Efficient Algorithms and Problem Complexity
– Reductions, NP-completeness, and Cook’s Theorem –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. Reductions
2. NP-Completeness
3. Cook’s Theorem
What are We Aiming at?

- We suspect some problems in $\text{NP}$ to be significantly harder than each of the problems in $\text{P}$.
- What is significantly harder (or not significantly harder) in this context?
- The most suitable definition of these terms would consider
  1. the difference between problems in $\text{P}$ to be insignificant, but
  2. the difference between problems in and outside $\text{P}$ to be significant.
- The Composition Theorem gives us a strong hint: We can apply any polynomial-time preprocessing to a problem in $\text{P}$ without leaving the realm of polynomial-time algorithms!
- The preprocessing may in particular translate one problem into another.
- Such translations are called polynomial-time reductions.
Polynomial-Time Reductions

Definition (polynomial-time reduction)

Let $A$ and $B$ be decision problems. A polynomial-time reduction from $A$ to $B$ is a function $f$ such that:

1. for every input $x$, $f(x) \in B \iff x \in A$, and
2. $f$ can be computed (deterministically) in polynomial time.

We say that $A$ is polynomial-time reducible to $B$, and write $A \leq_p B$.

Note: $\leq_p$ is a preorder (also called a quasi-order), that is,

- it is transitive ($A \leq_p B \leq_p C$ implies $A \leq_p C$) [WHY?], and
- it is reflexive ($A \leq_p A$) [WHY?],
- but it is not acyclic (we may have $A \leq_p B \leq_p A$ for $A \neq B$).
Insignificance of the Difference Between Problems in P

Does this view “the difference between problems in \( P \) to be insignificant”? We want that a polynomial-time reduction from \( A \) to \( B \) exists if \( A, B \in P \).

Here is such a reduction \( f \): Choose any \( y^+ \in B \) and \( y^- \notin B \), and define

\[
f(x) = \begin{cases} 
y^+ & \text{if } x \in A \\
y^- & \text{otherwise.}
\end{cases}
\]

Clearly, \( f(x) \in B \iff x \in A \), and computing \( f \) takes polynomial time.

Notes:

- Intuitively, the reduction does all the work itself.
- It can do so, because it has all the resources \( A \) requires.
- There are exactly two problems \( B \) for which this does not work. Which ones?
Significance of the Difference Between Problems In and Outside P

**Theorem** (P is (backwards) closed under reductions)

If \( A \leq_p B \) and \( B \in P \) then \( A \in P \).

(In other words, no \( A \notin P \) is polynomial-time reducible to any \( B \in P \).)

**Proof:** Let \( M \) decide \( B \) and let \( M_0 \) compute a reduction \( f \) from \( A \) to \( B \), both in polynomial time.

- \( M' = M \circ M_0 \) runs in polynomial time (Composition Theorem).
- We have \( M'(x) = M(M_0(x)) = M(f(x)) \). Since
  \[
  M(f(x)) = 1 \iff f(x) \in B \iff x \in A,
  \]
  this means that \( M' \) decides \( A \) (in polynomial time).
NP-Completeness

Definition (NP-complete)

A decision problem $A$ is NP-complete if

- $A \in \text{NP}$ and
- $B \leq_p A$ for every problem $B \in \text{NP}$.

Notes:

- If the first condition is dropped, we say that $A$ is NP-hard.
- For classes above NP (such as EXP), completeness is defined similarly.
- For P and classes inside P, we would need another type of reduction (see Slide 6).
Proving that a Problem Is NP-Completeness

Theorem

If there is an NP-complete problem $B$ such that $B \in P$, then $P = NP$.

Proof: Since $B$ is NP-complete, we have $A \leq_p B$ for all $A \in NP$. By the closedness of $P$ under reductions, this means that $A \in P$. 
Proving that a Problem Is NP-Completeness

Lemma

If \( A \in \text{NP} \) and \( B \leq_p A \) for an NP-complete problem \( B \), then \( A \) is NP-complete.

Proof: For every \( C \in \text{NP} \), we have \( C \leq_p B \leq_p A \).

