NP and Latin Squares | Instructor: Padraic Bartlett

Lecture 2: NP-Completeness

Week 4 | *Mathcamp 2014*

In our last class, we introduced the complexity classes P and NP. To motivate why we grouped all of NP together (instead of, say, breaking it into subclasses like NP-with-2^n-runtime, NP-with-$n!$-runtime, ...), we started showing how solutions to some problems in NP can "create" solutions for other problems in NP! For example, we showed that any algorithm that could solve instances of the k-independent-set problem could also solve instances of 3SAT: we did this via a transformation process that turned any 3SAT formula into a graph, in such a way that k-independent sets in this graph corresponded to ways to satisfy our 3SAT formula.

In this class, we will make this notion of "reduction" formal and study a few examples. After doing this, we will introduce the concept of **NP**-hardness — a problem that every task in NP can be reduced to — and prove the surprising Cook-Levin theorem, which states that there is an NP-hard problem contained in NP!

We start with the notion of reduction, which we informally discussed yesterday:

1 Reductions

Definition. Take any two problems L, M. We say that L can be **polynomial-time reduced** to M if any algorithm that solves M can be turned into an algorithm to solve L, with an increase in runtime that is at most polynomial. Formally: given any algorithm A that solves instances x_L of L of size $|x_L| = n$ in time $t(n)$, there is a polynomial p and an algorithm B that solves any instance x_M of M of size $|x_M| = n$ in time $p(t(n))$.

Intuitively, this is a way of saying that the problem L is "essentially" M at its core: if you know how to solve M, then you mostly know how to solve L, up to some polynomial-time details.

We will simply use the word reduction in place of polynomial-time reduction throughout this class for brevity's sake.

We closed our last class by reducing 3SAT to k-independent set. We perform another reduction here:

Theorem. Consider the problem of determining whether a graph G on n vertices has a 3-coloring; that is, a way to assign the three colors R, G, B to the vertices of G, so that no edge is monochromatic (i.e. has both endpoints colored the same color.) Call this task 3-colorability. This task is clearly in NP, with a "proof" that a graph is 3-colorable given by a concrete 3-coloring of said graph.

3SAT is reducible to 3-colorability.

Proof. As before, we will describe a process that takes in any k-clause formula in 3-conjunctive normal form (i.e. the form that boolean formulas considered in 3SAT take), and transforms it into a graph on at most linear (k) vertices (specifically, $\leqslant 11k + 3$ vertices.)

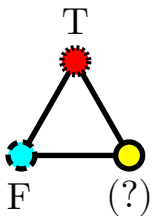
This transformation will take us polynomial time to create; furthermore, this graph will have a 3-coloring if and only if the corresponding boolean formula is satisfiable. Consequently, we can use any algorithm that can solve the 3-colorability process in time $t(n)$ to get an algorithm that solves the 3SAT in time poly$(t(n))$.

We do this as follows. Given our Boolean formula

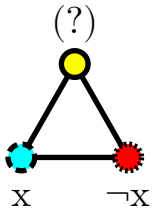
$$(l_{11} \vee l_{12} \vee l_{13}) \wedge (l_{21} \vee l_{22} \vee l_{23}) \wedge \ldots \wedge (l_{k1} \vee l_{k2} \vee l_{k3}),$$

where each l_{ij} is equal to some variable x or its negation $\neg x$, construct a graph G as follows:

1. The "triforce of truth:" Create a triangle with three vertices. Label one vertex T, a second F, and the third (?). If we can 3-color our graph, the three vertices of this triangle must all receive different colors; we will associate the color assigned to T with "true," the color assigned to F with "false," and sincerely hope that (?) doesn't come up anywhere else.

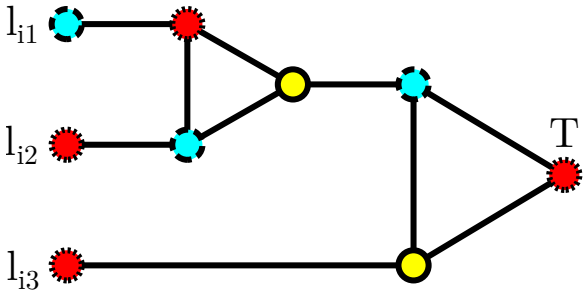


2. For each variable x that occurs in a literal in our formula, create two vertices labeled $x, \neg x$, and connect these two vertices to each other as well as to the (?) vertex we defined in step 1.



