CS124	NP-Completeness Review

Where We Are Headed

Up to this point, we have generally assumed that if we were given a problem, we could find a way to solve it. Unfortunately, as most of you know, there are many fundamental problems for which we have no efficient algorithms. In fact, by classifying these hard problems, we can show that there is a large class of simple problems for which there is (probably) no efficient algorithm– the NP-complete problems. Moreover, if you could design an efficient algorithm for *any one* of these problems, you could design an algorithm for *all of them!* It's an all or none proposition, so if you could solve just one of them, you would become rich and famous overnight. These notes will review the main concepts behind the theory of NP-complete problems.

One might ask why it is important to study what problems we cannot solve, instead of focusing on problems we can solve. Especially for an algorithms course. There are several possible responses, but perhaps the best is that if you do not know what is impossible, you might waste a great deal of time trying to solve it, instead of coming to terms with its impossibility and finding suitable alternatives (such as, for example, approximations instead of exact answers).

Polynomial Running Times

The faster the running time, the better. Linear is great, quadratic is all right, cubic is perhaps a bit slow. But how exactly should we classify which problems have efficient algorithms? Where is the cut off point?

The choice computer scientists have made is to group together all problems that are solvable in *polynomial time*. That is, we define a *class of problems* **P** as follows:

Definition: P is the set of all problems Z with a yes-no answer such that there is an algorithm A and a positive integer k such that A solves Z in $O(n^k)$ steps (on inputs of size n).

Let us clarify some points in the definition. The restriction to problems with a yes-no answer is really just a technical convenience. For example, the problem of finding the minimum spanning tree (on a tree with integer weights) can be recast as the problem of answering the following question: is the size of the minimum spanning tree at least j? If you can answer one question, you can answer the other; considering only yes-no problems proves more convenient.

From the definition, all problems with linear, quadratic, or cubic time algorithms are all in **P**. But so are problems with algorithms that require time $\Theta(n^{100})$. This may seem a little strange; for example, would a problem with an algorithm that runs in time $\Theta(n^{100})$ really be said to have an efficient solution? But the main point of defining the class **P** is to separate these problems from those that require *exponential time*, or $\Omega(2^{n^\varepsilon})$ steps (for some $\varepsilon > 0$. Problems that require this much time to solve are clearly *asymptotically* inefficient, compared with polynomial time algorithms. The class **P** is also useful because, as we shall see below, it is closed under polynomial time reductions.

Reductions

Let A and B be two problems whose instances require a "yes" or "no" answer. (For example, 2SAT is such a problem, as is the question of whether a bipartite graph has a perfect matching.) A (polynomial time) *reduction* from A to B is a polynomial time algorithm R which transforms an input of problem A into an input for problem B. That is, given an input x to problem A, R will produce an input $R(x)$ to problem B, such that the answer to x is yes for problem A if and only if the answer for $R(x)$ is yes for problem B.

This idea of reduction should not seem unfamiliar; all along we have seen the idea of reducing one problem to another. (For example, we recently saw how to reduce the matching problem into the max-flow problem, which could be reduced to linear programming.) The only difference is, right now, for convenience we are only considering yes-no type problems.

