Lecture 8,9,10: Computational Complexity

May 28th, June 2nd, June 4th, 2009

1 Polynomial Time Algorithms

A computational problem instance has an input and an objective it wants to compute on that input. An algorithm is a procedure to compute the objective function. The algorithm is said to run in polynomial time, or equivalently the algorithm is a polynomial time algorithm, if there exists a polynomial $p()$, and the time taken by the algorithm to solve every instance of the problems is upper bounded by $p(size(input))$, the polynomial evaluated at the *size of the input*.

To calculate the *time* taken to solve, we normally evaluate the number of *steps* the procedure takes to solve the problem, assuming each step takes *unit time*. The definition of *step* depends on the model of computation, however we will not go into the details of this, and suffice ourselves with the assumption that an arithmetic step or a comparison step taken one unit of time. Thus, for instance, if ever an algorithm requires the addition of two numbers, we will assume this can be done in unit time, *irrespective* of the value of the numbers.

The *size* of the input is more subtle. It consists of two parts. The first is the number of input points that must be provided for the input. For example, in a scheduling problem, $(1||\sum C_j)$ say, the input needs to provide the processing times of all the jobs on the single machine. Thus the number of input points is n, the number of jobs. The second is the size of each input *value*. The size of an input value p_j, the processing time, depends on how this input is encoded. For instance, the value p_j, assume it is an integer, could be stored as a bit-array of p_j 1's, in which case the size is exactly the value p_j, or it can be stored as a binary numeral, as it normally is, in which case the size is $\lceil \log_2 p_j \rceil$. The first method is called the *unary representation* of the input, the second is the *binary representation*. The default input mode is assumed to be the binary representation of the data. The size of a computational problem X is generally denoted as $|X|$.

So, coming back to the problem $(1||\sum C_j)$, the size of the input is $(n + \sum_j \lceil \log_2 p_j \rceil)$ which is at least $n + \log_2 p_{max}$. An algorithm for $(1||\sum C_j)$ is polynomial time if the running time is bounded by $p(n + \log_2 p_{max})$, where $p()$ doesn't depend on the input of the instance.

An algorithm which runs in time polynomial of the data when the data is represented in unary is called a *pseudo-polynomial time* algorithm. Pseudo-polynomial time algorithms are *not* polynomial time. An algorithm which runs in time polynomial of the number of input points, irrespective of the size of the actual data, is called a strongly polynomial time algorithm. Recall that the algorithm SPT, which orders the jobs in increasing order of processing time, runs in time $O(n \log n)$ and thus is strongly polynomial time. (Note that we assume here that the time taken to compare two numbers is unit even if the numbers are *huge*).

1.1 Polynomial Time Reductions

We have already looked at reduction among scheduling problems. The same can be extended to any computational problem.

Definition 1.1. A computational problem X is *polynomial time reducible* to a computational problem Y, if given an algorithm $\mathcal{A}$ for the problem Y with time complexity $T(|Y|)$ (note that T might or might not be a polynomial), we can solve an instance of X in time $(p(|X|) + q(|X|)T(|X|))$. Note that the input to the time function T is the size of X. We denote this as $X \trianglelefteq Y$ or $X \leq_P Y$.

Most often a polynomial time reduction proceeds in the following manner: given an instance of X, we come up with an instance of Y, and argue that given the solution to this instance of Y, one can recover, in polynomial time the solution for the instance of X. We give an example.

Example 1.2. *Consider the following two problems.*

Hamiltonian Circuit Problem (HCP)*: Given a graph G on n vertices and m edges, is there a hamiltonian circuit in G?*
Travelling Salesman Problem (TSP)*: Given a graph G on n vertices (cities) with distance d_{ij} between any two cities i and j, find a tour of minimum distance.*

Note that $HCP \leq_P TSP$. To see this, given an instance of HCP, construct the instance of TSP by defining the distance between vertices i and j as follows – $d_{ij} = 1$ is (i, j) is an edge, and $d_{ij} = 2$ otherwise. It is now not too hard to see that there is a hamiltonian path in the graph iff the optimal tour has cost n.

Why are polynomial time reductions useful? Well, if $X \leq_P Y$, and if we have a polynomial time algorithm for Y, then we have a polynomial time algorithm for X as well. Thus, these reductions help in classifying the problems into "easy" and "hard" problems. Furthermore, if there is a reason to believe that the problem X does not have any polynomial time algorithm, then Y cannot have a polynomial time algorithm either. This is where reductions come in most handy. Unfortunately, there is not a single problem for which we can say there are *no* polynomial time algorithms. However, as we see in the next section, there is a classification of problems into "easy" and "hard" problems with the property that if any of the hard problems have a polynomial time algorithm, then they all do. This is probably the most compelling reason why computer scientists believe that there can be no polynomial time algorithm for any of them. Pretty flaky? That is unfortunately the state of affairs!