   Notice that this gluing insures that none of these "variable-vertices" can ever be colored with the same color as the (?) vertex; moreover, we never have both $x, \neg x$ assigned to the same truth value.

3. Finally, for each clause $(l_{i1} \vee l_{i2} \vee l_{i3})$, find the three vertices corresponding to those literals that we created in step 2, and by adding in 5 new vertices + 10 new edges, create the following graph linking these vertices to the T vertex from step 1.



Observations you can make that will take you 5 minutes of casework: it is impossible for all three of l_{i1}, l_{i2}, l_{i3} to not have the same color as the T vertex. In other words, in each clause, at least one variable must be colored with the "true" color. Furthermore, if there is at least one literal with the "true" color, there is always a way to color the five intermediate vertices above without creating any conflicts.

By construction, colorings of this graph correspond precisely to ways to assign truth values to our formula (via 2) that satisfy our formula (via 3). Therefore, if we have an algorithm that can determine whether a graph is 3-colorable, we can use this construction to adapt this algorithm to solve instances of 3SAT! In other words, we have reduced the problem of 3SAT to that of 3-colorability, as desired. □

In the following section, we start to explore the concept of reducibility in more depth:

2 NP-Completeness and Cook-Levin

Definition. A problem L is called **NP-hard** if any problem in NP can be reduced to L. A problem L is called **NP-complete** if it is both NP-hard and in NP.

The existence of a NP-complete problem should be surprising at first; the class NP is staggeringly huge, so it seems odd that (in a sense) every problem in NP can be reduced down to one sort of "all-problem," that contains all of the other problems.

With the right perspective, though, it turns out that this is actually remarkably trivial. First: to do any reasoning about computation, we need to fix a model of computation itself! There are many such models we could pick out: Turing machines, Lambda calculus, recursive functions, ... In practice, it doesn't matter too much which model you pick; if the Church-Turing thesis holds, most any notion you pick to work in will give you equivalently powerful notions of what your algorithms can "do."

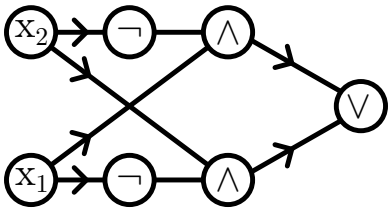
For this class, the formal notion of computability we will work with is perhaps one of the most intuitive that exists, if you're trying to capture what a computer can and cannot do: circuits! Consider the following definition:

Definition. A **circuit** is a directed acyclic graph with the following kinds of labeled vertices:

1. Input vertices: vertices with indegree 0, labeled with distinct variable symbols (i.e. $x_1, x_2, \ldots$)
2. AND vertices: vertices with indegree 2 and outdegree 1, all labeled with the same $\wedge$ symbol.
3. OR vertices: vertices with indegree 2 and outdegree 1, all labeled with the same $\vee$ symbol.
4. NOT vertices: vertices with indegree 1 and outdegree 1, all labeled with the same $\neg$ symbol.
5. The output vertex: a single vertex with indegree 1 and outdegree 0.

The size of any circuit is simply the number of vertices in this circuit.

Here is a sample circuit of size 8:



Any circuit has a corresponding boolean expression, and thus corresponds to some function $\{0,1\}^n \to \{0,1\}$. For example, the circuit above corresponds to the boolean function XOR, that takes in two variables and outputs true if they disagree, and false otherwise.

In general, any boolean function from $\{0,1\}^n \to \{0,1\}$ can be expressed as some corresponding circuit; on the homework you're asked to show that this is true! (In fact, any boolean function from $\{0,1\}^n \to \{0,1\}$ can be expressed as a circuit with at most $O(2^n/n)$ vertices! Moreover, almost any boolean function needs that many vertices to be so expressed. By way of contrast, we have yet to actually **find** any boolean functions that need anything more than poly(n) vertices to be expressed, which is deeply strange / one of the more interesting open questions in this field!)