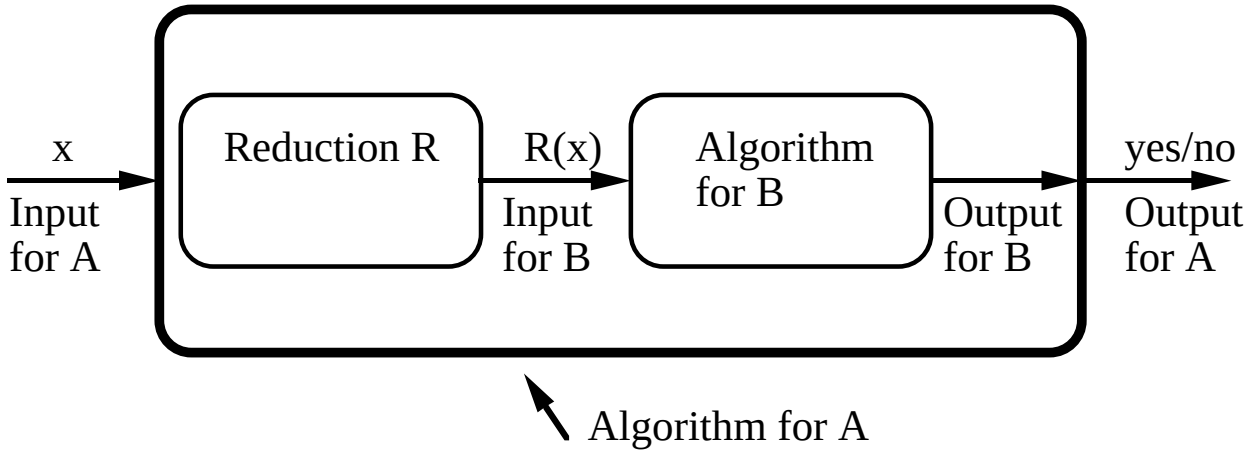

Figure 18.1: Reductions lead to algorithms.

A reduction from A to B, together with a polynomial time algorithm for B, yields a polynomial time algorithm for A. (See Figure 18.1.) Let us explain this in more detail. For any input x of A of size n, the reduction R takes time

$p(n)$, where p is a polynomial, to produce an input $R(x)$ for B. This input $R(x)$ can have size at most $p(n)$, since this is the largest input R could possibly construct in $p(n)$ time! We now submit $R(x)$ as an input to the algorithm for B, which we assume runs in time $q(m)$ on inputs of size m, where q is another polynomial. The algorithm for B gives us the right answer for B on $R(x)$, and hence also the right answer for A on x. The total time taken was at most $p(n) + q(p(n))$, which is itself just a polynomial in n!

This idea of reduction explains why the class **P** is so useful. If we have a problem A in **P**, and some other problem B reduces to it, then B is in **P** as well. Hence we say that **P** is *closed* under polynomial time reductions.

If we can reduce A to B, we are essentially establishing that, *give or take a polynomial,* A is no harder or B. We can write this as

$$\mathrm{A} \leq \mathrm{B},$$

where here the inequality is represents a fact about the complexities of the two problems. If we know that B is easy, then $\mathrm{A} \leq \mathrm{B}$ establishes that B is easy.

We can also look at this inequality the other way. If we know that A is hard, then the inequality establishes that B is hard. It is this implication that we will now use, to show that problems are hard. This way of using reductions is very different from the way we have used reductions so far; it is also much more sophisticated and counter-intuitive.

Short Certificates and the Class NP

We will now begin to examine a class of problems that includes several "hard" problems. What we mean by "hard" in this setting is that although nobody has yet shown that there are no polynomial time algorithms to solve these problems, there is overwhelming evidence that this is the case.

Recall that the class **P** is the class of yes-no problems that can be solved in polynomial time. The new class we define, **NP**, consists of yes-no problems with a different property: if the answer to the problem is yes, then there is a *short certificate* that can be checked to show that the answer is correct. A bit more formally, a short certificate must have the following properties:

- It must be *short*: the length of the polynomial is no more than polynomial in the length of the input.
- It must *certify*: there is a polynomial time checker (an algorithm!) that takes the input and the short certificate and checks that the certificate is valid.

The idea of the short certificate is the following: a problem is in **NP** *if someone else can convince you in*

polynomial time that the answer is yes when the answer is yes, and they cannot fool you into thinking the answer is yes when the answer is no.

Let us move from the abstract to some specific problems.

Compositeness: Testing whether a number is composite is in **NP**, since if somebody wanted to convince you a number is composite, they could give you its factorization (the short certificate). You could then check that the factorization was correct by doing the multiplication, in polynomial time. (Notice you can't be fooled!)

3SAT: 3SAT is like the 2SAT problem we have seen in the homework, except that there can be up to three literals in each clause. 3SAT is in **NP**, since if somebody wanted to convince you that a formula is satisfiable, they could give you a satisfying truth assignment (the short certificate). You could then check the proposed truth assignment in polynomial time by plugging it in and checking each clause. (Again, notice you can't be fooled!)

