P and NP

Lecture in INF4130

Department of Informatics

October 25th, 2018

Recap from Lecture 1

- Undecidability: no Turing Machine decides L
- Proving undecidability
  - First, we proved that the Halting problem is undecidable
  - Later, we proved more undecidability-results via *reductions*
- Our Turing machines had no resource-restrictions. We were interested in deciders, halting in a finite number of steps.

Today

- Not all deciders are useful in practice. Some take to much time, or use too much memory.
- We will introduce resource-limitations.
  - Time: How many steps does a decider for L use?
  - Space: How many tape-cells does a decider for L use? (Not a part of our focus in this course.)

Today

- Not all deciders are useful in practice. Some take to much time, or use too much memory.
- We will introduce resource-limitations.
  - Time: How many steps does a decider for L use?
  - Space: How many tape-cells does a decider for L use? (Not a part of our focus in this course.)
- We will define *complexity classes*, classes of problems of similar difficulty.
  - P: Class of languages decidable in *polynomial time* on a *deterministic* Turing machine.
  - NP: Class of languages decidable in *polynomial time* on a *nondeterministic* Turing machine.

Today

- Not all deciders are useful in practice. Some take to much time, or use too much memory.
- We will introduce resource-limitations.
  - Time: How many steps does a decider for L use?
  - Space: How many tape-cells does a decider for L use? (Not a part of our focus in this course.)
- We will define *complexity classes*, classes of problems of similar difficulty.
  - P: Class of languages decidable in *polynomial time* on a *deterministic* Turing machine.
  - NP: Class of languages decidable in *polynomial time* on a *nondeterministic* Turing machine.
  - (P: Class of problems where solutions can be found quickly.)
  - (NP: Class of problems where solutions can be checked quickly.)

Up to now, we have used *deterministic* Turing machines. We will need the notion of *nondeterminism*.

Up to now, we have used *deterministic* Turing machines. We will need the notion of *nondeterminism*.

Definition (Nondeterministic Turing machines)

A *Nondeterministic Turing machine* (NTM) is like a deterministic Turing machine (DTM), but where the transition-function of a DTM only returned one next step, the transition function of a NTM returns a set of possible next steps. The computation of a DTM, can be viewed as a straight line of configurations following each other, ending in an accept/reject state, or continuing for ever (if the DTM loops). For a NTM, the computation will look like a tree of configurations. If any branch of the computation-tree accepts, the NTM accepts its input.

Example (*HAMPATH*)

Let $HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$.

Example (*HAMPATH*)

Let $HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$.
N is a NTM that decides *HAMPATH*.
$N =$ "On input $\langle G, s, t\rangle$:

(1) Write down a list of $|G|$ nodes from G. (Nondeterministically guess a path)

(2) If the list contains repetitions, *reject*.

(3) Check that the first node in the list is s and that the last is t. If not, *reject*

(4) For each pair of nodes (a, b) in the list where a occurs right before b, check that (a, b) is an edge in G. If not, *reject*

(5) All tests have been passes, *accept*."

Time-complexity

Definition (Time-complexity)

Let M be a deterministic decider (DTM that halts on all inputs). The *running time* or *time complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps M uses on any input of length n. We say that M runs in time $f(n)$. We will usually use n as the length of the input, $|w|$.

Time-complexity

Definition (Time-complexity)

Let M be a deterministic decider (DTM that halts on all inputs). The *running time* or *time complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps M uses on any input of length n. We say that M runs in time $f(n)$. We will usually use n as the length of the input, $|w|$.

Notice that this gives us the *worst case* running time of M. Here $f(n)$ equals the number of steps M uses on the "hardest" instance of length n.
Typically, we are not interested in *exact* running times. We will use *big-O notation* to abstract away minor differences in running time.

Definition (The TIME classes)

Let $TIME(t(n))$ be a time complexity class containing all languages decidable by a deterministic Turing machine running in $O(t(n))$ time.

Definition (The TIME classes)

Let $TIME(t(n))$ be a time complexity class containing all languages decidable by a deterministic Turing machine running in $O(t(n))$ time.

Example ($TIME(n^2)$)

$TIME(n^2)$ is the class containing all languages that can be decided in $O(n^2)$ time.

Definition (The TIME classes)

Let $TIME(t(n))$ be a time complexity class containing all languages decidable by a deterministic Turing machine running in $O(t(n))$ time.

Example ($TIME(n^2)$)

$TIME(n^2)$ is the class containing all languages that can be decided in $O(n^2)$ time.

