[Overall Information]

We finished the outline of the proof of Cook's Theorem, and then looked at complexity classes beyond NP, as well as undecidable problems.

[Announcements]

1. The graded quiz#5 was returned in today’s class. Because there are quite a lot of students who received a score less than 3 points, it is recommended those students should do some further study of NP-Completeness part and turn to TA or professor for help if they have any questions.

2. The homework set No.6 will be assigned this Wednesday. The corresponding solution will be provided next Monday.

3. The quiz#6 is scheduled on 04/26/2006. It is the last quiz on the last day of class.

4. The programming contest will take place this Saturday (04/22/2006) from 1:00 to 6:00pm. Prizes of several hundreds dollars are available to the top participants. Interested students should send email to professor to register.
At the beginning of today’s lecture, we continued our study of Cook’s Theorem. The idea for showing that every NP problem can be reduced to SAT is to show how to encode polynomial computation with polynomial-sized formulas.

Let $x_{i,t}$ represents the ith memory cell status the computation of the verifier algorithm at time t. The important observation is that if these are really all memory cells (including the processor state, etc.), then the action taken at step t+1 (overwriting one or more bits) depends only on the state at time t.
Therefore, we can write polynomially many formulas of the form:

\((\text{not} \ x_{i_1,t}) ^\land (\text{not} \ x_{i_2,t}) ^\land \ldots \Rightarrow (\text{not} \ x_{i,t+1})\) \hspace{1cm}\text{(*)}

which completely describe the algorithm’s behavior. (some of the variables will be negated, others unnegated, and the right-hand side says that a bit in the next step has a certain value.)

After all the computation, if the given input is accepted, the certifier would set \(x_{0,t_{\text{max}}}\) to true, otherwise to false. Thus, we can add the formula consisting of just \(x_{0,t_{\text{max}}}\) and connect it with an “and” with the other parts of the formula. Thus, the overall formula can only be true if that particular bit is true.
The above only concerns the computation of the algorithm. As for the input part, we assume at time 0, memory cells: $x_{1,0}$, $x_{2,0}$, ..., $x_{|X|,0}$ contain the input string $X$ (which we enforce by adding formulas $x_{i,0} = X_i$) and the following $S(|X|)$ cells (called certificate cells) are reserved for the proof $(y)$. All the rest of the cells are initialized to be 0. Thus, all variables except for the certificate cells at time 0 are determined as soon as the certificate variables at time 0 are determined, and the final answer will be true only if the certificate is accepted.
Thus, the formula can be made true if and only if there is a setting of the certificate variables at time 0 that makes the final output true, which happens if and only if there is a certificate leading to the acceptance of x. This proves that the reduction is correct, because the formula is satisfiable if and only if x is a “Yes” instance.
5. Next, the class moved beyond NP-Complete problems. The set $\text{EXPTIME}$ is defined as those problems or languages $X$ that can be decided in exponential time $O(2^{nk})$. We can also define $\text{EXPTIME}$-Completeness using the same idea of reduction as before (a language in $\text{EXPTIME}$ is complete if all other languages can be reduced to it in exponential time). It can then be seen that $\text{EXPTIME}$-complete problems cannot be solved in polynomial time. $\text{EXPTIME}$ gives us explicitly more time to solve a problem, and with more time, one can solve more difficult problems. Thus, these are examples of problems that can be solved, but provably not in polynomial time.
6. While EXPTIME-complete problems are a lot rarer than NP-complete ones, there are some reasoning problems in AI which are known to be EXPTIME-complete; one does encounter them in practice from time to time. Also, it is important to notice why we were able to so easily claim that EXPTIME contains problems that can be solved in polynomial time, when the same question about NP is so difficult. EXPTIME explicitly gives us more time to solve a problem, whereas NP consists of problems for which solutions can be verified efficiently. Thus, comparing P and NP is comparing different resources: verification versus computation time. It is very hard to decide which resource is more valuable. On the other hand, having more time is definitely better.