Finally, note that **P** is a subset of **NP**. To see why, note that if a problem is in **P**, we don't even need a short certificate; someone can convince themselves of the correct answer just by running the polynomial time algorithm!

Now, let us see an example of a problem which does not appear to have short certificates:

not-satisfiable-3SAT: This is like 3SAT, but now the answer is yes if there is no satisfying assignment for the formula. Given a formula with no solution, how can we convince people there is no solution? The obvious way is to list all possible truth assignments, and show that they do not work, but this would not yield a *short* certificate.

NP-completeness

The "hard" problems we will be looking at will be the hardest problems in **NP**; we call these problems **NP**-complete. An **NP**-complete problem will have two properties:

- it is in **NP**
- all other problems in **NP** reduce to it

Thus, our concept of "being the hardest" is based on reductions. If all other problems in **NP** reduce to a problem, it must be at least as hard as any of them! It may seem surprising, that there are problems in **NP** that have this property.

We will start by proving (well, sketching a proof) that an easily stated problem, `circuit SAT`, is **NP**-complete. Once we have a first problem done, it will turn out to be much easier to prove that other problems

are **NP**-complete. This is because once we have one NP-complete problem, it is much easier to prove others:

Claim 18.1 *Suppose problem A is* **NP***-complete, problem B is in* **NP***, and problem A reduces to problem B. Then problem B is* **NP***-complete.*

Intuitively, this must be true because if A reduces to B, then B is at least as hard as A. So as long as B is in **NP**, and the hardest problems in **NP** are the **NP**-complete ones, then B must also be **NP**-complete.

Slightly more formally, we have to show that every problem in **NP** reduces to B. But we already know that every problem reduces to A, and A reduces to B. By combining reductions, as in the picture below, we have that every problem in **NP** reduces to B. So once we have one problem, we can start building up "chains" of **NP**-complete problems easily.

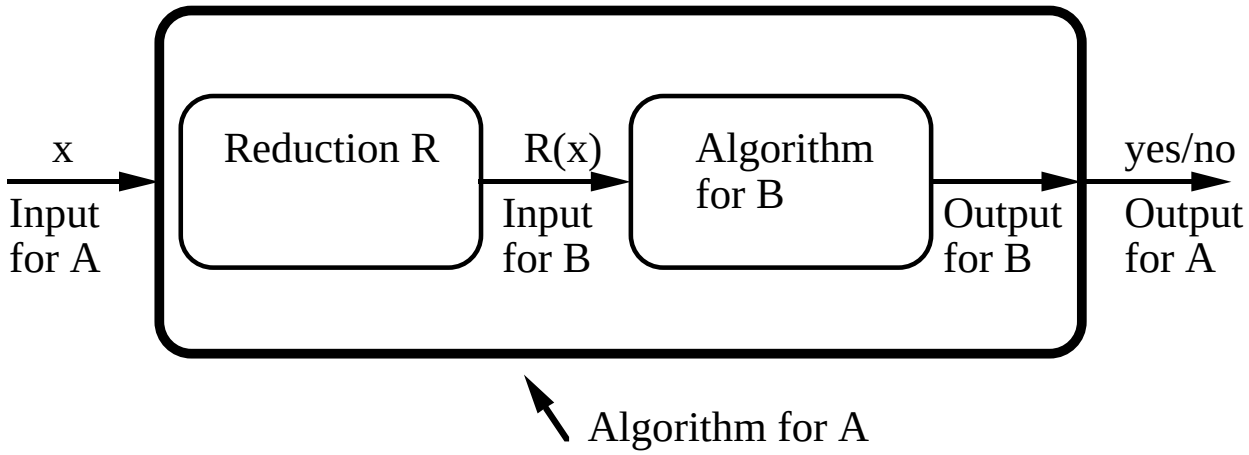

Figure 18.2: If C reduces to A, and A reduces to B, then C reduces to B. (Transitivity!)

