SE 3310b

Theoretical Foundations of Software Engineering

NP-Complete and the Million Dollar Question: Does P=NP?

Aleksander Essex

Western Engineering

P vs. **NP**

P vs. **NP**

How do **P** and **NP** relate? First off we note that:

- **P** is the class of problems solvable by a **DTM in polynomial time**. Informally it is the class of problems that can be solved ***quickly***,
- **NP** is the class of problems solvable by an **NTM in polynomial time**. Informally it is the class of problems to which proposed solutions can be ***checked*** quickly.

Western Engineering

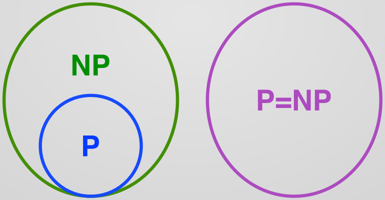
The Million Dollar Question

The first thing to notice is that **P** is a subset of **NP**: recall all DTMs are ultimately a subset of NTMs. So whatever a DTM can solve in polynomial time, an NTM can solve in polynomial time. Now there are only two possibilities:

- **P**!=**NP**: There exist problems solvable by an NTM in polynomial time for which there exists no DTM that can solve them in polynomial time,
- **P**=**NP**: Every problem solvable by an NTM in polynomial time has an equivalent DTM that can solve it in polynomial time.

P vs. **NP**

So which is it? As it turns out, this presently the greatest open problem in computer science. In fact the Clay Mathematics Institute is offering one million dollars for a solution.



Although we still don't know, but many people believe that **P**!=**NP**.

FACTOR: An Example NP Problem

FACTOR: An Example NP Problem

By the fundamental theorem of arithmetic, every integer greater than 1 can be expressed as a *unique* product of prime numbers. Given such an integer, factoring is the process of recovering the associated prime factors.

In it's basic form, factoring is not a **decision** problem, so let's restate it as one:

FACTOR($< n, b >$):
    If n has a factor f for $1 < f < b$, return *True*.
    Else return *False*.

Western Engineering

FACTOR

Theorem: FACTOR ∈ **NP**.
Proof: FACTOR (as stated above) is a decision problem and if FACTOR ∈ **NP** then there exists a polytime verfifer for it. Let VerFactor be defined as follows:

```
VerFactor($< n, b >, c$)
    If $b$ divides $n$ return *true*,
    Else return *false*
```

Clearly VerFactor is a verifier for FACTOR: if c is a certificate for FACTOR (i.e., is a factor of n below bound b), VerFactor will identify as such. It also clearly runs in polytime: checking if a divides b can be done with the well known polytime Euclidean Algorithm. □

Factoring Algorithms

The difficulty of FACTOR is central to the security of the RSA cryptosystem, upon much of the internet's communicaition security is based. Currently the General number field sieve is the best known algorithm for integer factorization running in subexponential time:

$$O(2^{((\frac{64}{9})^{\frac{1}{3}}+o(1))\log(n)^{\frac{1}{3}}\log\log(n)^{\frac{2}{3}}})$$

The GNFS is rather difficult to comprehend, so the (somewhat slower but much simpler) Quadratic Sieve is a good place to start if you're interested (the link gives an intuitive example).

NP-Completeness: The Hardest **NP** Problems

NP-Completeness

Now that we've seen a number of **NP** problems we're starting to get the sense that some problems might be harder than others. For example, multiplying two integers is easier than factoring their product (so it would seem based on our state-of-the-art algorithms).

A natural question arises: are some problems in **NP** fundamentally harder than others? Alhough we cannot answer this until we settle the **P** vs. **NP** question, we can ask a related question:

Are there problems in **NP** that are at least **as hard** as every other problem in **NP**?

Western Engineering

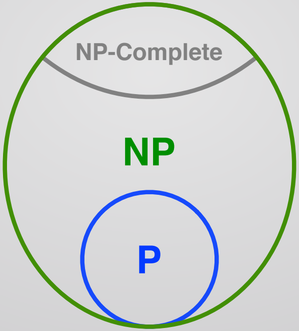
NP-Completeness

At first this might seem like an unsatisfying question: if we don't know if some problems in **NP** are fundamentally harder than others (because we don't know if **P**=**NP**), then what does it really *mean* to talk about some problems being *at least as hard* as all the others?

The idea is we're placing an *upper bound* on the hardness of **NP** problems. If we can prove problem $X \in$ **NP** is *at least as hard* as all other **NP** problems then we know that problem $Y \in$ **NP** is no harder.