This is going to be our model of computation! This is a little odd, though, in that circuits all take in inputs of fixed length. I.e. the XOR gate above takes in only two inputs; it's not obvious how we'd extend it to more inputs by just looking at it!

Consequently, what we want isn't simply circuits: it's **circuit families**.

Definition. A **circuit family** is a collection $\{C_n\}_{n\in\mathbb{N}}$ of circuits, such that each C_n has n input vertices.

Definition. An **algorithm** A is simply some family of circuits $\{C_n\}_{n\in\mathbb{N}}$. We interpret A applied to an input $(x_1, \ldots x_n) \in \{0,1\}^n$ as just C_n with these x_i's as inputs.

With this notion of algorithm established, the Cook-Levin theorem is actually fairly trivial:

Theorem. (Cook-Levin) There is an NP-complete problem. In particular, consider Circuit-SAT, which is the task of determining whether a given circuit of size n can be made to output true; this problem is NP-complete.

Proof. Circuit-SAT is equivalent to SAT (as we're just linking variables via AND, OR and NOT's), and thus is in NP. So it suffices to prove that Circuit-SAT is NP-hard: i.e. that we can reduce any problem in NP to Circuit-SAT.

To do this, let's revisit what it means for a problem to be in NP. Originally, we made the following definition:

Definition. A problem L is in NP if there is a polynomial-time algorithm $A(x,u)$, that takes in instances x of L along with claimed "proofs" u that x holds, and outputs T if that "proof" demonstrates that x has a yes answer in L. In other words,

$$x \text{ has a yes answer in } L \Leftrightarrow \exists u, A(x,u) = T.$$

Moreover, notice that there is some polynomial p such that the length of u is less than $p(\text{length}(x))$. This is because if L is a polynomial-time algorithm that uses u, its runtime is an upper bound on the total length of whatever proof L needs to read to perform its work.

If we change the language here to that of circuits, we have the following:

Definition. A problem L is in NP if there is a circuit family $\{C_n\}_{n\in\mathbb{N}}$ that takes in instances x of L along with claimed "proofs" u that x holds, and outputs T if that "proof" demonstrates that x has a yes answer in L. In other words,

$$(x \text{ has a yes answer in } L) \Leftrightarrow (\exists u \text{ with length}(u) < p(\text{length}(x) \text{ such that } C_n(x, u) = T).$$

Given any NP-complete problem L, instance x of L, and corresponding polynomial-time proof-checking circuit family $\{C_n\}$, consider the circuit family $\{C_n\big|_x\}_{n\in\mathbb{N}}$, formed by taking the circuits C_n and "fixing" their instance inputs to the instance x. These are now circuits that take in possible proofs u of the instance x and output either true or false depending on whether or not u actually corresponds to a proof.

Also, notice that because the circuits $\{C_n\}_{n\in\mathbb{N}}$ can be evaluated in polynomial time, they must have polynomial size as well (as otherwise it would take too long to evaluate these circuits!)

By our comments above, there is some polynomial $p(\text{length}(x))$ such that if any such u exists, it has length $\leqslant p(\text{length}(x))$.

Therefore, to determine whether the instance x of our problem L evaluates to true, it suffices to simply solve the Circuit-SAT problem for all of the circuits $\{C_n\big|_x\}_{n=1}^{p(\text{length}(x))}$. There are polynomially many of these circuits; consequently, any algorithm that can solve an instance of Circuit-SAT of length n in time $t(n)$ can determine whether the instance x of our problem L evaluates to true in time $q(t(n))$, for some polynomial q.

In other words, we've reduced the problem L to Circuit-SAT, for any problem L in NP. □

Cool! So there is a NP-complete problem. If you do the homework (where you're asked to show that SAT reduces to 3SAT,) then we've actually found several NP-complete problems: Circuit-SAT, SAT, 3SAT, k-independent set, and 3-colorability! In tomorrow's class, we'll begin to examine the NP-complete problems this class really cares about – those involving Latin squares!