NP Completeness

Essential facts about **NP**-completeness:

- Any **NP**-complete problem can be solved by a simple, but exponentially slow algorithm.
- We don't have polynomial-time solutions to any **NP**-complete problem.
- We can prove that either *all* **NP**-complete solutions have polynomial-time solutions, or *none* of them do.
- It is generally believed that none of them do. But proving this would require solving the **P versus NP problem**, one of the best known unsolved problems in mathematics, much less theoretical computer science.

NP-completeness is a property of **decision problems**. It can be used to prove that other types of problems are **NP-hard**, which means that (unless $\mathrm{P} = \mathrm{NP}$) they have no polynomial-time solutions. These include **search**, **optimization**, and **approximation** problems.

Definition: The class **P** is the set of decision problems for which there exists an algorithm solving them in $O(n^k)$ time for some constant k.

Definition: A formal language A is in **NP** if there exists another language B in **P**, such that for any string x, x is in A iff there exists a string y, with $|y| = |x|^{O(1)}$, such that (x, y) is in B.

An equivalent, more algorithmic definition of **NP** is as follows. An **NP-procedure** consists of a **guess phase** and a **verification phase**. Given an input x, the guess phase chooses an arbitrary string y such that $|y| = |x|^{O(1)}$. The verification phase is an algorithm that takes both x and y as input and returns a bit. We say that x is in the language of the **NP**-procedure iff *it is possible* for the procedure to guess a y making the verification phase output "true".

There *is* an obvious deterministic decision procedure for any **NP** language – simply cycle through all possible strings y and see whether the verification procedure accepts (x, y). The problem is that there are $2^{n^{O(1)}}$ possible y's, of course. Our question is whether there is a better way to decide membership in the **NP** language, one that might run in polynomial time.

Proving a language to be in **NP** is generally simple. We need to define a string to be guessed, and define a poly-time verification procedure that accepts the input and a guess iff the guess *proves* that the input is in the language.

We need to determine relationships among the languages in **NP**, using the notion of **reduction**. We want to show that *if* problem B is in **P**, then so is problem A. One way to do this is to describe an algorithm for A would run in polynomial time if it were allowed to make calls to a hypothetical poly-time algorithm to decide membership in B. This is called a **Cook reduction**. For defining **NP**-completeness, though, we will need a slightly different notion of reduction.

Let A and B be two formal languages, possibly over different alphabets. A **Karp reduction** from A to B is a poly-time computable function f such that for any string x, $x \in A$ if and only if $f(x) \in B$. If such a reduction exists we say that A is **poly-time reducible** to B and write "$A \leq_p B$". We also sometimes read this as "A is no harder than B".

Two languages A and B are said to be **p-equivalent** if both $A \leq_p B$ and $B \leq_p A$. The relation $\leq_p$ is a partial order on the equivalence classes. We are interested in the maximal equivalence class in **NP**:

Definition: A language B is **NP**-complete if (1) it is in **NP**, and (2) for any language A in **NP**, $A \leq_p B$.

Thus the **NP**-complete languages, if they exist, are the *hardest* languages in **NP**. It should be easy to see that they form an equivalence class, and that if any **NP**-complete language is also in **P**, it follows that $\mathbf{P} = \mathbf{NP}$.

If a language is **NP**-complete, then, we have strong evidence that it is not in **P**. We can use the **NP**-completeness of a language to talk about non-decision problems, even though by definition these cannot be **NP**-complete.

Definition: A problem X (with boolean output or otherwise) is said to be **NP-hard** if there is a Cook reduction from X to some **NP**-complete problem B. That is, there is a poly-time algorithm, with access to a hypothetical poly-time algorithm for X, that decides B.

It should be clear that if X is **NP**-hard and there actually *is* a poly-time algorithm for X, then $\mathbf{P} = \mathbf{NP}$.

Our general methodology will be to develop a library of **NP**-complete problems. Once we have one problem, there is a straightforward way to get more:

Lemma: Let B be an **NP**-complete language and A be a language. If we prove:

- $A \in \mathbf{NP}$
- $B \leq_p A$

then we may conclude that A is **NP**-complete.

