignment, since it cannot contain two vertices from the same clause, and thus makes a literal in each of the k clauses True. It is a valid assignment as it cannot contain two literals which are each other's negation.

□

To finish the proof, we need to argue that the reduction is polynomial. G is polynomial in the input size, since the number of vertices is the number of literals in the input and the number of edges is at most that squared. The calculation for each of these elements is polynomial, so we can generate G in polynomial time. □

0.1 More NP-Complete Problems

There are many problems we know are NP-Complete. Here are a few.

Exercise: For each of the following, give an NTM or verifier to prove it is in NP, and think about from what we should reduce to show that it is NP-Hard.

- Hamiltonian Paths
- Vertex Cover: Given a graph, find a set of vertices such that every edge in the graph has at least one endpoint in the set. For a decision problem, is there a vertex cover containing k vertices?
- Subset Sum: Given a set of numbers and a target value, is there a subset with that sum?