Example ($L \in TIME(n^2)$?)

How do we show $L \in TIME(n^2)$? Well, that is "easy". Just show a DTM that decides L in time $O(n^2)$. Okay, but how do we show that $L \notin TIME(n^2)$? Must somehow show that no TM decides L quickly enough. This is challenging.

Now we can define a very important complexity class.

Now we can define a very important complexity class.

Definition (P (polynomial time))

Let $P = \bigcup_{k \in \mathbb{N}} TIME(n^k)$. P is the class of languages decidable in polynomial time on a deterministic Turing machine.

Example ($PATH$)

Let $PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a directed path from s to $t\ \}$.
Turns out $PATH \in P$.
M = "On input $\langle G, s, t\rangle$:

(1) Mark node s.

(2) Repeat until no new nodes are marked:

(3) Look at the edges of G. If an edge (a, b) is found where a is marked and b is unmarked, mark b.

(4) If t is marked, *accept*. If not, *reject*"

Some problems known to be in P

- Given a graph, confirm that it is connected.
- Given a set of natural numbers and a goal-number t, can you pick out three elements summing to t?
- *PRIMES* from the previous lecture (not obvious!).
- Given binary natural numbers a,b and c, does $a * b = c$?

Polynomials are robust

- P is closed under complement.
- Polynomials are closed under addition.
  - P is closed under intersection.
  - P is closed under union.
- Polynomials are closed under multiplication.
  - P is closed under concatenation.
- *All reasonable deterministic computational models are polynomially equivalent.*

Polynomial time reductions

Mapping reductions let us translate between instances of different problems. Now we want to be able to do the same, only this time, the translation itself must be efficient.

Polynomial time reductions

Mapping reductions let us translate between instances of different problems. Now we want to be able to do the same, only this time, the translation itself must be efficient.

Definition (Polynomial functions)

A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial time computable function* if some polynomial time Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Polynomial time reductions

Mapping reductions let us translate between instances of different problems. Now we want to be able to do the same, only this time, the translation itself must be efficient.

Definition (Polynomial functions)

A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial time computable function* if some polynomial time Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Definition (Polynomial reductions)

Language A is *polynomial time reducible* to language B, written $A \leq_P B$, if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$, where for every w:
$w \in A \leftrightarrow f(w) \in B$. The function f is called the polynomial (time) reduction from A to B.

Polynomial time reductions

Mapping reductions let us translate between instances of different problems. Now we want to be able to do the same, only this time, the translation itself must be efficient.

Definition (Polynomial functions)

A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial time computable function* if some polynomial time Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Definition (Polynomial reductions)

Language A is *polynomial time reducible* to language B, written $A \leq_P B$, if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$, where for every w:
$w \in A \leftrightarrow f(w) \in B$. The function f is called the polynomial (time) reduction from A to B.

With this definition, we know that if $A \leq_P B$, then we can efficiently translate instances of A to instances of B.

Theorem

If $A \leq_P B$, and $B \in P$, then $A \in P$.

Theorem

If $A \leq_P B$, and $B \in P$, then $A \in P$.

Proof

Let M_B be the polynomial time algorithm for deciding membership in B, and let f be the polynomial reduction from A to B. We construct M_A which decides A in polynomial time as follows:

$M_A =$ "On input w:

(1) Compute $f(w)$.

(2) Run M_B on input $f(w)$

(3) If M_B accepts, *accept*.

(4) If M_B rejects, *reject*."

M_A gives the correct answer, since f is a reduction from A to B. Furthermore, M_A runs in polynomial time, since the all steps take polynomial time. Note that the composition of two polynomials is a polynomial. □

Definition (Running time of NTMs)

Let N be a nondeterministic decider (NTM that halts on all inputs). The *running time* or *time complexity* of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps N uses **on any branch of its computation** on any input of length n. We say that N runs in time $f(n)$.

Definition (Running time of NTMs)

Let N be a nondeterministic decider (NTM that halts on all inputs). The *running time* or *time complexity* of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps N uses **on any branch of its computation** on any input of length n. We say that N runs in time $f(n)$.

Definition (The NTIME classes)

Let $NTIME(t(n))$ be a time complexity class containing all languages decidable by a nondeterministic Turing machine running in $O(t(n))$ time.

Definition (Running time of NTMs)

Let N be a nondeterministic decider (NTM that halts on all inputs). The *running time* or *time complexity* of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps N uses **on any branch of its computation** on any input of length n. We say that N runs in time $f(n)$.

