W4231: Analysis of Algorithms

11/18/99

- NP and NP-completeness
- NP-completeness of Circuit Sat and CNF SAT.

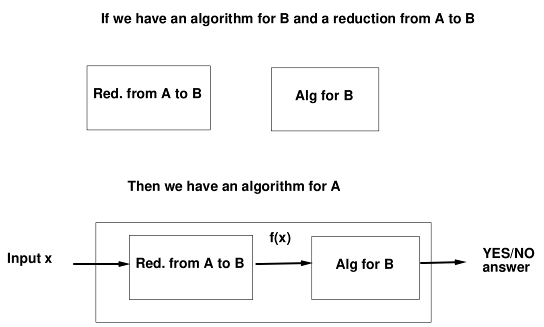
Formal Definition of Reduction

For a decision problem A, we use the notation $x \in A$ to mean "x is an input instance for which the right solution according to problem A is YES".

We say that A is reducible to B is there is a polynomial time computable function f such that

$$x \in A \text{ if and only if } f(x) \in B$$

Use of a Reduction



NP — Informal Definition

A decision problem A is in NP if A can be expressed as follows

"on input x, the answer is YES iff there is some y which has a certain efficiently testable property with respect to x."

For example:

- Given a graph G and an integer k, is there a simple path of length at least k in G?
- Given a set of integers $a_1, \ldots, a_n$, is there a subset S of them such that $\sum_{a \in S} a = \sum_{a \notin S} a$?

NP — Formal Definition

A problem A is NP if there exist a polynomial p and a polynomial-time algorithm $V()$ such that

$x \in A$ iff there exists a y, with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.

Notation: we call P the set of decision problems that are solvable in polynomial time.

Observation: every problem in P is also in NP.

NP-hard and NP-complete Problem

A problem A is NP-hard if for every N in NP, N is reducible to A. A problem A is NP-complete if it is NP-hard and is contained in NP.

Interesting fact: if A is NP-complete, then A is in P if and only if P=NP.

Possibilities:

- A has no efficient algorithm.
- All the infinitely many problems in NP, including factoring and all conceivable optimization problems are in P.

NP-complete problems exist

Here is an NP-complete problem that we call U (for *universal*).

The *unary* representation of a number k is a sequence of k ones, also written 1^k.

- **Definition.** Given a program P, an input x, and numbers k, t represented in unary, is there a y (of length $\leq k$) s.t. $P(x, y)$ terminates in $\leq t$ steps and outputs YES?

- **U is in NP.** We show the existence of an algorithm V such that $(x, 1^k, 1^t, P) \in U$ iff there exists a y with $length(y) \leq k$ such that $V(x, k, y)$ outputs YES. We define $V(x, k, y)$ as the algorithm that simulates $P(x, y)$ for t steps, and accepts iff P terminates and accepts.

- **U is NP-hard.** Suppose A is NP via $V_A()$ running in time $q(n)$ and p. Then we can reduce A to U using the reduction that maps x to $(x, 1^{p(length(x))}, 1^{q(length(x))}, V_A)$.

Use of the result U

Clearly, we do not care about U itself.

However suppose that we can prove that U reduces to Hamiltonian path.

Then if follows that Hamiltonian path is NP-complete.

By proving one reduction, we get infinitely many more for free.

Note: if A reduces to B and B reduces to C, then A reduces to C.

General Result

Suppose A is NP-complete, and B is a problem in NP such that A is reducible to B. Then B is NP-complete.

Proof: We just show that B is NP-hard (we are given as an assumption that is in NP). Fix a problem N in NP. Then N reduces to A. By assumption A reduces to B. Then N reduces to B. QED

The number of corollaries is the square of the number of theorems

It suffices a single reduction to show the NP-completeness of a new problem.

With a thousand reductions, we can show a thousand problems that are NP-complete.

So we also show that a million reductions exist (between any two such problems).

A More Useful Starting Point

U is somewhat undefined, in that we have not said what is the language in which we write programs. We could use Turing machines (because we want the simplest possible language, and they can simulate in polynomial time every other model).

Even if we use Turing machines, U is a complicated language to reduce from.

We will prove that a simpler problem, circuit satisfiability, is NP-hard.

Circuits

A circuit is made of *gates* and *wires*. We consider circuits without feedback, so we can see a circuit as a directed acyclic graph.

A gate can be: a *input gate*, a *OR* gate, a *AND* gate, or a *NOT* gate.

AND and OR gates have fan-in two and unbounded fan-out.

There is a special gate that is called the *output* gate.

A circuit C with n input gates computes a boolean function $C()$ in the following way: if $x_1, \ldots, x_n$ are the values received by the n input gates, then $C(x_1, \ldots, x_n)$ is the value of the output gate.

Circuit Satisfiability

Definition. Given the description of a circuit C with n gates, does there exist a sequence of n Boolean values $(x_1, \ldots, x_n) \in \{0,1\}^n$ such that $C(x_1, \ldots, x_n) = 1$.

Belongs to NP**.** It is easy to see that circuit satisfiability belongs to NP. The algorithm $V()$ takes in input the description of a circuit C and a sequence of n Boolean values $x_1, \ldots x_n$, and $V(C, x_1, \ldots, x_n) = C(x_1, \ldots, x_n)$. I.e. V *simulates* or *evaluates* the circuit.

Main Result

Suppose A is a decision problem that is solvable in $p(n)$ time by some program P, where n is the length of the input. Also assume that the input is represented as a sequence of bits.

Then, for every fixed n, there is a circuit C_n of size about $O((p(n)^2))$ such that for every input $x = (x_1, \ldots, x_n)$ of length n, we have

$$x \in A \text{ if and only if } C_n(x_1, \ldots, x_n) = 1$$

That is, circuit C_n solves problem A on all the inputs of length n.

Furthermore: there exists an efficient algorithm (running in time polynomial in $p(n)$) that on input n and the description of P produces C_n.

Sketch of the Proof of the Main Result

Without loss of generality, we can assume that the language in which P is written is some very low-level machine language (as otherwise we can compile it).

Let us restrict ourselves to inputs of length n. Then P runs in at most $p(n)$ steps. It then accesses at most $p(n)$ cells of memory.

At any step, the "global state" of the program is given by the content of such $p(n)$ cells plus $O(1)$ registers such as program counter etc. No register/memory cell needs to contain numbers bigger than $\log p(n) = O(\log n)$. Let $q(n) = (p(n) + O(1))O(\log n)$ denote the size of the whole global state.

We maintain a $q(n) \times p(n)$ "tableau" that describes the computation. The row i of the tableau is the global state at time i. Each row of the tableau can be computed starting from the previous one by means of a small circuit (of size about $O(q(n))$). In fact the microprocessor that executes our machine language is such a circuit (this is not totally accurate).

There is a lot of handwaving here.

The proof can be made formal if we assume P was a set of instructions for a Turing machine.

Then we have to argue that every program can be translated into a Turing machine while preserving efficiency.

Circuit Satisfiability is NP-hard

Take an NP-problem A. We reduce A to Circuit Satisfiability.

Since A is in NP, there is some polynomial-time computable algorithm V_A and a polynomial p_A such that

$x \in A$ if and only if there exists a y, with $length(y) \leq p_A(length(x))$, such that $V(x, y)$ outputs YES.

The reduction

On input x of length n, we construct a circuit C that on input y of length $p(n)$ decides whether $V(x, y)$ outputs YES or NOT.

Since V runs in time polynomial in $n + p(n)$, the construction can be done in polynomial time.

Now we have that the circuit is satisfiable if and only if $x \in A$.