NP Completeness

Richard Kelley

Spring 2026

1 Reductions

We want to formalize the statement:

> Problem X is "at least as hard as" problem Y.

One way to do this is via *reduction:*

> If we can solve X, then we can use that to solve Y.

(This assumes we can somehow convert between disparate problems, which happily turns out to be the case.)

Suppose we have a "black box" that can solve an instance of X in one step. Then we ask

> Can arbitrary instances of Y be solved using a polynomial number of "standard" steps plus a polynomial number of calls to the black box that solves X?

If yes:

Definition 1. $Y \leq_P X$. We read this as "Y is polynomial-time reducible to X" or "X is at least as hard as Y with respect to polynomial time."

This is a bit like having a "co-processor" for problem X (like a GPU).

Recall our definition that P is the set of problems X for which there exists an algorithm A with polynomial running time that solves instances of X.

Theorem 1. *Suppose* $Y \leq_P X$. *If* $X \in P$ *then* $Y \in P$.

Proof. If $X \in P$ we have an algorithm to solve instance of X in polynomial time. Because $Y \leq_P X$, we have in the definition of $\leq_P$ replace calls to the black box with the algorithm that solves X. Then we can solve Y with a polynomial number of standard steps plus a polynomial number of calls to a polynomial-time algorithm, so $Y \in P$. □

This is logically equivalent to:

Theorem 2. *Suppose* $Y \leq_P X$. *If* $Y \notin P$ *then* $X \notin P$.

In other words, if we know that Y is "hard" and that $Y \leq_P X$, then we know that X must be "hard" as well. Otherwise we could use X to easily solve Y.

We can use this to establish relationships of *relative difficulty* even if we don't know that either X or Y is actually "hard."

We also have this nice property:

Theorem 3. *If $Z \leq_P Y$ and $Y \leq_P X$, then $Z \leq_P X$.*

Proof. Each of $Z \leq_P Y$ and $Y \leq_P X$ implies the existance of an algorithm for the reduction. Run the algorithm for Z. In each point of calling to the black box for Y, instead use the method that solve Y by solving X. This solves Z using a polynomial number of steps plus a polynomial number of calls to a black box that solves instance of X, so $Z \leq_P X$. □

2 The Class NP

In contrast to the definition of P, our formal definition of NP is a bit more involved:

Definition 2. An algorithm B is an *efficient certifier* for problem X if

1. B is a polynomial-time algorithm of two inputs s and t.
2. There exists a polynomial p such that for every string s, $s \in X$ if and only iff there is a string t such that $|t| \leq p(|s|)$ and $B(s,t) = \texttt{yes}$.

It helps to think of t as a "certificate" proving that $s \in X$. For standard NP-completeness, this is always a "candidate" solution.

Then we define

Definition 3. NP is the set of all problems that have an efficient certifier.

We can immediately establish part of the relationship between P and NP:

Theorem 4. *$P \subseteq NP$.*

Proof. Let $X \in P$. Then there is an algorithm A that solves X in polynomial time. Our goal is to find an efficient certifier B for X. Given pair of strings (s,t), have B return $A(s)$. This

1. Has polynomial running time.
2. Has $\forall s \in X$ and all t such that $|t| \leq p(|s|)$, $B(s,t) = \texttt{yes}$, and $\forall s \notin X$ and all t with $|t| \leq p(|s|)$, $B(s,t) = \texttt{no}$.

These establish that $X \in NP$, so $P \subseteq NP$. □

The other direction of inclusion is famously open: Is $NP \subseteq P$? The general belief is that the answer is no: there must be some problem $X \in NP$ with $X \not{in} P$. This matches our intuition that finding solutions to problems seems harder than checking candidate solutions. There has also been enormous effort to find efficient (polynomial-time) algorithms for problems in NP. All such efforts have failed.

3 NP-Completeness

Within NP is another class that seems like it may help us to resolve the relationship between P and NP:

Definition 4. A problem X is *NP-complete* if

1. $X \in NP$, and
2. $\forall Y \in NP$, $Y \leq_P X$.