But how do we start this process? In CMPSCI 601, we prove the **Cook-Levin Theorem**, that the language SAT is **NP**-complete. Last time, we gave an unconditional proof that a particular **generic NP language** is **NP**-complete. We'll still need the Cook-Levin Theorem, because SAT is a much more convenient starting place to build up a library of **NP**-complete languages.

SAT and 3-SAT

Remember that a boolean formula is defined to be **satisfiable** if there is at least one setting of its input variables that makes it true. The language SAT is the set of boolean formulas that are satisfiable.

Recall that a boolean formula is in **conjunctive normal form (CNF)** if it is the AND of zero or more **clauses**, each of which is the OR of zero or more **literals**. A formula is in **3-CNF** if it is in CNF and has at most three literals in any clause. The language CNF-SAT is the set of CNF formulas that are satisfiable, and the language 3-SAT is the set of 3-CNF formulas that are satisfiable.

Note that 3-SAT is a subset of CNF-SAT, which is a subset of SAT. In general, if $A \subseteq B$, we can't be certain that $A \leq_p B$. Although the **identity function** maps elements of A to elements of B, we can't be sure that it doesn't map a non-element of A to an element of B. But here we know more – we can easily test a formula to see whether it is in CNF or 3-CNF. To reduce 3-SAT to SAT, for example, we map a formula φ to itself if it is in 3-CNF, and to 0 if it is not. A similar reduction works to show $A \leq_p B$ whenever A is such an **identifiable special case** of B – that is, when $A = B \cap C$ and $C \in \mathbf{P}$.

The Cook-Levin Theorem tells us that SAT is **NP**-complete, essentially by mapping an instance of the generic **NP** problem to a formula that says that a particular string is a witness for a particular **NP**-procedure on a particular input. (Recall that if A is an **NP** language defined so that $x \in A$ iff $\exists y : (x, y) \in B$, we call y variously a **witness**, **proof**, or **certificate** of x's membership in A.)

We'd like to show that CNF-SAT and 3-SAT are also **NP**-complete. It's clear that they are in **NP**, but the easy special case reduction does *not* suffice to show them **NP**-complete. We can reduce 3-SAT to SAT, but what we need is to reduce the *known* **NP**-complete language, SAT, to the language we want to show to be **NP**-complete, 3-SAT.

On HW#4 I'll have you work through the general reduction from SAT to 3-SAT. Here, I'll present the easier reduction from CNF-SAT to 3-SAT. (The proof of the Cook-Levin Theorem given in CMPSCI 601 actually shows directly that CNF-SAT is **NP**-complete.)

Let's now see how to reduce CNF-SAT to 3-SAT. We need a function f that takes a CNF formula φ, in CNF, and produces a new formula $f(\varphi)$ such that $f(\varphi)$ is in 3-CNF and the two formulas are either both satisfiable or both unsatisfiable. If we could make φ and $f(\varphi)$ equivalent, this would do, but there is no reason to think that an arbitrary CNF formula will even have a 3-CNF equivalent form. (Every formula can be translated into CNF, but not necessarily into 3-CNF.)

Instead we will make $f(\varphi)$ have a different meaning from φ, and even a different set of variables. We will add variables to $f(\varphi)$ in such a way that a satisfying setting of both old and new variables of $f(\varphi)$ will exist if and only if there is a satisfying setting of the old variables alone in φ. In fact the old variables will be set the same way in each formula.

Because φ is in CNF, we know that it is the AND of clauses, which we may name $(\ell_{11} \vee \ldots \vee \ell_{1k_1})$, $(\ell_{21} \vee \ldots \vee \ell_{2k_2})$,...$(\ell_{1m} \vee \ldots \vee \ell_{mk_m})$, where the ℓ's are each literals. For each of these clauses in φ, we will make one or more 3-CNF clauses in $f(\varphi)$, possibly including new variables, so that the one clause in φ will be satisfied iff *all* the corresponding clauses in $f(\varphi)$ are satisfied.