Cook's Theorem

The problem **circuit SAT** is defined as follows: given a Boolean circuit and the values of some of its inputs, is there a way to set the rest of its inputs so that the output is T? It is easy to show that **circuit SAT** is in **NP**.

Claim 18.2 *A problem is in* **NP** *if and only if it can be reduced to* `circuit SAT`.

This statement is known as Cook's theorem, and it is one of the most important results in Computer Science.

One direction is easy. If a problem A can be reduced to **circuit SAT** , it can easily be shown to be in **NP**. A short certificate for an input to problem A consists of the short certificate for the circuit that results from running the reduction from A to **circuit SAT** on the input. Given this short certificate, a polynomial time algorithm could run the reduction on the input to A to get the appropriate circuit, and then use the short certificate to check the circuit.

The other direction is more complicated, so we offer a somewhat informal explanation. Suppose that we have a problem A in **NP**. We need to show that it reduces to **circuit SAT**. Since A is in **NP**, there is a polynomial time algorithm that checks the validity of inputs of A together with the appropriate certificates. But we could program this algorithm on a computer, and this program would really be just a huge Boolean circuit. (After all, computers are just big Boolean circuits themselves!) The input to this circuit is the input to problem A along with a short certificate. Now suppose we are given a specific instance x of A. The question of whether x is a yes instance or no instance is exactly the question of whether there is an appropriate short certificate, which is exactly the same question ask asking if there is some way of setting the rest of the inputs to the Boolean circuit so that the answer is T. Hence, the construction of the circuit we described is the sought reduction from A to **circuit SAT**!

More NP-complete problems

Now that we have proved that **circuit SAT** is **NP**-complete, we will build on this to find other **NP**-complete problems. For example, we will now show that **circuit SAT** reduces to **3SAT**, and since **3SAT** is clearly in **NP**, this shows that **3SAT** is **NP**-complete.

Suppose we are given a circuit C with some input gates unset. We must (quickly, in polynomial time) construct from this circuit a **3SAT**-formula $R(C)$ which is satisfiable if and only if there is a satisfying assignment of the circuit inputs. In essence, *we want to mimic the actions of the circuit with a suitable formula*.

The formula $R(C)$ will have one variable for each gate (that is, each input, and each output of an AND, OR, or NOT), and each gate will also lead to certain clauses, as described below:

1. If x is a T input gate, then add the clause (x).
2. If x is a F input gate, then add the clause $(\overline{x})$.
3. If x is an unknown input gate, then no clauses are added for it.
4. If x is the OR of gates y and z, then add the clauses $(\overline{y} \vee x)$, $(\overline{z} \vee x)$, and $(\overline{x} \vee y \vee z)$. (It is easy to see that the conjunction of these clauses is equivalent to $[x = y \vee z]$.

5. If x is the AND of gates y and z, then add the clauses $(\overline{x} \vee y)$, $(\overline{x} \vee z)$, and $(\overline{y} \vee \overline{z} \vee x)$. (It is easy to see that the conjunction of these clauses is equivalent to $[x = y \wedge z]$.

6. If x is the NOT of gate y, then add the clauses $(\overline{x} \vee \overline{y})$ and $(x \vee y)$. (It is easy to see that the conjunction of these clauses is equivalent to $[x = \overline{y}]$.

7. Finally, if gate x is the output gate, add the clause (x), expressing the condition that the output gate should be T.

The conjunction of all of these clauses yields the formula $R(C)$. It should be apparent that this reduction R can be accomplished in polynomial (in fact, in linear) time. To verify it is a valid reduction, we must now show that *C has a setting of the unknown input gates that makes the output T if and only if R(C) is satisfiable*.

Suppose C has a valid setting. Then we claim $R(C)$ can be satisfied by the truth assignment that gives each variable the same value as the appropriate gate when C is run on this valid setting. This truth assignment must satisfy all the clauses of $R(C)$, since we constructed $R(C)$ to compute the same values as the circuit. Note that the output gate is T for C, and hence the final clause listed above is also satisfied.