This gives us a direct line to potentially solving P vs NP:

Theorem 5. *Suppose that X is NP-complete. Then X is solvable in polynomial time if and only if $P = NP$.*

Proof. Suppose $P = NP$. Then since $X \in NP = P$, X can be solved in polynomial time. On the other hand, if X can be solved in polynomial time, let Y be any other problem in NP. Then $Y \leq_p X$. This means that Y can be solved in polynomial time. So $NP \subseteq P$, and thus $P = NP$. □

Moreover, once we know that a particular problem Y is NP-complete, that property extends to every problem we can reduce Y to:

Theorem 6. *If Y is NP-complete, and $X \in NP$ with $Y \leq_P X$, then X is NP-complete.*

Proof. $X \in NP$ by assumption. Let $Z \in NP$. Then since Y is NP-complete we have $Z \leq_P Y$. By assumption $Y \leq_P X$. So by transitivity $Z \leq_P X$. So X is NP-complete. □

So if you want to show that your problem X is NP-complete, you just have to find some known NP-complete problem Y and reduce that to your X.

This seems useful, but we still have to answer the question: do NP-complete problems even exist? It's not obvious that they do, and it seems like there are several ways they might not:

- There may be problems X' and X'' that are *incomparable:* Neither $X' \leq_P X''$ nor $X'' \leq_P X'$ is true.
- There may be some infinite sequence of problems $X_1, X_2, \ldots$ such that

$$X_1 \leq_P X_2 \leq_P X_3 \leq_P \cdots .$$

We have to find a first NP-complete problem.

4 The Cook-Levin Theorem

Recall the operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not).

Definition 5. A *circuit* K is a labeled directed acyclic graph with the following properties:

- The *sources* in K are labeled with 0, 1, or the name of a *variable.* The variables are the *inputs* to the circuit.

- Every other node is labeled with $\wedge$, $\vee$ or $\neg$.
  - Nodes labeled with $\wedge$ and $\vee$ have 2 incoming edges.
  - Nodes labeled with $\neg$ have 1 incoming edge.
- There is a single node with no outgoing edges, representing the *output* of K.

An example:

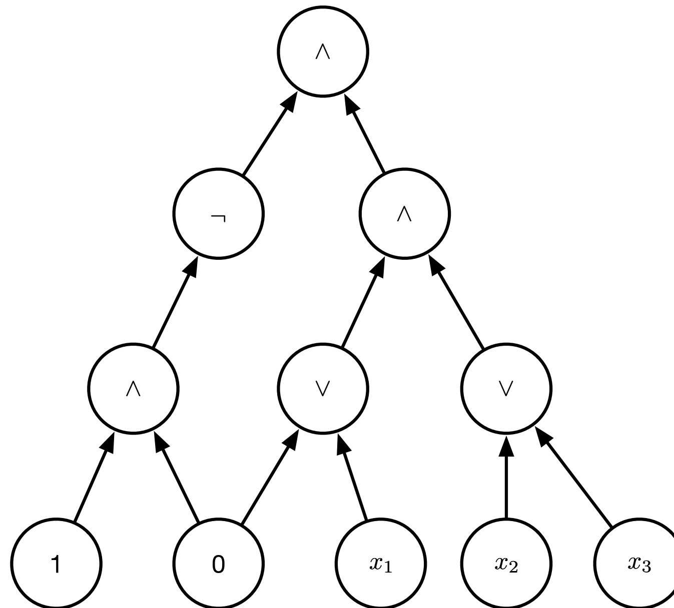

Figure 1: An example circuit with inputs x_1, x_2, x_3. You should verify that the assignment $x_1 = 1, x_2 = 0, x_3 = 1$ causes K to have the output 1.

Once we have the notion of a circuit, we can define:

Definition 6. We say that a circuit K is *satisfiable* if we can find inputs to our circuit that make the output 1. An assignment that results in 1 is called a *satisfying assignment.*

This gives us our first NP-complete problem:

Definition 7. (**Circuit-SAT**) Given a circuit K, decide if K has a satisfying assignment.

