Class NP, NP-complete, and NP-hard problems

W. Hämäläinen

November 6, 2006

1 Class NP

Class NP contains all computational problems such that the corresponding decision problem can be solved in a polynomial time by a nondeterministic Turing machine.

I.e. NP is a time complexity class which contains a set of problems. We should just define what are

- corresponding decision problems,
- a non-deterministic Turing machine, and
- the time requirement of a Turing machine.

1.1 Computational problems and corresponding decision problems

Any problem which in principle can be modelled to be solved by a computer, is called a computational problem. Examples of computational problems are:

- Calculate the answer of an arithmetic expression e.
- Sort customers' names into alphabetic order.
- Given a map of cities and roads, search the shortest route which goes through all cities exactly once and returns to the starting point, if such exists.

Non-computational problems cannot be solved by a computer or by any mechanical means. They are often related philosophy, ethics, or emotions. For example,

- What is the meaning of life?
- Is it right to eat meat on Fridays?
- Phantom and Batman are fighting. Who wins?

In computer science, we consider only computational problems.

For any computational problem, we can inevent at least one related **decision problem**. The decision problem can have only answers "yes" or "no". For example, the following decision problems correspond the previous computational problems:

- Given an arithmetic expression e and some number x, is x the solution of e?
- Given a list of customers' names, are they in an alphabetic order?
- Given a map of cities and roads, does it contain a route which goes through all cities exactly once and is at most 50 km?

Such decision problems can be used to estimate the difficulty of the original problem. If the corresponding decision problem requires exponential time, the original problem cannot be solved any faster, and it is also (at least) exponential. The goal is to invent such a decision problem which measures the difficulty of the original problem as well as possible (i.e. has the same difficulty).

1.2 Non-deterministic Turing machines

A Turing machine is an abstract model of computation. According to Church-Turing thesis, any problem which can be solved by a computer can be solved by a Turing machine. (Note that this is only a thesis, and nobody has been able to prove it universally true, but it is widely believed to be true.) This means that we can use Turing machines to find out whether a problem is solvable or not. If we can solve it by a Turing machine which halts by all inputs, then the problem is solvable, and otherwise unsolvable. In addition, Turing machines can be used to estimate how difficult a problem is or what is its time or space complexity. For example, if a problem requires a Turing machine which runs in exponential time, we cannot invent a Java program which would solve the problem in a polynomial time. The benefit of Turing machines is that they hide all implementation details and it is easier to make proofs and analyze the complexity of problems.

Turing machines are divided into two main types: deterministic and non-deterministic. A **deterministic** machine is like a state machine which reads the input character by character and enters to the next state according to the current state and the read character. In the same time, it writes a character to the tape and moves its reading-writing head to the right or left. This means that the functionality of the machine is **fully determined by the current state and the currently read character**. The computation path is always one sequence of states with any input (from the beginning state to the final state).

A **non-deterministic** Turing machine works otherwise like a deterministic machine, but it can have states where there are several alternative successor states, given the same input character. This means that the computation is not a direct path, but contains branches (it is like a tree with several paths from the same beginning state to final states). The macine always selects the "correct" alternative among possible successor states. I.e. if there is at least one possible path which leads to the accepting final state, the machine accepts the input (answers "yes"), and otherwise it rejects it (answers "no"). You can imagine that the non-deterministic machine has power to "guess" the correct path or that it simulates all possible (direct) computation paths simultaneuosly and selects the best one. (If we want to implement a non-deterministic Turing machine as a computer program, we really have to simulate all possible computations with that input and test if any of them leads to the accepting final state.)

