Class 48: "Nondeterministic Computing" and Reducibility

Reading

Sections 34.2 and beginning of 34.3 in *Introduction to Algorithms*.

Homework

The HAM-CYCLE problem is as follows: Given a graph G, is there a *cycle* in G that includes every vertex in the graph? A list of vertices is a *cycle* if the list defines a path in the graph, and if there is an edge connecting the last vertex in the list to the first. (Check out this link, which has a nice illustration and then goes on to talking about "hamiltonian cycles" in biology.) Give a polynomial time reduction time reduction of HAM-CYCLE to LongPath(G,k). Note that the graph in the original HAM-CYCLE problem and the graph in the LongPath(G,k) problem do not have to be the same. The T/F *answer* to the original HAM-CYCLE problem instance and the LongPath(G,k) instance you generate must be the same!

P vs NP

Now we know what P and NP are. It's clear that if you can solve a problem in polynomial time on a deterministic machine you can solve it in polynomial time on a nondeterministic machine - just don't use the nondeterministic capabilities - so we know that P $\subseteq$ NP. However, is it true that any problem that can be solved in polynomial time on a nondeterministic machine can be solved in polynomial time on a deterministic machine? in which case we would have P = NP. Well ... I don't know. If you can either prove or disprove P = NP, however, tell me. There are people who are willing to give a lot of meony to anyone who can answer this question. Notice that the "P = NP" problem is first on this list of problems for which the Clay Institute will give $1,000,000 to the solver. They have a nice one paragraph description of P vs NP, and a nice article tying the Minesweeper game to "P vs NP".

This question of whether there are problems in NP that cannot be solved in polynomial time is the biggest open problem in computer science and one of the biggest in math as well. Truth of the matter is, most of us are pretty convinced that P $\neq$ NP, even though we can't prove it.

Reducibility

A common way to solve a new problem is to **reduce** it to an old problem - typically an old problem we know how to solve. For example, in class we talked about how we could reduce the problem of finding the median value in an array to the problem of sorting an array. The idea is that solving an instance of the new problem boils down to solving an instance of the old problem. With decision problems, the idea of "reducing" can be made quite precise.

The decision Problem B is **reducible** to the decision Problem A if there is an algorithm that transforms instance $I_B$of problem A into an instance I_B of problem B that is **equivalent**. In this context, "equivalent" means that the answer to Instance A is true if and only if the answer to Instance B is true.

Example of equivalent problem Instances

Problem A - Set Partition: Given positive integer weights w_1, ..., w_n, *Can the weights be partitioned into two sets of equal weight?*

Problem B - 0/1 Knapsack: Given positive integer weights w_1, ..., w_n and capacity C, *Does some subset of the weights sum to exactly C?*

Instance of B								**Instance of A**							
w_1	w_2	w_3	w_4	w_5	w_6	C	---- Reduce to ---->	w_1	w_2	w_3	w_4	w_5	w_6	w_7
3	2	7	6	5	9	20		3	2	7	6	5	9	8

Instance of B								**Equivalent Instance of A**						
w_1	w_2	w_3	w_4	w_5	w_6	C	<-- Solutions to Both -->	w_1	w_2	w_3	w_4	w_5	w_6	w_7
3	2	7	6	5	9	20		3	2	7	6	5	9	8

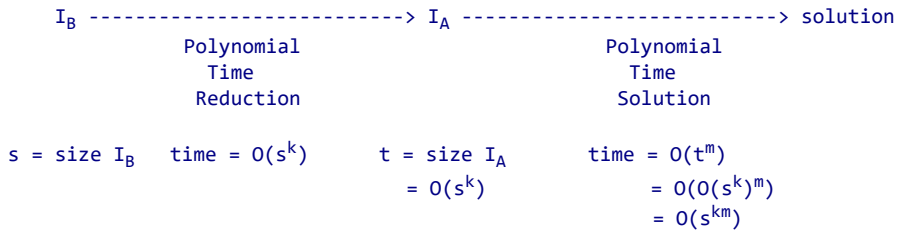
Polynomial Time Reducibility

To investigate the P = NP question we'll be interested in situations in which this "reducing" can be done in polynomial time.

The decision Problem B is **reducible** to the decision Problem A if there is a polynomial time algorithm that transforms instance $I_B$of problem B into an instance I_A of problem A that is **equivalent**. In this context, "equivalent" means that the answer to I_A is true if and only if the answer to I_B is true.