Conversely (and this is more subtle!), if there is a valid truth assignment for $R(C)$, then there is a valid setting for the inputs of C that makes the output T. Just set the unknown input gates in the manner proscribed by the truth assignment for $R(C)$. Since $R(C)$ effectively mimics the computation of the circuit, we know the output gate must be T when these inputs are applied.

From 3SAT to Integer Linear Programming

We must take a 3SAT formula and convert it to an integer linear program. This reduction is easy. Restrict all variables so that they are either 0 or 1 by including the constraint $0 \leq x \leq 1$. Now a clause such as $(x \vee \overline{y} \vee z)$ can be turned into a linear constraint by replacing $\vee$ by $+$, a literal x by x, and a literal $\overline{x}$ by $(1-x)$, and then forcing the whole thing to be at least 1. For example, the above clause becomes $x + (1-y) + z \geq 1$. The appropriate clause is clearly satisfied if and only if this constraint is; all terms on the left of the equation are either 0 or 1, and there is at least one 1 if and only if one of the literals of the clause is true.

It is somewhat strange that linear programming can be solved polynomial time, but when we try to restrict the solutions to be integers, then the problem appears not be solvable in polynomial time (since it is **NP**-complete).

From 3SAT to Independent Set

In an input to **Independent Set** we are given a graph $G = (V, E)$ and an integer K. We are asked if there is a set

$I \subseteq V$ with $|I| \geq K$ such that if $u, v \in I$ then $(u,v) \notin E$. That is, we are asked to find a set of vertices of size at least K such that no two are connected by an edge. The problem is clearly in **NP**. (Why?)

We reduce **3SAT** to **Independent Set**. That is, given a Boolean formula ϕ with at most 3 literals in each clause, we must (in polynomial time) come up with a graph $G = (V,E)$ and an integer K so that G has an independent set of size K or more if and only if the formula ϕ is satisfiable.

The reduction is illustrated in Figure 18.3. For each clause, we have a group of vertices, one for each literal in the clause, connected by all possible edges. Between groups of vertices, we connect two vertices if they correspond to opposite literals (like x and $\overline{x}$). We let K be the number of clauses. This completes the reduction, and it is clear that it can be accomplished in polynomial time. We now show there is a satisfying truth assignment for ϕ if and only if there is an independent set of size at least K.

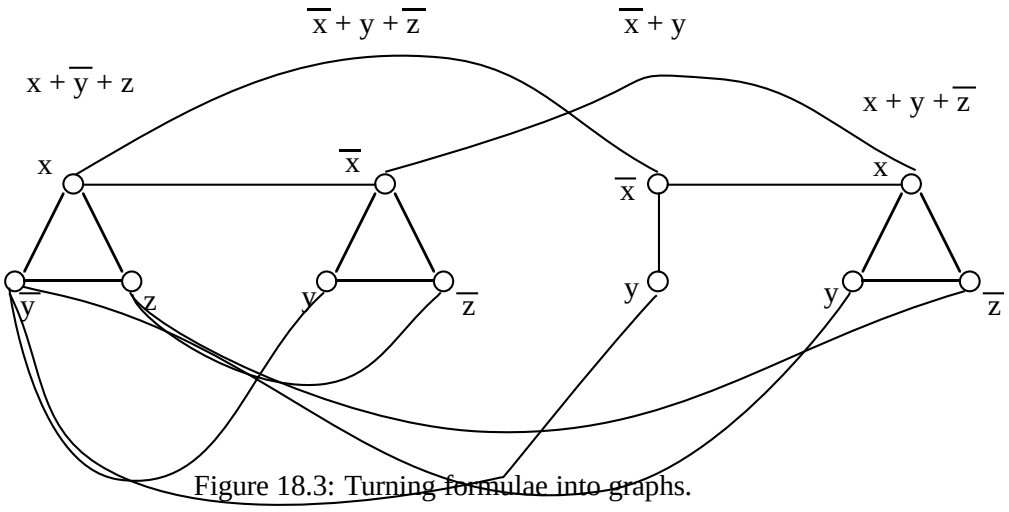

Figure 18.3: Turning formulae into graphs.