This yields the most common way to prove that \( A \) is NP-complete:

1. show that \( A \in \text{NP} \),
2. choose a suitable NP-complete problem \( B \), so that you manage to find a reduction \( f \) from \( B \) to \( A \) (note the direction!),
3. argue that \( f \) is polynomial-time computable,
4. show that \( f(x) \in A \) if \( x \in B \), and
5. show that \( f(x) \notin A \) if \( x \notin B \).
Cook’s Theorem

For the method on the previous slide to be useful, someone has to find a first NP-complete problem...

Theorem (Cook 1971)

SAT is NP-complete.

The proof...

- shows how to construct a polynomial-time reduction from $A$ to SAT, for arbitrary $A \in \text{NP}$,
- cannot be based on anything else than the existence of a nondeterministic polynomial-time decision algorithm for $A$.

Rather than nRAMs, we use Turing machines (TMs) for the proof sketch.
Proof Sketch of Cook’s Theorem

We use the characterization of NP by polynomially bounded binary relations. Let $M$ be a deterministic TM and $p, q$ polynomials such that

- $M$ accepts only strings of the form $x\#y$, where $y \in \{0, 1\}^{p(n)} (n = |x|)$,
- $x \in A$ if and only if $\exists y \in \{0, 1\}^{p(n)}$ with $M(x\#y) = yes$, and
- $M$ runs at most $q(n)$ steps.
- When $M$ halts (after $q(n)$ steps), its head scans the first tape cell.
- $M$ accepts the input by entering a special state $z_{yes}$.

**Major proof step:** From $x\#y$ we construct, in polynomial time, a Boolean formula (in CNF) that encodes the computation of $M$ with this input.
Proof Sketch of Cook’s Theorem

A computation of $M$ as a $q(n) \times q(n)$-array of symbols:

$\begin{array}{cccccc}
  & a^z_0 & b & b & \cdots & \# & 0 & 1 & \cdots \\
  c & b^z_1 & b & \cdots & \# & 0 & 1 & \cdots \\
  \cdots & \cdots & b & a^z_k & c & \cdots \\
  \cdots & \cdots & b & b & c^z_l & \cdots \\
  b^{z_{\text{yes}}} & \cdots
\end{array}$

- Encode contents of cell $C_{ij}$ by $k$ boolean variables $\overline{x}_{ij} = x_{ij}^1, \ldots, x_{ij}^k$.
- Row 1 should contain the input
  $$\Rightarrow \text{we use formulas } \varphi_b^i \equiv " \overline{x}_{1j} \text{ represents } b".$$  
- Contents of $C_{i+1,j}$ is given by contents of $C_{ij-1}, C_{ij}, C_{ij+1}$
  $$\Rightarrow \text{we use } \varphi_j^i \equiv " \overline{x}_{i+1j} \text{ represents the successor of } \overline{x}_{ij-1}, \overline{x}_{ij}, \overline{x}_{ij+1}".$$  
- Cell $C_{q(n)1}$ must indicate acceptance
  $$\Rightarrow \text{we use } \varphi_{\text{yes}} \equiv " \overline{x}_{q(n)1} \text{ represents a cell containing } z_{\text{yes}}".$$
Proof Sketch of Cook’s Theorem

Computing the reduction for input $x\#y = b_1 \cdots b_m$:

1. Compute $q(n)$.
2. Print “$\varphi_1^{b_1} \land \cdots \land \varphi_{b_m}^{m} \land \varphi_{m+1}^{m} \cdots \varphi_{q(n)}^{m}$”.

For $i = 2, \ldots, q(n)$ and $j = 1, \ldots, q(n)$ print “$\land \varphi_i^j$”.
4. Print “$\land \varphi_{\text{yes}}$”.

⇒ the resulting formula $\varphi$ is satisfiable if and only if $M$ accepts $x\#y$.

But we have only $x$ and want $\varphi$ to be satisfiable if there exists such a $y$!

**Solution:** In Step 2 above, print $\varphi_{b_j}^j$ only for the $b_j$ in $x\#$. Leave the remaining values of variables in row 1 unspecified.
Please read Section 10.3 in the textbook. (It does not contain the proof of Cook’s Theorem.)