To show that Circuit-SAT is NP-complete, we need to show that Circuit-SAT $\in NP$, and that for any problem $X \in NP$, $X \leq_P$ Circuit-SAT. When you think about what a circuit is, this maybe doesn't seem so daunting: a circuit is basically made up of a bunch of logic gates connected by wires, which is exactly the sort of thing that a general-purpose computer is made out of. Following this intuition, we can sketch the proof for the following theorem, famous enough to be named:

Theorem 7. *(**Cook-Levin**) Circuit-SAT is NP-Complete.*

Proof. Again, this is only the sketch:

1. Any algorithm that takes n bits as input and outputs either 0 or 1 can be encoded as a circuit. We can convert an algorithm A for a decision problem into a circuit by having the circuit output 1 exactly when the algorithm outputs `yes`.

2. If A takes polynomial time, you can show that the corresponding circuit has polynomial size.

3. Given an arbitrary problem $X \in NP$, we want to show that $X \leq_P$ Circuit-SAT. What do we know about X? We know that X must have an efficient verifier B. To determine if $s \in X$ for input s with $|s| = n$, we need to answer the question: is there a t with $|t| = p(|s|)$ such that $B(s,t) =$ yes? We'll try to use Circuit-SAT to solve this particular membership problem.

4. Remember we have a black box for Circuit-SAT, and the desired reduction exists if we can solve X with a polynomial number of steps plus a polynomial number of calls to the Circuit-SAT black box.

5. Think of B as an algorithm that takes $n + p(n)$ bits of input and outputs a 0 or 1.

6. Convert this to a polynomial-sized circuit K on $n + p(n)$ bits:

   - n sources for K hard-coded with the bits representing s, and
   - $p(n)$ sources with bits of t. These are the inputs to K.

7. Note that $s \in X$ if and only if we can set input bits to K so that K produces an output of 1. That is, if and only if K is satisfiable.

This gives us a polynomial-time reduction from X to Circuit-SAT. □

(Aside: At this point it's worth noting that there are at least 2 notions of reduction that this presentation is not being careful to distinguish: In a *many-one reduction* we transform an instance of X to an instance of Y and then solve Y. In a *turing reduction* we are able to make arbitrary calls to our "black box." We defined reduction in terms of a black box but only use that box to give us a many-one reduction in the proof. This is good because the standard definition of NP-completeness uses many-one reduction.)

5 Example Reductions

We are going to end by considering a few problems in NP, and show that they are NP-complete. We won't go into all of the details, but this will give you a sense of what goes on in this kind of work.

5.1 3-SAT

Instead of working with Circuit-SAT we're going to consider a related problem. First some definitions:

Definition 8. A *literal* is either a boolean variable or the negation of a boolean variable. We call variables *positive literals* and negated variables *negative literals*.

We want to consider logical formulas that are very structured. There are many ways to do this, called *normal forms*, but the one we'll focus on uses conjunction and disjunction in a certain way:

Definition 9. A *clause* is a disjunction of literals.

Definition 10. A formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses.

As an example, consider

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3).$$

This is a CNF formula with 2 clauses each having 3 variables. Note that if we set $x_1 = x_2 = x_3 = 1$ here then we can make $\phi = 1$. As in Circuit-SAT, an assignment of values to variables that makes a formula 1 is called a *satisfying assignment*.

The *satisfiability problem* is, given a formula ϕ, does ϕ have a satisfying assignment?

It turns out this is not enough structure, and in any case any formula in propositional logic can be converted to a formula in conjunctive normal form using simple logical equivalences. So we focus on variants of the problem that are in CNF form.

Definition 11. In the k-SAT problem, the formula ϕ is a CNF formula in which each clause has k literals. The problem is to determine if ϕ has a satisfying assignment.

There are two cases of particular interest. The first is because it is easy:

Theorem 8. *2-SAT* $\in P$.

And the second because it is hard:

Theorem 9. *3-SAT is NP-complete.*

Proof. Circuit-SAT $\leq_P$ 3-SAT. We won't show it, but it is possible to reduce from Circuit-SAT to 3-SAT by converting the logic gates of the circuit to CNF clauses in a well-defined way. □

5.2 Subset Sum

This is a very different looking problem:

Definition 12. An instance of SubsetSum consists of numbers $a_1, \ldots, a_n$ and a target T and asks if there is a subset of the a_is that adds up to T.

Theorem 10. *SubsetSum is NP-complete.*

Proof. First, SubsetSum $\in NP$. Intuitively, given an instance of SubsetSum and a candidate solution, we can add the numbers in linear time and verify that they add up to the target.