If there is a truth assignment for ϕ, then there is at least one true literal in each clause. Pick just one for each clause in any way. The set I of corresponding vertices must give an independent set of size K. This is because we use only one vertex per clause, so the only way I could not be independent is if it included two opposite literals, which is impossible, because the satisfying assignment cannot set two opposite literals to T.

Now suppose G has an independent set I of size K. Since there are K groups, and each group is completely interconnected, there must be one vertex from each group in I. Consider the assignment that sets all literals in the assignment to T, their opposites to F, and any unused variables arbitrarily. It is clear that this is a valid truth assignment (since if a variable is set to T, its opposite must be set to F).

From Independent Set to Vertex Cover and Clique

Let $G = (V,E)$ be a graph. A *vertex cover* of G is a set $G \subseteq V$ such that all edges in E have at least one endpoint

in C. That is, each edge is adjacent to at least one vertex in the vertex cover. The **Vertex Cover** problem is, given a graph G and a number K, to determine if G has a vertex cover of size at most K.

The reduction from **Independent Set** to **Vertex Cover** is immediate from the following observation: C is a vertex cover of $G = (V,E)$ if and only if $V - C$ is an independent set! (For example, suppose I is an independent set, and consider some edge (u,v). Both u and v can't be in the independent set, so $V - I$ contains either u or v or both, and the edge is covered.) So the reduction is trivial; given an instance (G,K) of **Independent Set**, we produce the instance $(G,|V|-K)$ of **Vertex Cover**.

A *clique* in a graph is a set of fully connected nodes– every possible edge between every pair of the nodes is there. The **clique** problem asks whether there is a clique of size K or larger in the graph. Again, the reduction from **Independent Set** is immediate from a simple observation. Let $\overline{G}$ be the *complement* of G, which is the graph with the same nodes as G, but the edges of $\overline{G}$ are precisely those edges that are missing from G. Then C is a clique in $G = (V,E)$ if and only if C is an independent set in $\overline{G}$. (See Figure 18.4.)

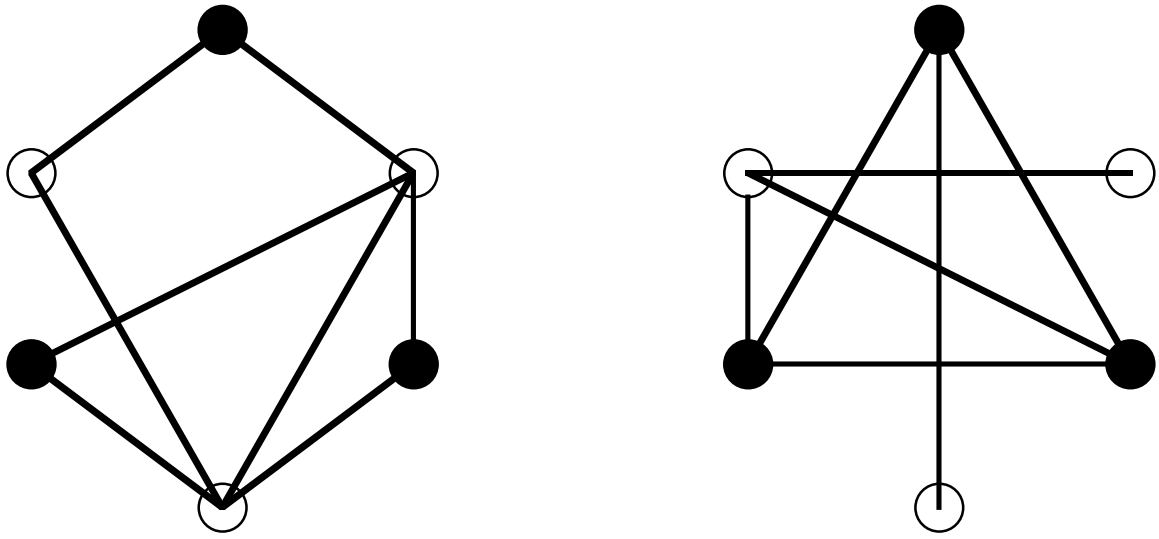

Figure 18.4: Independent sets become cliques in the complement.