COMP3630 / COMP6363

week 7 & 8: **Time Complexity**

This Lecture Covers Chapter 10 of HMU: Time Complexity

slides created by: Dirk Pattinson, based on material by
Peter Hoefner and Rob van Glabbeck; with improvements by Pascal Bercher

convenor & lecturer: Pascal Bercher

The Australian National University

Semester 1, 2023

Content of this Chapter

- NP-Hardness
- Polytime Reductions
- SAT is NP-hard

Additional Reading: Chapter 10 of HMU.

P $\stackrel{?}{=}$ NP

Question 10.1.1 (P = NP problem)

Can we simulate a non-deterministic TM (NTM) in polynomial time on a (deterministic) TM?

Recall:

- **P**—problems that can be solved in polynomial time on a TM.
- **NP**—problems that can be solved in polynomial time on an NTM.

At this point, no one knows for sure, but "no" might be a good bet.

NP-complete problems

This is about decision problems (problems with yes/no answers). Equivalently, solving the membership problem $x \in L$.

Obviously $\mathbf{P} \subseteq \mathbf{NP}$.

Nobody knows for sure whether $\mathbf{NP} \subseteq \mathbf{P}$

Intuitively, **NP**-complete problems are the "hardest" problems in **NP**.

P Reducibility

Definition 10.1.2

$f : \Sigma^* \longrightarrow \Sigma^*$ is a polynomial time-computable (or **P**-computable) function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input $w \in \Sigma^*$.

Definition 10.1.3

$A \subseteq \Sigma_1^*$ is polynomial time mapping-reducible (or **P**-reducible) to $B \subseteq \Sigma_2^*$, written $A \leq_{\mathbf{P}} B$, if a **P**-computable function $f : \Sigma_1^* \longrightarrow \Sigma_2^*$ exists that is also a reduction (from A to B).

Definition 10.1.4

- A reduction is a polynomial-time translation of the problem, say r.
- If w is an instance of problem A, then $r(w)$ is an instance of problem B.
- r must have two properties:
  1. it preserves the answer. So the answer to w is "yes" iff the answer to $r(w)$ is "yes." (The same automatically holds for the "no" due to the "iff".)
  2. $r(w)$ can be computed in time polynomial in $|w|$.

P Reducibility cont.

Theorem 10.1.5

If $A \leq_{\mathbf{P}} B$ *and* $B \in \mathbf{P}$ *then* $A \in \mathbf{P}$*.*

Proof.

To decide $w \in A$ first compute $f(w)$ (in **P**) where f is the **P** reduction from A to B, and then run a **P** decider for B. This is still in **P** because $p_1(p_2(n))$ is a polynomial if $p_1(n)$ and $p_2(n)$ are. □

NP Membership, Hardness, and Completeness

Definition 10.1.6 (NP completeness, NP membership, NP hardness)

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

1. $B \in \mathbf{NP}$ — $=$ **NP** membership
2. every $A \in \mathbf{NP}$ is **P**-reducible to B. — $=$ **NP** hardness

- So from the second property we get $A \leq_{\mathbf{P}} B$ for all A, and therefore we know that B is "hard/expressive enough" to solve all other problems in NP.
- Therefore, **NP**-complete problems are the hardest ones in **NP**.
  (E.g., we probably can't solve other **NP** problems using a **P** problem!)
- Note that if $\mathbf{P} \neq \mathbf{NP}$, there do exist problems, which are in **NP**, not in **P**, but not **NP**-hard! In other words: If $\mathbf{P} \neq \mathbf{NP}$ (so non-determinism can't be compiled away in poly-time), non-membership to P (which implies that we need non-determinism for poly-time) does not imply that a problem is also **NP**-hard (and thus **NP**-complete). (Ladner's theorem, 1975)

Motivation

Why are we interested in showing **NP**-hardness/completeness in the first place?

- If we fail in providing a **P** procedure for a new problem it could be:
  - Because we just did not think hard enough (it exits and we could find it)
  - Somebody else did just not think hard enough (it exists and somebody more fortunate could find it)
  - It doesn't even exist!
- So ... How to find out whether we should just work harder? (Or ask "this friend that's always better/quicker than me"?)
  - If we can prove **NP**-completeness, then at least we know that nobody before you (and possibly long after you) found a **P**-procedure. (And maybe none even exists for it, which follows directly once somebody proves $\mathbf{P} \neq \mathbf{NP}$.)
- Why **NP**-completeness? Why not just showing **NP**-hardness?
  - Since the problem could be even harder! (E.g., **PSPACE** (week 10), **EXPTIME**, **NEXPTIME**, ..., $\mathbf{RE} \setminus \mathbf{R}$ (undecidable), and *infinitely* more!)
  - Each problem class has specific "properties". E.g., "**NP** looks like Logic", "**PSPACE** looks like planning".