So let's consider a single clause $\ell_1 \vee \ldots \vee \ell_k$ in φ. If $k \leq 3$, we can simply copy the clause over to $f(\varphi)$, because it is already suitable for a CNF formula. What if $k = 4$? We can add one extra variable and make two clauses: $(\ell_1 \vee \ell_2 \vee x_1)$ and $(\neg x_1 \vee \ell_3 \vee \ell_4)$. It's not too hard to see that both of these clauses are satisfied iff at least one of the ℓ's is true. If ℓ_1 or ℓ_2 is true, we can afford to make x_1 false, and if ℓ_3 or ℓ_4 is true, we can make x_1 true.

The general construction for $k > 4$ is similar. We have $k - 2$ clauses and $k - 3$ new variables: The clauses are $(\ell_1 \vee \ell_2 \vee x_1)$, $(\neg x_1 \vee \ell_3 \vee x_2)$, $(\neg x_2 \vee \ell_4 \vee x_3)$, and so on until we reach $(\neg x_{k-4} \vee \ell_{k-2} \vee x_{k-3})$ and finally $(\neg x_{k-3} \vee \ell_{k-1} \vee \ell_k)$.

If we satisfy the original clause with some ℓ_i, this satisfies one of the new clauses, and we can satisfy the others by making all the x_i's before it true and all those after it false. Conversely, if we satisfy all the new clauses, we cannot have done it only with x_i's because there are more clauses than x_i's and each x_i only appears at most once as true and at most once as false, and so can satisfy at most one clause.

Since this reduction is easily computable in polynomial time, it shows that CNF-SAT $\leq_p$ 3-SAT, and thus (with the quoted result that CNF-SAT is **NP**-complete) that 3-SAT is **NP**-complete.

3-SAT is often the most convenient problem to reduce to something else, but other variants of SAT are also sometimes useful. One we'll use later is **not-all-equal-SAT** or **NAE-SAT**. Here the input is a formula in 3-CNF, but the formula is "satisfied" only if there is both a true literal and a false literal in each clause.

Let's prove that NAE-SAT is **NP**-complete. Is it in **NP**? Yes, if we guess a satisfying assignment it is easy (in linear time) to check the input formula and verify that there is a true literal and a false literal in each clause. So we need to reduce a known **NP**-complete problem to NAE-SAT – we'll choose 3-SAT itself. Again we'll transform each old clause into the AND of some new clauses, in this case three of them.

Given the clause $\ell_1 \vee \ell_2 \vee \ell_3$, we introduce two new variables x and y that appear only in the new clauses for this clause, and a single new variable α that appears several times. The three new clauses are

$$(\ell_1 \vee \ell_2 \vee x) \wedge (\neg x \vee \ell_3 \vee y) \wedge (x \vee y \vee \alpha).$$

We must show that the three new clauses are jointly NAE-satisfiable iff the original clause is satisfiable in the ordinary way. First, we assume that the old clause is satisfied and show that we can choose values for x, y, and α to NAE-satisfy the new clauses. We make α true (for all the clauses in the formula) and consider the seven possibilities for the values of ℓ_1, ℓ_2, and ℓ_3. In each of the seven cases, we can set x and y, not both true, to NAE-satisfy the three new clauses – we'll check this on the board.

Now assume that the three new clauses are NAE-satisfied – we will show that at least one of the ℓ_i's is true. First assume that α is true, because if the new formula is NAE-satisfied with α false we can just negate every variable in the formula and get a setting that NAE-satisfies all the clauses but has α true.

If α is true, then either x or y must be false. If x is false, then either ℓ_1 or ℓ_2 must be true. If y is false and x is true, then ℓ_3 must be true. So one of the three ℓ's must be true, and the original clause is satisfied in the ordinary way.

CLIQUE and VERTEX-COVER

So far we've seen that several problems in logic are **NP**-complete. In fact there are **NP**-complete problems in a huge array of domains – we'll next look at some problems in **graph theory**, similar to some problems we've already solved in polynomial time.

Let G be an undirected graph. A **clique** in G is a set A of vertices such that all possible edges between elements of A exist in G. Any vertex forms a clique of size 1, the endpoints of any edge form a clique of size 2, and any triangle is a clique of size 3.