The harder part is to show that all problems in NP reduce to SubsetSum. We'll use the NP-completeness of 3-SAT. Given an instance of 3-SAT, we need to construct an instance of SubsetSum. If our 3-SAT instance has n variables and m clauses, we'll construct numbers a_k as follows: Each a_k will have $n+m$ decimal digits. The first n digits will correspond to the variables of our problem and the last m digits will correspond to the clauses. For each variable x_l in the 3-SAT problem, we'll create two numbers: v_l^t and v_l^f. Here's how we define them:

- v_l^t has a 1 in the l-th digit and 0 in the other of the first n digits. For the last m digits it has a 1 in place corresponding to a clause where x_l appears and 0 elsewhere.
- v_l^f has a 1 in the l-th digit and 0 in the other of the first n digits. For the last m digits it has a 1 in place corresponding to a clause where $\neg x_l$ appears and 0 elsewhere.

For each clause j we also add two *slack variables*, having a 1 in position $n + j$ and 2 in position $n + j$, with 0 elsewhere. The *target* for our problem will have the form n 1's followed by m 4's.

How does this encode an instance of 3-SAT? The 1's in the target force each boolean variable to get a value of either 0 or 1. The 4's force each clause to get at least one variable having a value that satisfies the clause: if no variable in the assignment causes a clause to be satisfied, then 4 is unreachable. If 1 variable causes the clause to be satisfied, we can add both slack variables to reach 4. If 2 variables cause the clause to be satisfied we can add the slack variable with 2, and so on. This is a polynomial-time construction showing that we can convert an instance of 3-SAT to an instance of SubsetSum, giving the required reduction and showing that SubsetSum is NP-Complete. □

To get more intuition for this construction, let's consider the example from above:

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3).$$

Let's build the corresponding SubsetSum problem instance. We have a 3-SAT instance with 3 variables and 2 clauses, so our SubsetSum problem will have 5 digit numbers. We'll build our numbers first:

v_1^t	(1,0,0,1,0) = 10010
v_1^f	(1,0,0,0,1) = 10001
v_2^t	(0,1,0,0,1) = 1001
v_2^f	(0,1,0,1,0) = 1010
v_3^t	(0,0,1,1,1) = 111
v_3^f	(0,0,1,0,0) = 100
$s_{1,1}$	(0,0,0,1,0) = 10
$s_{1,2}$	(0,0,0,2,0) = 20
$s_{2,1}$	(0,0,0,0,1) = 1
$s_{2,2}$	(0,0,0,0,2) = 2

This just leaves our target value, which has as digits three 1's followed by 2 4's: 11144.

This is our subset sum problem. A satisfying instance of the 3-SAT problem is $x_1 = x_2 = T$, and $x_3 = F$. This corresponds to the initial sum:

$$v_1^t + v_2^t + v_3^f = 11111,$$

to which we can add slack variables to get our target 11144:

$$v_1^t + v_2^t + v_3^f + s_{1,1} + s_{1,2} + s_{2,1} + s_{2,2} = 11144.$$

5.3 Knapsack

We've already seen Knapsack. Recall that an instance of knapsack has a set of n items with weights w_i and values v_i, and a knapsack *capacity* C. In its optimization form, the goal of Knapsack is to find the subset S of items that maximizes $\sum_{i \in S} v_i$ and has $\sum_{i \in S} w_i \leq C$. The decision problem version add a parameter V and asks if there is a feasible subset S with $\sum_{i \in S} v_i \geq V$.

As promised earlier in the semester, Knapsack is hard:

Theorem 11. *Knapsack is NP-complete.*

Proof. Reduction from SubsetSum. Given an instance of SubsetSum with numbers $a_1, \ldots, a_n$ and target T, construct an instance of Knapsack with n items, where $v_i = w_i = a_i$, and $C = V = T$. Then in the knapsack problem a solution S will have both $\sum_{i \in S} v_i = \sum_{i \in S} a_i \geq T$ and $\sum_{i \in S} w_i = \sum_{i \in S} a_i \leq T$. This means that $\sum_{i \in S} a_i = T$, giving us a subset with the desired sum. □