NP-Hardness

Theorem 10.1.7

If B is **NP***-hard and $B \leq_{\mathbf{P}} C$, then C is* **NP***-hard.*

Corollary 10.1.8

If B is **NP***-complete and $B \leq_{\mathbf{P}} C$ for $C \in$* **NP***, then C is* **NP***-complete.*

Proof.

Polynomial time reductions compose. □

Important note! Corollary 10.1.8 is of major importance!! Why?

→ It gives us a convenient procedure to show **NP**-completeness!

- First, show **NP**-membership. (That's almost always very easy.)
- Then, show hardness by grabing any **NP**-complete problem and reducing it to yours!

Open issue: We need "a very first" **NP**-complete problem... (Hardness is the issue!)

NP-Hardness by Reduction (Recap!)

Typical method to show **NP**-hardness:

- Reduce a known **NP**-hard problem A to the new problem B (Theorem 10.1.7).
  That is: Take **NP**-hard A from the literature and show $A \leq_{\mathbf{P}} B$, where B is the (new) problem for which you want to show **NP**-hardness.

Why would we want to do so?

- We just had some reasons a few slides back (see our *Motivation* slide!).
- One point is: we know that nobody has found a **P** solution to your problem B yet! (That hopefully makes a good excuse!)

Consequences of **NP**-Completeness

Theorem 10.1.9

If B is **NP***-complete and $B \in$* **P** *then* **P** $=$ **NP***.*

Proof.

Since B is **NP**-hard, by Def. 10.1.6, for every $A \in$ **NP** holds $A \leq_{\mathbf{P}} B$.
Since B is in **P**, and since polynomial time reductions compose, each A is in **P**. □

Question: Did we need **NP**-completeness of B? Would **NP**-hardness have sufficed?
→ Yes! But it's less likely to show $B \in$ **P** if it's not **NP**-complete. (Discuss in tutorials.)

Also:

- All **NP**-complete problems can be translated in deterministic polytime into every other **NP**-complete problem. I.e., all **NP**-complete problems can be reduced to each other.
- So, if there is a **P** solution to one **NP**-complete problem, there is a **P** solution to every **NP** problem. (This can be another "motivation" behind all this.)

Let A be **NP**-complete and $B \in$ **NP**. What can we conclude (at the moment)?

1. $B \leq_{\mathbf{P}} A$? Yes, by definition. Since A is **NP**-hard.
2. $A \leq_{\mathbf{P}} B$? No! Maybe **P** $\neq$ **NP**, and B might be in **P**.
3. $A \leq_{\mathbf{P}} B$ if $B \notin$ **P**? Still no! Maybe **P** $\neq$ **NP**, then Ladner's theorem says that there are non-**NP**-hard problems in **NP** $\setminus$ **P**! (And maybe that's our B.)

Basic Proof Strategy (Another Recap!)

NP-completeness is a good news/bad news situation.

- Good news: The problem is in **NP**! (Why good? It's "not" harder!)
- Bummer: The problem is **NP**-hard! (Why bad? Likely not in **P**...)

So, a typical **NP**-completeness proof consists of two parts:

1. Prove that the problem is in **NP** (i.e., it has **P** verifier – or a non-deterministic TM).
2. Prove that the problem is at least as hard as other problems in **NP**.

A TM can simulate an ordinary computer in polynomial time, so it is sufficient to describe a polynomial-time checking algorithm that will run on any reasonable model of computation. (Recall the pseudocode for gcd! That wasn't a TM either.)

NP-hardness: How (not) to do it

Important warning:

- Make sure you are reducing the known problem to the unknown problem! "Unknown" here means that it's the "new" one that has unknown complexity.
- Recall Corollary 10.1.8: Show $B \leq_{\mathbf{P}} C$ for $C \in \mathbf{NP}$, i.e., C is the unknown problem and B was an **NP**-complete problem. (Any **NP**-hard problem will do for B, but if it's harder than **NP**, you likely won't be able to do the reduction.)
- So, again, carefully double-check that you reduce in the right direction!