1.3 Time requirement of a Turing machine

The time complexity of a problem is estimated by Turing machines which solve the problem. If we know a solution (a Turing machine) to the problem and it has complexity x, we know that the problem has at most complexity x. It is possible that in the future somebody invents a faster solution, and thus x is only an upperbound for the complexity. Let's take an example:

Let's suppose that we have a Turing machine M which solves the Travelling Salesman's problem. We give the machine an input, i.e. a graph which describes the cities and roads, and ask if it contains a route of length ≤ 20 km which goes through all cities exactly once and returns to the starting place. The size of the input is measured by the number of cities n and the number of roads between them (edges of the graph), e. We know that a graph of n nodes can contain at most $\frac{n(n-1)}{2}$ edges[1], and thus the input size is at most $\frac{n(n-1)}{2}$. Now we can express the time requirement of the Turing machine as a number of steps it takes to solve the problem. I.e. how many steps the machine takes, before it can answer "yes, the map contains such a route" or "no, it doesn't". The time requirement should be expressed as a function of n, e.g. $f(n)$, so that for any input the machine takes at most $f(n)$ steps. This is the time **complexity of the solution algorithm**. In addition, it tells that the problem has at most complexity $f(n)$ (so it can be less complex, if we just invent a better solution).

Note that usually the complexity of a solution algorithm is expressed by O-notation which tells only the order of function $f(n)$. I.e. we give a simpler function $g(n)$ which has the same "growing rate" as $f(n)$. For example, if $f(n) = 77n^5 + 16n^3 - 12n^2 + 3$, we say that the worst case complexity is $O(n^5)$, because it is the most important term in $f(n)$ and dominates its growth when n increases. The constant factor 77 is dropped from the simpler expression. Or, if $f(n) = a2^{n^2-5n+7} + 3n^2$, we say that the complexity is $O(2^{n^2})$. The simpler function $g(n)$ doesn't have to hold for small values of n, but it is enough that the real comlexity $f(n)$ is at most $ag(n)$ when n is high enough, given some constant factor a. I.e. we can use complexity $O(g(n))$ instead of $f(n)$, if $f(n) \leq ag(n)$, when $n \geq n_0$ for some n_0 and some constant a.

1.4 Complexity classes P, E and NP

If a problem can be solved **by a deterministic Turing machine in polynomial time**, the problem belongs to the complexity class P. All problems in this class have a solution whose time requirement is a polynom on the input size n. I.e. $f(n)$ is of form $a_k n^k + a_{k-1}n^{k-1} + ... + a_2 n^2 + a_1 n + a_0$ where $a_k, ..., a_0$ are contant factors (possibly 0). The order of polynom is the largest exponent k such that $a_k \neq 0$ (if $a_k = 0$, then the polynom is $a_{k-1}n^{k-1} + ... + a_0$ and the order is $k-1$, unless $a_{k-1} = 0$. If $a_{k-1} = 0$, too, then the polynom is $a_{k-2}n^{k-2} + ... + a_0$ etc.).

If the problem can be solved **by a deterministic Turing machine in exponential time**, then it belongs to class E. The time complexity can be for example $f(n) = 2^n$ or $f(n) = n^n$. It doesn't matter, if $f(n)$ contains also polynomial part, because the exponential part (where n is in the exponent) dominates the complexity. So, for example $f(n) = 2^{n/3} + 10x^{10}$ has complexity $O(2^{n/3}) = O(2^n)$, and is an exponential function. Note that if a problem belongs to class P, then it also belongs to class E, because $P \subset$

[1]The first city can have roads to $n-1$ other cities, the second one to $n-2$ cities in addition to city 1, the third city to $n-3$ cities, etc.

E, but usually we give only the smallest (fastest) class where the problem belongs. Note also that E is not the most complex class of problems, but for example problems with complexity $O(n!)$ ($n! = n(n-1)(n-2)...1$) are more difficult.

An interesting class of problems is class NP which contains all problems which can be solved **by a non-deterministic Turing machine in polynomial time**. Class NP is between classes P and E: $E \subseteq NP \subseteq E$. This means that all problems in class NP belong to class E, too, and they can be solved by a deterministic Turing machine (or a computer program) in an exponential time. It is also possible that some of them can be solved faster, in polynomial time, and they actually belong to class P. However, it is not known, if all of them could be solved in polynomial time by some clever algorithms. So, the important open question is whether $P = NP$ or $P \subsetneq NP$. (A million dollars is yours, if you solve this!)