The class **NP-Complete**

Here NP-complete is shown as a region inside **NP** where the ”hardest” problems reside.



The class **NP-Complete**

NP-complete is a sub-class of **NP** containing problems that can be shown to be *at least as hard* as all other problems in *NP*. The problems in NP-complete are related to the complexity of the entire class:

- If **P**=**NP**: Finding a polynomial-time algorithm for an **NP-complete** problem would prove **P**=**NP** (more on this to follow)
- If **P**!=**NP**: The **NP-complete** class provides a reference set of *the hardest* problems in **NP**.

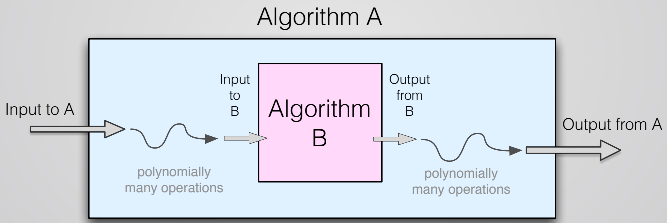
Polynomial-time Reduction

Let’s move on to a more formal definition. At the heart of NP-completeness is the notion of a polynomial-reduction. The idea of a poly-time reduction is to take an a problem, like deciding if s_A is in a language L_A and *efficiently* transform it into deciding if s_B is in language L_B.

Ok, suppose there was a way to efficiently turn some problem A into problem B. So what?

Polynomial-time Reduction

Here's the idea: suppose you have an algorithm for solving problem B, and that you can use it as a *sub-routine* as part of an algorithm for solving problem A:



We're interested in the time A takes to run (*not including* the call to B), because if it's in polynomial time, then we know we can solve A in whatever time it takes to solve B, plus a polynomial factor.

Western Engineering

Polynomial-time Reduction

Here's more pseudo-code explanation of a reduction. Let M_A and M_B be deciders for L_A and L_B respectively. We can use polynomial-time reduction (if one exists) to implement M_A as follows:

$M_A(s)$:

- Some code transforming s into an appropriate s'
- $out_B = M_B(s')$ // Call M_B on s'
- Some code transforming out_B into appropriate out_A
- Return out_A.

We say L_A polynomial-time reduces to L_B.

Polynomial-time Reduction

So what? Well you can see where this is going: The existence of a polynomial-time reduction from problem A to problem B means that solving problem A is essentially *no harder* than solving problem B.

That's because if you can prove the existence of a polynomial-time reduction from A to B, you're effectively saying A takes however long B takes, plus a polynomial overhead (of converting an instance of A into an instance of B).

Polynomial-time Reduction

Definition 1 (Polynomial-time Reduction)**.**

A language A is **polynomial-time reducible** to a language B, written $A \leqslant_P B$, if a polynomial-time function $f : \{0,1\}^* \to \{0,1\}^*$ exists such that for all w:

$$w \in A \iff f(w) \in B$$

In other words: f is a polynomial-time reduction from A to B if given some string w, (a) you can compute $x = f(w)$ in polynomial time, and (b) $w \in A$ if and only if $x \in B$.

Western Engineering

The class **NP-Complete**

Definition 2 (**NP-Complete**).

A language B is **NP**-complete if:

1. $B \in$ **NP**
2. For all $A \in$ **NP**, $A \leqslant_P B$

In other words, B is **NP**-complete if it is an **NP** problem, and there exists a polynomial-time reduction from every other **NP** problem to it.

Consequences of NP-complete

If you have an algorithm to solve an NP-complete problem B in time $O(f(n))$, any other problem in **NP** is solvable in time $O(f(n) + \text{poly(n)})$. Here are some interesting consequences:

- If B is NP-complete and $B \in$ **P** then **P**=**NP**.
- If B is NP-complete and $B \leqslant_P C$ and $C \in$ **NP**, then C is NP-complete
- If B and C are both NP-complete, then $B \leqslant_P C$ but also $C \leqslant_P B$

The Cook-Levin Theorem

Let's first begin with a definition. The Boolean satisfiability problem (SAT) is defined as follows: let ϕ be a *conjunctive-normal form* (CNF) boolean equation of k clauses and n variables:

$$\phi = (a_1\ OR\ a_2\ OR\ a_3 \ldots)\ AND\ (a_4\ OR\ a_5 \ldots)\ AND \ldots$$

SAT asks the following question: given ϕ, is there *any* assignment of variables $a_1 \ldots a_n \in \{0, 1\}$ such that ϕ equals TRUE?

The Cook-Levin Theorem

Theorem: SAT is NP-complete.