7. Similar to NP, we can define NEXPTIME as those problems that have an exponential time certifier. The analogous question to whether P=NP, namely “Is EXPTIME=NEXPTIME?”, is also open. It is
known that if P=NP, then EXPTIME=NEXPTIME, but it is not known if the converse is true. Thus, again, it is not clear whether finding a proof is any more difficult than verifying one.

8. We can now continue, and define doubly exponential time (EXP-2) as those problems that can be solved in time \( O(2^{2^{n^k}}) \). This is a strict superset of EXPTIME. And of course, we can define NEXP-2, and complete problems, and so forth.

9. Instead of the resource “time”, we can ask what problems can be solved with a given amount of “space” or memory. In particular, the class PSPACE is defined as those problems solvable using only a
polynomial amount of space/memory. How does PSPACE relate with the complexity classes defined in terms of time?

A good observation is whenever a program reaches the exact same memory state as before (including clock, processor, etc.), it will never terminate, because it will repeat exactly the same instructions, and arrive again at the same state, and so forth. For PSPACE problem, if $O(n^k)$ memory cells are used, then there are at most $2^{O(n^k)}$ (exponentially many) possible memory states. So any program that always terminates cannot repeat any state, and must terminate in $2^{O(n^k)}$ computation steps. So we know that PSPACE is contained in EXPTIME.
10. On the other hand, because polynomial space allows us to do exhaustive search over all possible certificates, we also know that NP is contained in PSPACE. So we know that

\[ P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE} \subseteq \text{EXP}^2 \subseteq \text{NEXP}^2 \subseteq \ldots \]

However, not only is it not known if P=NP, it is not even known if P=PSPACE! Again, the problem is to compare the resource time with the different resource memory. PSPACE-complete problems are not all that rare. They include many two-player games (Chess, Checkers, Geography), in the question: “Given a current state of the game, is there a way to play perfectly and force a win against a perfect opponent?”. Another example is the puzzle game Sokoban.

Many more complexity classes beyond the ones above have been
defined, depending on the amount of randomness that an algorithm uses, whether it can ask for the solution to known hard problems, and so forth.

11. So far, we have seen that there are problems requiring arbitrarily much time or space to solve. Next, we see that some problems just cannot be solved, never mind the amount of time or space we are willing to spend. The first question is: do such problems exist?

There are uncountably infinitely many decision problems in the world. The reason is the following. We can consider problems as functions f: N --> {0, 1}, where f(x) = 1 means that the string/number x is in the language, and f(x) = 0 means that it isn't. We can also interpret f as the infinite binary expression of a real number with f(i) in the ith digit. As there are uncountably many real numbers, there are uncountably many problems
On the other hand, the number of algorithms is only countable infinite since the length of each algorithm must be finite. For instance, we can interpret each algorithm as a binary string, add a 1 at the beginning, and then just number the algorithms this way (the 1 is needed to distinguish between 0 and 000).

Thus, not only are there problems that cannot be decided by any algorithm, but in fact, virtually all problems cannot be solved by algorithms. However, fortunately, most problems that we care about in the real world can be solved – the unsolvable ones for the most part have no intuitive meaning.

However, we will see that there are in fact natural problems that one would really like to be able to solve which cannot be solved.
The halting problem is the language of all pairs \((P, x)\) such that \(P\) is a program, and when run on input \(x\), \(P\) terminates (as opposed to running into an infinite loop). So the question is “Does program \(P\) halt on input \(x\)”? Obviously, it would be useful to be able to answer this, in particular to find out whether a program we have written can ever run into an infinite loop. In the next lecture, we will prove that no program or algorithm can decide the halting problem.

\[
\text{HALT} = \{(P, x) : \text{Program } P \text{ on input } x \text{ terminates}\}
\]

Solution would have to
1. always terminate
2. On input \((P, x)\), output “Yes” if \(P(x)\) terminates; “No” otherwise.

**Theorem**: No algorithm/program can solve the halting problem.