2 P, NP and all that Jazz

Before going into the classification of problems, we comment on two different flavours of problems we have seen.

2.1 Decision vs Optimization Problems

Look at the example above. The problem HCP was a "yes/no" question. It asked whether a graph has a hamiltonian cycle or not. The answer can be given in one bit. Such problems are called *decision* problems. The problem TSP however was an optimization problem. It asked for the minimum distance tour. We will concern ourselves with decision problems in this section (although all problems in this course are optimization questions). Every optimization problem X has a decision version X'. For instance, the decision version of TSP has input the input of TSP and an extra integer B, and the problem is: "Given an input of TSP, is there a tour of distance at most B?". Note that this is a decision problem. Furthermore, $TSP' \leq_P TSP$, for if we have an algorithm for the optimization problem, we can surely solve the decision problem. The converse is always not true. However, in most cases, the solution to the decision problem also comes up with a solution which can be used for the optimization problem. Lastly, note that since $X' \leq_P X$, X is *harder* than X'.

2.2 The complexity classes

A computational (decision or optimization) problem is said to be in the complexity class **P** if there exists an polynomial time algorithm to solve it.

A decision problem X is said to be in **NP**, if for every "yes" instance there exists a polynomial sized "certificate" which can be "verified" in polynomial time. Let us elucidate. A "yes" instance of the problem X is one in which the answer is "yes". For instance, a graph which has a hamiltonian circuit is a yes-instance for HCP. A "certificate" for the yes-instance is a proof that the instance is indeed a yes-instance. For instance, given a graph which has a hamiltonian circuit, the circuit is the certificate. The size of the certificate is required to be a polynomial in the size of the input. A "verifier" is nothing but a polynomial time algorithm which takes input the yes-instance and the certificate. If for each yes-instance there exists a certificate, and if the yes-instance can be verified in polynomial time in the size of the input, then the problem is in **NP**. Since given a candidate hamiltonian circuit of the graph, we can verify it is hamiltonian by simply checking the existence of the various edges claimed by the circuit, we get that the problem HCP is in **NP**.

Let us look at another example. Consider the problem $COMPOSITE$ which given an input a natural number N, outputs yes if the number is composite, and no if the number is prime. (What is the size of the input?) We claim that $COMPOSITE \in \mathbf{NP}$. Why? Because for every yes-instance, that is, a composite number N, we can give a certificate, the factors of the numbers, say a and b, and this can be checked to be a proper certificate by just multiplying the two numbers and seeing if it gives N or not.

Remark 2.1. The P in **P** stands for "polynomial time". However, **NP** *does not* stand for "not polynomial". It stands for *non-deterministic polynomial time.* This is because any problem in **NP** *can* be solved in polynomial time of one could "*non-deterministically* guess" the polynomial sized certificate for every yes-instance.

We now give a rigorous definition of the classes. Any decision problem X has two instances, the yes-instance and the no-instance. A certificate or a proof, y, of an instance is a string of $\{0, 1\}$ which may depend on x. y is said to have polynomial size if $|y| = poly(|x|)$. A verifier V takes as input an instance, x, and a proof y, and outputs yes or no.

Definition 2.2. A decision problem $X \in \mathbf{P}$, if there exists a polynomial time algorithm $\mathcal{A}$ such that for every yes instance $x \in X$, $\mathcal{A}x =$ yes; and for every no instance $x \in X$, we have $\mathcal{A}x =$ no.

Definition 2.3. A decision problem $X \in \mathbf{NP}$, if there exists a polynomial time verifier V such that

For every yes instance $x \in X$, there exists a polysized proof y such that $V(x, y) =$ yes.
For every no instance $x \in X$, and for every proof y, we have $V(x, y) =$ no.

What is the relation between the classes $\mathbf{P}$ and $\mathbf{NP}$? Well $\mathbf{P} \subseteq \mathbf{NP}$.

Theorem 2.4. $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. Let X be any decision problem in $\mathbf{P}$ (the optimization problems can be handled similarly). Let $\mathcal{A}$ be an algorithm to solve the problem. Let x be a yes-instance of X. As a verifier, one can just run $\mathcal{A}$ on x to check it is a yes-instance. Thus, the certificate is empty and the verifier is just the algorithm $\mathcal{A}$. $\square$

Are all decision problems in $\mathbf{NP}$? All the problems we have looked at so far are in $\mathbf{NP}$. But that doesn't mean that all problems lie in this class, or at least are not known to lie in this class.

Example 2.5. *Let us turn the COMPOSITE problem on its head to get the PRIMES problem: Given a natural number N, decide whether the number N is a prime number or not. Can you now find a good certificate for yes-instances? That is, given a prime number N, can you provide some extra proof that N is a prime which can be verified in polynomial time? It turns that one can, however the answer is not simple and one needs to use algebra to give this certificate.*

We end this section with the following million dollar exercise.