In practice, there are now thousands of known **NP**-complete problems.
A great start: "Karp's 21 NP-complete problems" – google it!
(And attend/watch Alban's guest lectures on examples! No slides!)

A good technique is to look for one similar to the one you are trying to prove **NP**-hard.

Making our life easier...

So for **NP**-completeness we need to show **NP**-hardness. For this, we had two options:

1. Use Definition 10.1.6, i.e., show that all problems in **NP** reduce to our problem, or
2. use Theorem 10.1.7, i.e., reduce from an **NP**-hard problem.

So in the first case we need to show a property for all problems, in the second we only need a single reduction... What's easier? :)

So we need a very first problem that's shown to be NP-hard – from then on we can start reducing!

For this, we will use SAT!
(Note that this / the first choice is actually also just a single reduction!)

Boolean Formulae

Let $Prop = \{x, y, \ldots\}$ be a (finite) set of Boolean variables (or propositions).
A CFG for Boolean formulae over *Prop* is:

$$\phi \to p \mid \phi \land \phi \mid \neg\phi \mid (\phi)$$
$$p \to x \mid y \mid \ \ldots$$

We use abbreviations such as

$$\phi_1 \lor \phi_2 = \neg(\neg\phi_1 \land \neg\phi_2) \qquad \phi_1 \Rightarrow \phi_2 = \neg\phi_1 \lor \phi_2$$
$$\text{FALSE} = (x \land \neg x) \qquad \text{TRUE} = \neg\text{FALSE}$$

(Technically, we could handle countably infinite sets *Prop* if we had a naming scheme for variables, say, x_n for binary representations n of natural numbers. We won't need this!)

Semantics of Boolean Formulae

A Boolean formula is either $\top$ (for "true") or $\bot$ (for "false"), possibly depending on the interpretation of its propositions. Let $\mathbb{B} = \{\bot, \top\}$.

Definition 10.2.1

An interpretation (or assignment) of *Prop* is a function $\pi : Prop \longrightarrow \mathbb{B}$.
For Boolean formulae ϕ we define π satisfies ϕ, written $\pi \models \phi$, inductively by:
Base: $\pi \models x$ iff $\pi(x) = \top$.
Induction:

- $\pi \models \neg\phi$ iff $\pi \not\models \phi$.
- $\pi \models \phi_1 \wedge \phi_2$ iff both $\pi \models \phi_1$ and $\pi \models \phi_2$.
- $\pi \models (\phi)$ iff $\pi \models \phi$.

ϕ is satisfiable if there exists an interpretation π such that $\pi \models \phi$.

SAT—An **NP**-Complete Problem

$$SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula } \}$$

Theorem 10.2.2 (Cook-Levin Theorem – or: Cook's Theorem, 1971/1973)

*SAT is **NP**-complete.*

Proof of $SAT \in$ NP.

If $\pi \models \phi$ we use $\langle \pi \rangle$ as certicate. (I.e., guess it and verify.) Had we chosen a countably infinite *Prop*, we'd restrict π to the propositions occurring in ϕ. □

Proof of *SAT* is NP-hard.

The entire rest of these slides! □

Proof of **NP**-Hardness of *SAT*

Let $A \in \mathbf{NP}$. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a deciding NTM with $L(M) = A$ and let p be a polynomial such that M takes at most $p(|w|)$ steps on any computation for any $w \in \Sigma^*$.

Construct a **P** reduction from A to *SAT*:

- Input w is turned into a Boolean formula ϕ_w that describes M's possible computations on w.
- M accepts w iff ϕ_w is satisfiable. The satisfying interpretation resolves the nondeterminism in the computation tree to arrive at an accepting branch of the computation tree.

Remains to be done: define ϕ_w.

Proof of **NP**-Hardness of *SAT* cont.

Recall that M accepts w if an $n \leq p(|w|)$ exists and a sequence of configurations $(C_i)_{0 \leq i \leq n}$ (IDs), where

1. $C_0 = q_0 w$,
2. each C_i can yield C_{i+1}, and
3. C_n is an accepting ID.
4. Note that we have at most $n + 1$ IDs if the TM can make at most $n \leq p(|w|)$ steps.

ϕ_w

The Boolean formula ϕ_w shall represent all such sequences $(C_i)_{0<i\leq n}$ beginning with $q_0 w$.