The language CLIQUE is the set of pairs (G, k) such that G is an undirected graph that contains some clique of size k. It should be clear that CLIQUE is in the class **NP**. Our **NP**-procedure guesses an arbitrary set A of vertices (by guessing a bitvector of length n). Then the verification phase checks that A has size exactly k and that there is an edge between every pair of vertices in A.

We'll prove CLIQUE to be **NP**-complete by reducing 3-SAT to it. Recall that this means defining a function from 3-CNF formulas to graph-integer pairs, such that satisfiable formulas are mapped to pairs (G, k) such that G has a k-clique and unsatisfiable formulas are mapped to pairs where G does not have a k-clique.

The essential element of any reduction is a correspondence between the witnesses of the two **NP**-problems. The 3-CNF formula is satisfied or not satisfied by a bitvector of length n, and the possible cliques are also denoted by bitvectors. We want to arrange the graph so that a satisfying instance corresponds to a k-clique and vice versa, and we get to pick k for our convenience.

Here's the construction. We have a node for each *appearance of a literal in the formula*. So if there are m clauses, each with k_i literals, the number of vertices in the graph is the sum from 1 to m of k_i. Now we need edges. We place an edge between nodes x and y if they refer to literals that *occur in different clauses* and are *not in conflict* (aren't negations of one another). We set k to be m, the number of clauses.

Nodes: Appearances of literals in clauses

Edges: Pairs of nodes that are in different clauses and not in conflict.

We claim that the 3-CNF formula is satisfiable iff there is an m-clique in the graph. First assume that there is a satisfying assignment, which means that there is at least one literal in each clause that is set true. Fix a set containing exactly one true literal in each clause. The m nodes corresponding to these literals must form a clique. No two of them are in the same clause, and no two of them can be in conflict, so all possible edges between then exist.

Conversely, suppose that we have an m-clique in the graph. The m nodes must occur in m different clauses, since edges only connect nodes in different clauses. Because the m nodes also contain no conflicts, we can construct a setting of the variables consistent with all those literals. (We may have to arbitrarily set variables that don't occur in the set either as true or as false.) This setting makes at least one literal in each clause true, so it satisfies the formula.

With some easy reductions, we can use CLIQUE to prove some similar problems to be **NP**-complete.

Again let G be an undirected graph. A set of vertices A is an **independent set** if there are *no* edges in G between vertices in A. The language IND-SET is the set of all pairs (G, k) such that there exists an independent set of size k in G. Clearly IND-SET is in **NP**, because we can guess the set A, verify its size, and verify that it contains no edges.

We prove IND-SET to be **NP**-complete by reducing CLIQUE to it, now that we know CLIQUE to be **NP**-complete. This reduction will be a function that takes pairs (G, k) to pairs (H, ℓ) such that G has a clique of size k iff H has an independent set of size ℓ.

But this is easy! The problems are very similar, so much so that we can give H the same set of vertices as G and arrange that a set A is a clique in G iff it is an independent set in H. How do we do this? We want to map sets with all the edges to sets with none of the edges, so we just make H the **complement** of G – the graph that has an edge (x, y) exactly when that edge is *not* an edge of G. Then the function taking (G, k) to (H, k) is the desired reduction.

Another similar problem is VERTEX-COVER. A set A of nodes of an undirected graph G is a **vertex cover** if every edge of G has at least one endpoint in A. The language VERTEX-COVER is the set of pairs (G, k) such that G has a vertex cover of size k. As before, it is clear that VERTEX-COVER is in **NP**.

The Adler notes give a direct reduction from 3-SAT to VERTEX-COVER, which is very similar to the reduction from 3-SAT to CLIQUE. But we don't need to use this reduction, because it's very easy to reduce CLIQUE or IND-SET to VERTEX-COVER and thus use our previous work to prove VERTEX-COVER to be **NP**-complete.

If $A \subseteq V$ is a vertex cover in G, look at the set $V \setminus A$ of nodes *not* in A. There are no edges between these nodes, since every edge has at least one endpoint in A. So $V \setminus A$ is an independent set – in fact it is an independent set *iff* A is a vertex cover.

So G has a vertex cover of size k iff it has an independent set of size $n - k$. Thus the function taking (G, k) to $(G, n - k)$ is a reduction from IND-SET to VERTEX-COVER, proving that the latter problem is **NP**-complete.