2 NP-complete problems

NP-complete problems are special problems in class NP. I.e. they are a subset of class NP. An problem p is NP-complete, if

1. $p \in NP$ (you can solve it in polynomial time by a non-deterministic Turing machine) **and**
2. All other problems in class NP can be reduced to problem p in polynomial time.

This means that the NP-compelete problems are the most difficult problems in class NP. If we could solve just one of them in a polynomial time, we could solve all problems in class NP in a polynomial time (and win 1 000 000 dollars).

If you want to show a new problem to be NP-complete, you have to do two things:

1. Invent a non-deterministic Turing machine which solves the problem in polynomial time.
2. Reduce one known NP-complete problem to the new problem in a polynomial time.

The first part is usually easy. You just construct a machine which contains two parts. The first part guesses an answer non-deterministically (e.g. a list of cities $c_1, c_2, ..., c_n$ for TSP) and the second part checks that the answer is correct (e.g. the list of cities contains all cities exactly once and the length of route $c_1 - -c_2 - c_3 - ... - c_n - c_1$ is less than required).

The idea of the second part is the following:

1. You want to show that all poblems in class NP can be reduced to problem p.
2. You select a known NP-complete problem q. This means that all problems $p_i \in NP$ can be reduced to q in polynomial time pol_i (and you don't have to show it anymore!).
3. You give a method to reduce q to p in polynomial time pol
4. Because all $p_i \in NP$ can be reduced to q and q can be reduced to p, all $p_i \in NP$ can be reduced to p in time $pol_i + pol$ which is also a polynom (a polynom + polynom is still a polynom).

Notice that if your reduction from p to q would take exponential time, then you could just say that you can reduce all problems in NP to p in exponential time (an exponential function + a polynom = an exponential function). That's why the reduction should also take a polynomial time.

Inventing the reductions is often a matter of art, and it is best to study some exaples from text books. Usually it is enough to describe the idea of the reduction method and show that it takes polynomial time. The reduction from q to p should be such that

1. It transforms all inputs of problem q to inputs of problem p, and
2. If the solution to p is "yes", the solution to q is "yes", and if the solution to p is "no", the solution to q is "no".

For example, we can reduce Independet set (IS) problem to Clique problem as follows:

If graph G contains an independent set of size k (i.e. there are at least k vertices which are not connected to each other), then the complement graph $\overline{G}$ contains a clique of size k (there are at least k vertices which are all connected to each other). Complement graph $\overline{G}$ means a graph which contains an edge between two nodes, if G didn't, and it doesn't contain an edge between two nodes, if G did. Thus, the reduction is a method, which trasfroms G to its complement graph $\overline{G}$.

This reduction can be done in a polynomial time, because we just have to change 0s and 1s in the matrix describing graph G. The matrix contains

$n \times n$ cells, but if the graph is undirected, it is enough to use only $\frac{n(n-1)}{2}$ of them (If $v1$ is connected to $v2$, then $v2$ is also connected to $v1$. Edges from the vertex to itself, e.g. from $v1$ to $v1$, are not marked.) So the reduction takes time $O(n^2)$ and is polynomial.

3 NP-hard problems

NP-hard problems are partly similar but more difficult problems than NP-complete problems. They don't themselves belong to class NP (or if they do, nobody has invented it, yet), but all problems in class NP can be reduced to them. Very often, the NP-hard problems really require exponential time or even worse.

Notice that NP-complete problems are a subset of NP-hard problems, and that's why NP-complete problems are sometimes called NP-hard. It can also happen that some problem which is nowadays known to be only NP-hard, will be proved to be NP-complete in the future – it is enough that somebody just invents a nondeterministic Turing machine, which solves the problem in polynomial time.

To summarize the relationships:

NP-complete $\subsetneq$ NP-hard
NP-complete $\subseteq NP$
$P \subseteq NP \subseteq E$