Proof. See Theorem 7.37 in the text for the proof. The idea is you can convert any problem in **NP** into a CNF Boolean circuit in polynomial time. This is known as the *Cook-Levin theorem* after Stephen Cook and Leonid Levin who first proved this result.

Cook-Levin Theorem: So what?

The Cook-Levin theorem establishes SAT as the first NP-complete problem by showing that all problems in NP are polytime reducible to SAT. That initial proof was the hard part.

Now if you have some new problem X and want to prove it NP-complete, you only have to prove two things:

1. Problem X is in **NP**
2. 3SAT$\leqslant_P X$

Cook-Levin Theorem: So what?

Showing a problem is in **NP** is straightforward enough: show it's a decision problem that is checkable in polynomial time.

Finding a polynomial-time reduction from 3SAT to X (if it exists) is typically easier than showing *all* problems in **NP** reduce to X. Thus Cook-Levin gives us a kind of toe-hold into NP-completeness.

Reduction Example 1

3SAT is a version of SAT where each clause has 3 variables *e.g.*,

$$\phi = (a_1 \ OR \ \bar{a_2} \ OR \ a_3) \ AND \ (\bar{a_1} \ OR \ a_4 \ OR \ a_5) \ \ldots$$

CLIQUE(G, n) is the problem where, given a graph G and a number n, decide if there are n nodes in G that are fully connected to each other (*i.e.*, form an n-clique). For example, if you were friends with two people on Facebook who were also friends with each other, your "social graph" would contain a 3-clique.

Theorem: 3SAT$\leqslant_P$CLIQUE.

Reduction Example 1

Proof. First you would show that you can convert ϕ into an associated graph G in polynomial time. By "associated" we mean ϕ is satisfiable if and only if G contains a 3-clique. See **Theorem 7.32** for the full proof.

Theorem: CLIQUE is NP-complete.
Proof.

1. It is easy to see CLIQUE is in **NP**: evaluating an assignment can be done in polynomial time in ϕ.
2. All problems in **NP** are polynomial-time reducible to CLIQUE because all problems in **NP** reduce to 3SAT by the Cook-Levin theorem, and as we saw, 3SAT reduces to CLIQUE.

Western Engineering

Reduction Example 2

Example 2: Prove SAT is polynomial-time reducible to the acceptance problem for Turing machines, *i.e.*, $SAT \leqslant_P A_{TM}$.

Proof. Let B be an algorithm for solving SAT and C be an algorithm for deciding A_{TM}, *i.e.*, whether a given TM M accepts an input s. We can construct B such that it turns an instance of SAT into an instance A_{TM} in polynomial time (see next slide).

Reduction Example 2

B accepts a string $\langle \phi \rangle$ corresponding to an instance of 3SAT, ϕ. It defines a TM M (see below) and passes $\langle M, \phi \rangle$ to C, and returns whatever C returned:

$B(\langle \phi \rangle)$:

1. Construct a string $\langle M, \phi \rangle$ where M is the description of TM M that runs on $\langle \phi \rangle$ with the following behavior. M loops through all possible assignments of ϕ. If a truth assignment t such that $\phi(t) =$ *true*, M halts and outputs accept. Otherwise M tries the next assignment.
2. $out_C \leftarrow C(\langle M, \phi \rangle)$ // Call C to decide if M ever accepts ϕ.
3. return out_C

Reduction Example 2

Proof cont'd. We need to prove 2 things: (1) B decides 3SAT, and (2) B runs in polynomial time (not including the call to C).

1. M eventually halts and accepts if ϕ is satisfiable. If ϕ is unsatisfiable, M loops forever. By definition C accepts $\langle M, \phi \rangle$ if and only if M accepts ϕ, and B simply outputs what C outputs. Therefore B accepts if and only if ϕ is satisfiable. Therefore algorithm B decides $3SAT$.

2. Clearly B runs in polynomial time: constructing string $\langle M, \phi \rangle$ takes polynomial time in the size of ϕ, and outputting C's result takes polynomial (*i.e.*, constant) time. $\square$

Reduction Example 2

Theorem: A_{TM} is NP-Hard.
Proof.

1. All problems in **NP** are polynomial-time reducible to A_{TM} because all problems in **NP** reduce to SAT by the Cook-Levin theorem, and as we saw, SAT reduces to A_{TM}.

Notice we didn't say A_{TM} is NP-Complete. For that to be true we would also have to show A_{TM} was in **NP**, which means solutions are checkable in polynomial time. Except A_{TM} is undecidable. Thus it is not in **NP**.