Exercise 2.6. *Prove or disprove:* $\mathbf{P} = \mathbf{NP}$.

2.3 NP-completeness

It is safe to say that most researchers believe that $\mathbf{P} \neq \mathbf{NP}$. However, many researchers also believe we do not have an inkling of an idea how to solve this problem. What researchers have done however is they can identify the "hardest" problems in $\mathbf{NP}$.

Definition 2.7. A problem Y is called $\mathbf{NP}$-hard if any problem $X \in \mathbf{NP}$ can be reduced to Y, that is, $X \leq_P Y$. Y is called $\mathbf{NP}$-*complete* if $Y \in \mathbf{NP}$.

Thus, $\mathbf{NP}$-complete problems are the hardest problems in $\mathbf{NP}$, if one can be solved in polynomial time then so can all other problems in $\mathbf{NP}$. A priori there is no reason to believe that there will be any $\mathbf{NP}$-complete problem. However, Stephen Cook from UofT showed that there is at least one. The problem is *CIRCUIT SAT*, which asks given a circuit (made of gates and wires) with n boolean ($\{0, 1\}$) inputs and 1 boolean output, is there an assignment of the inputs which makes the output 1?

Theorem 2.8 (S. Cook, 1971). *CIRCUIT SAT* $\in \mathbf{NP}$.

Now comes the beauty of reductions. Given an **NP**-complete problem Y, if we can reduce it to another problem $Z \in \mathbf{NP}$, then Z also is **NP**-complete. This is because the reductions are transitive; $X \leq_P Y$, $Y \leq_P Z$ implies $X \leq_P Z$. Therefore when one problem is proved to be **NP**-complete, reductions can be used to prove many are. Shortly after Cook's paper was published, Richard Karp from Berkeley published the idea of reductions and gave a list of 21 **NP**-complete problems. Now there are hundreds of **NP**-complete problems.

We now give a list of some useful **NP**-complete problems which will be useful to prove **NP**-completeness of some scheduling problems.

1. HCP and TSP are **NP**-complete.
2. **Partition**: Given n positive integers $a_1, \ldots, a_n$, with $B = \sum_i a_i$ assumed to be even, is there a partition of the numbers into two sets S and T such that $\sum_{a_i \in S} a_i = \sum_{a_j \in T} a_j = B/2$.
3. **3-partition**: Given $3n$ numbers $a_1, \ldots, a_{3n}$, and a target B, with $B/4 \leq a_i \leq B/2$ and $\sum a_i = nB$, can we partition the numbers into n-sets $S_1, \ldots, S_n$ each of cardinality 3 such that each set has numbers summing up to exactly B.
4. **Vertex Cover**: Given a graph G and a number B, are there at most B vertices such that each edge in G is adjacent to at least one of these vertices.

In the next lecture or two we will see a list of scheduling problems which are **NP**-complete. The basic schema we will use to prove **NP**-completeness of a problem X will be the following:

1. Show $X \in \mathbf{NP}$ (This is normally easy).
2. Pick a suitable **NP**-complete problem Y.
3. Reduce $Y \leq_P X$.

To reduce a problem X to Y, normally the following three steps is used.

- Come up with a procedure which for an arbitrary instance $x \in X$ gives an instance $y \in Y$ in polynomial time such that
  - YES case: If x is a yes-instance of X, then y is a yes-instance of Y.
  - NO case: If x is a no-instance of X, then y is a no-instance of Y.
- Thus if Y can solved in polynomial time, one can use the above procedure to convert an instance of X to an instance of Y, and check if the Y-instance is yes or no, using the algorithm.

Claim 2.9. *Knapsack problem is* **NP***-complete.*

Proof. The decision version of the knapsack problem asks if there exists a feasible subset of items with profit at least P, where P is an input parameter. We leave checking if Knapsack in **NP**as an exercise. We reduce **Partition** to the knapsack problem. Given an instance of the partition problem: $\{a_1, \ldots, a_n\}$ construct a knapsack instance with n items with item i having weight a_i and profit a_i as well. Let the capacity of the knapsack be $B/2$ and the value of P be $B/2$ as well.

If the instance of partition is a yes-instance, then the instance of knapsack is also a yes-instance. This is because the set of numbers which add up to $B/2$ gives a profit of $B/2$ as well. If the instance of partition is a no-instance, then any set of numbers which add up to at most $B/2$ must be *strictly* less. Thus, any feasible solution of the knapsack instance must have total weight and therefore total profit equalling strictly less than P. Thus, it is a no-instance of the knapsack problem.

This completes the proof. In fact, it proves the special case of knapsack when weight of an item equals its profit is also **NP**-complete. $\square$