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

The different sub formulae serve the following purposes:

- ϕ_{cell}: Defines all existing "cells", which encode all possible IDs.
- ϕ_{start}: Sets the initial row of these cells: TM's initial ID.
- ϕ_{move}: Enforces legal TM transitions.
- ϕ_{accept}: Enforces ending up in an accepting state.

ϕ_{cell}

...describes an n^2 grid using propositions $Prop = \{\ x_{i,k,s} \mid i,k \in \{0,\ldots,n\} \land s \in \Sigma_\phi\ \}$, where $\Sigma_\phi = Q \cup \Gamma$ (recall that $B \in \Gamma$) is the "alphabet of the SAT formula" used to encode the IDs. Also recall that TM IDs contain the non-trivial tape and the state.

First, why is $i,k \in \{0,\ldots,n\}$? Why $s \in \Sigma_\phi$?

- i: encode the rows. We need one for every possible ID ($n+1$ many!)
- k: encodes the columns. Each column is a possible value of an ID symbol.
  n symbols are the TM cells that can be reached, and one is the state.
- s: The content of ID i at position k, i.e., a tape symbol or the state.

$$\phi_{\text{cell}} = \bigwedge_{0 \le i,k \le n} \left(\left(\bigvee_{s \in \Sigma_\phi} x_{i,k,s} \right) \land \left(\bigwedge_{s \neq t \in \Sigma_\phi} (\neg x_{i,k,s} \lor \neg x_{i,k,t}) \right) \right)$$

Meaning: "There is exactly one symbol at each cell".

ϕ_{start}

…specifies that the first row of the grid contains $q_0 w$ where $w = w_1 \dots w_{|w|}$:

$$\phi_{\text{start}} = x_{0,0,q_0} \wedge \bigwedge_{1 \le i \le |w|} x_{0,i,w_i} \wedge \bigwedge_{|w| < i \le n} x_{0,i,B}$$

So the first line of our grid contains:

- the q_0 symbol in the first cell,
- followed by the symbols of our initial tape word,
- followed by the blank symbol until the end.

ϕ_{move}

...ensures that C_i yields C_{i+1} by describing legal 2×3 windows of cells. We need 3 cells to cover the cell on the left of the state, the state, and on its right (to enable left and right movements of the head).

$$\phi_{\text{move}} = \bigwedge_{0<i,k<n} \bigvee_{\begin{array}{|c|c|c|}\hline a_1 & a_2 & a_3 \\ \hline a_4 & a_5 & a_6 \\ \hline \end{array} \text{ is legal}} \left(\begin{array}{l} x_{i,k-1,a_1} \wedge x_{i,k,a_2} \wedge x_{i,k+1,a_3} \wedge \\ x_{i+1,k-1,a_4} \wedge x_{i+1,k,a_5} \wedge x_{i+1,k+1,a_6} \end{array} \right)$$

(Some border cases are not be covered here for simplicity, e.g., i can never be zero.)
What is legal depends on the transition function δ.

Example: Let the current ID be $w_1 w_2 q w_3 w_4$ (so we have blanks before and after it). Whether we go to the left or to the right, we only need to change 3 cells!

- $w_1 w_2 q w_3 w_4$ – current ID
- $w_1 w_2 x q_1 w_4$ – if $\delta(q, w_3) = (q_1, x, R)$
- $w_1 q_2 w_2 y w_4$ – if $\delta(q, w_3) = (q_2, y, L)$

Are we still complete?
We can't seem to be able to move to the left of the initial head position!

- Not a problem: We showed equivalence for semi-infinite tapes under polytime.
- We could alternatively have created a grid of size $(2n)^2$, which also goes n to the left.

ϕ_{accept} – and concluding the Proof

...states that the accept state is reached:

$$\phi_{\text{accept}} = \bigvee_{0 \leq i,k \leq n, q_F \in F} x_{i,k,q_F}$$

Concluding the Proof:

Recall:

$$\phi_w = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$$

Finally we check that the size of ϕ_w is polynomial in $|w|$ and that ϕ_w is constructable in polynomial time. (Both is true!)

So finding a valuation to this formula means deciding $w \in L(M)$ for the arbitrary non-deterministic TM M! So SAT is **NP**-hard! (It can express every problem in **NP**!)

We have our patient zero now – so now we can prove **NP**-hardness of other problems by reducing from SAT. (And we build our portfolio...)