We don't want to get carried away with this sort of argument, though. Consider the language NON-CLIQUE, defined to be the set of pairs (G, k) such that G does *not* have a clique of size k. Is this problem **NP**-complete?

In all likelihood, it is not. A Karp reduction must take yes-instances of one problem to yes-instances of the other, so the identity map is not a Karp reduction. In fact it's not at all clear that NON-CLIQUE is in **NP**, because there is nothing that we can guess to prove that a clique does *not* exist.

We define a class **co-NP** to be the set of languages whose complements are in **NP**. (Note that this is quite different from the complement operation taking us from CLIQUE to IND-SET.) We can define **co-NP**-completeness analogously to **NP**-completeness, and see that a language is **co-NP**-complete iff its complement is **NP**-complete. Could a language be both? It follows easily from the definitions that if there is, **NP** and **co-NP** are the same class. This is considered unlikely, though not quite as unlikely as **P** and **NP** being the same.

A *Cook* reduction is allowed to take the answer of a query and negate it, so there is a simple Cook reduction from CLIQUE to NON-CLIQUE. (To decide whether (G, k) is in CLIQUE, determine whether the same pair is in NON-CLIQUE and reverse the answer.) So the **co-NP**-complete problems are all **NP**-hard, though probably not **NP**-complete.

A final note – if we insist that k or $n - k$ be a constant, CLIQUE and these other problems become solvable in **P**, because we now have time to guess all possible cliques (or independent sets, or vertex covers). Sometimes an identifiable special case of an **NP**-complete problem is easier.

The SUBSET-SUM Problem

For our final problem today, we revisit the SUBSET-SUM problem – the input is a set of numbers $\{a_1, \ldots, a_n\}$ and a target number t, and we ask whether there is a subset of the numbers that add exactly to t. Using dynamic programming, we showed that we could decide this language in time that is polynomial in n and s, the sum of all the a_i.

Now we allow the numbers to get larger, so that they now might be n bits long. The problem is still in **NP**, because we can guess a subset by guessing a bitvector, add the numbers in the set, and verify that we get t. But it's no longer clear that we are in **P**, and in fact we will now see that the general problem is **NP**-complete.

We reduce 3-SAT to SUBSET-SUM (with large numbers). We first assume that every clause in our input formula has exactly three literals – we can just repeat literals in the same clause to make this true. Our numbers will be represented in decimal notation, with a column for each of the v variables and a column for each clause in the formula.

We'll create an item a_i for each of the $2v$ literals. This item will have a 1 in the column for its variable, a 1 in the column of each clause where the literal appears, and zeroes everywhere else. We also have two items for each clause, each with a 1 in the column for that clause and zeroes everywhere else. The target number has a 1 for each variable column and a 3 for each clause column.

We now have to prove that there is a subset summing to the target iff the formula is satisfiable. If there is a satisfying assignment, we choose the item for each literal in that assignment. This has one 1 in each variable column, and somewhere from one to three 1's in each clause column. Using extra items as needed, we can reach the target.

Conversely, if we reach the target we *must* have chosen one item with a 1 in each variable column, so we have picked v variables forming an assignment. Since we have three 1's in each clause column and at most two came from the extra items, we must have at least one 1 in each clause column from our assignment, making it a satisfying assignment.

Given a problem with numerical parameters, we say that it is **pseudopolynomial** if it becomes polynomial when those parameters are given in unary. If it is **NP**-complete with parameters given in unary, we say that it is **strongly NP-complete**. The SUBSET-SUM problem is pseudopolynomial, but all our graph problems are strongly **NP**-complete.

Recall that the KNAPSACK is similar to SUBSET-SUM but has a **value** for each item as well as its **weight**. We are asked to find whether a set of at least a given value exists with at most a given weight. Since SUBSET-SUM is an identifiable special case of KNAPSACK (where weight and value are both equal), we know that SUBSET-SUM $\leq_p$ KNAPSACK. Since KNAPSACK (as a decision problem) is in **NP**, it is **NP**-complete. The associated optimization problem is thus **NP**-hard.