Here's why polynomial time redicibility is such a big deal:

Suppose Problem B is polynomial time reducible to problem A. If there is a polynomial time algorithm for solving A, then we have a polynomial time algorithm for solving B. Here it is:



So we deduce that the time to solve I_B is $O(s^{km})$, which is polynomial in the size of I_B. The only real mystery is why we were able to deduce that t = size I_A = $O(s^k)$, and that is explained by the following.

Time vs. Space

Suppose we know that Algoritm X runs in polynomial time? What can we infer from this? Well, oddly enough, we can infer that the problem can be solved using an amount of memory that's polynomial in the input size! This probably seems a bit odd: We know something about the **time** required to solve this problem, and we somehow infer something about the **space** required to solve the problem. Well, consider this analogy:

```
Suppose that a program runs in 1/10 of a second, and the machine it
runs on executes an instruction every 1/1,000,000 of a second.  What's
the most memory this program could possibly use?

The most memory would be used if every instruction wrote to a new
memory location.  Since instructions reference at most a word at a
time, this would use 1 word = 4 bytes per instruction.  In 1/10 of a
second, our machine performs 100,000 instructions, so we use at
most 400,000 bytes, or 400K.
```

So, you should see how information about runing time translates into information about space usage. We know now that Algorithm X requires only polynomial space, so any output from X will have bit-size that's polynomial in the input size.

Polynomial time reduction of Simple Knapsack to Set Partition

Here is a straightforward algorithm that takes an instance of the Simple Knapsack problem and returns an equivalent instance of the Set Partition problem:

Input: Simple Knapsack problem $w_1, w_2, ..., w_n$ and capacity C
Output: A Set Partition problem that's equivalent to the input Knapsack Problem.

1. set $S = w_1 + w_2 + ... + w_n$
2. if (C >= S/2) set $w_{n+1} = 2*C - S$
   else set $w_{n+1} = S - 2*C$
3. return Set Partition problem $w_1, w_2, ..., w_n, w_{n+1}$

We should prove that the given 0/1 Knapsack problem and the Set Partition problem produced by our algorithm are equivalent.

> **Proposition:** Given Simple Knapsack problem $w_1, w_2, ..., w_n$ and capacity C, the above algorithm returns an equivalent Set Partition problem.
> **Proof:** Let $S = w_1 + w_2 + ... + w_n$. First, we'll show that if the Knapsack problem is true, so is the set partition problem. Suppose that some subset of weights fills the knapsack to

capacity C. The sum of all the wieghts not in the knapsack is S - C. If C ≥ S/2 the partition problem returned by the algorithm adds weight w_{n+1} = 2*C - S, so adding w_{n+1} to the set of weights not in the knapsack gives gives a set of total weight S - C + 2C - S = C, and we have a partition into two sets of weight C. If C < S/2 the partition problem returned by the algorithm adds weight w_{n+1} = S - 2*C, so adding w_{n+1} to the set of weights in the knapsack gives a set of total weight C + S - 2C = S - C, which is equal to the set of weights not in the knapsack. Thus if the original knapsack problem can be solved, so can the returned partition problem.

Next we'll show that if the Partition Problem is true, so is the Knapsack problem. Suppose that the n+1 wieghts in the partition problem can be partitioned into two sets of equal weight. If C ≥ S/2 the partition problem returned by the algorithm adds weight w_{n+1} = 2*C - S, so the sum of all weights in the partition problem is S + 2C - S = 2C. Thus, both partition sets sum to C, and whichever doesn't have the weight w_{n+1} is a solution to the original knapsack problem. If C < S/2 the partition problem returned by the algorithm adds weight w_{n+1} = S - 2*C, so the sum of all weights in the partition problem is S + S - 2*C = 2*(S - C). Thus, both partition sets sum to S - C, and if we remove weight w_{n+1} from whichever set has it, the remaining weights sum to S - C - (S - 2*C) = C and thus provide a solution to the original knapsack problem. Thus if the returned partition problem can be solved, so can the original knapsack problem.

We can't have one problem solvable without the other one being solvable as well, so either both are solvable or neither are solvable. Thus the two problems are equivalent.

Clearly, our algorithm takes polynomial time. Thus, Simple Knapsack is polynomial time reducible to Set Partition, which means among other things that if we can solve Set Partition in polynomial time we can also solve Simple Knapsack in polynomial time. So there's another way to win your million dollars!

Christopher W Brown
Last modified: Fri Apr 22 16:16:41 EDT 2005