Definition (The NTIME classes)

Let $NTIME(t(n))$ be a time complexity class containing all languages decidable by a nondeterministic Turing machine running in $O(t(n))$ time.

Definition (NP (nondeterministic polynomial time))

Let $NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$. NP is the class of languages decidable in polynomial time on a nondeterministic Turing machine.

Definition (Running time of NTMs)

Let N be a nondeterministic decider (NTM that halts on all inputs). The *running time* or *time complexity* of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps N uses **on any branch of its computation** on any input of length n. We say that N runs in time $f(n)$.

Definition (The NTIME classes)

Let $NTIME(t(n))$ be a time complexity class containing all languages decidable by a nondeterministic Turing machine running in $O(t(n))$ time.

Definition (NP (nondeterministic polynomial time))

Let $NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$. NP is the class of languages decidable in polynomial time on a nondeterministic Turing machine.

We will look at a second (possibly more useful) definition of NP soon, but first a familiar example.

Example (*HAMPATH*)

Let $HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$.
$HAMPATH \in NP$.
$N =$ "On input $\langle G, s, t\rangle$:

(1) Write down a list of $|G|$ nodes from G. (Nondeterministically guess a path)
(2) If the list contains repetitions, *reject*.
(3) Check that the first node in the list is s and that the last is t. If not, *reject*
(4) For each pair of nodes (a, b) in the list where a occurs right before b, check that (a, b) is an edge in G. If not, *reject*
(5) All tests have been passes, *accept*."

Example (*HAMPATH*)

Let $HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$.
$HAMPATH \in NP$.
$N =$ "On input $\langle G, s, t\rangle$:

(1) Write down a list of $|G|$ nodes from G. (Nondeterministically guess a path)
(2) If the list contains repetitions, *reject*.
(3) Check that the first node in the list is s and that the last is t. If not, *reject*
(4) For each pair of nodes (a, b) in the list where a occurs right before b, check that (a, b) is an edge in G. If not, *reject*
(5) All tests have been passes, *accept*."

This is **not** how we typically show membership in *NP*.

Definition (Verifiers and Certificates)

A *verifier* for language L is an algorithm V, where
$L = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$.
We say that c is a *certificate* for membership in L. We call V a *polynomial time verifier* if it runs in polynomial time with respect to the length of w. Note that the length of c is not mentioned. So if V is a polynomial time verifier, then the certificate must be polynomial (in the length of w) since if it was longer, V would not have time to read it.

Definition (Verifiers and Certificates)

A *verifier* for language L is an algorithm V, where
$L = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$.
We say that c is a *certificate* for membership in L. We call V a *polynomial time verifier* if it runs in polynomial time with respect to the length of w. Note that the length of c is not mentioned. So if V is a polynomial time verifier, then the certificate must be polynomial (in the length of w) since if it was longer, V would not have time to read it.

Definition (NP (nondeterministic polynomial time))

NP is the class of languages that have polynomial time verifiers.

Definition (Verifiers and Certificates)

A *verifier* for language L is an algorithm V, where
$L = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$.
We say that c is a *certificate* for membership in L. We call V a *polynomial time verifier* if it runs in polynomial time with respect to the length of w. Note that the length of c is not mentioned. So if V is a polynomial time verifier, then the certificate must be polynomial (in the length of w) since if it was longer, V would not have time to read it.

Definition (NP (nondeterministic polynomial time))

NP is the class of languages that have polynomial time verifiers.

Why?

Think certificats, not verifiers!

A verifier can be view as a simple check, that checks that $w \in L$ given the "membership proof" c. Typically, creating V is easy if we know that we have certificates.

Think certificats, not verifiers!

A verifier can be view as a simple check, that checks that $w \in L$ given the "membership proof" c. Typically, creating V is easy if we know that we have certificates.
Can you think of certificates for yes-instances of *HAMPATH*?

Think certificats, not verifiers!

A verifier can be view as a simple check, that checks that $w \in L$ given the "membership proof" c. Typically, creating V is easy if we know that we have certificates.
Can you think of certificates for yes-instances of *HAMPATH*?
Informally: $L \in NP$ if L has "short" certificates for the yes-instances. By short we mean polynomial in n.

Comparing the two definitions of NP

If L has a polynomial time verifier V (running in time n^k), then we can construct a NTM deciding L as follows:
$N =$ "On input w of langth n:

(1) Nondeterministically select a string c of length at most n^k.

(2) Run V on input $\langle w, c \rangle$.

(3) If V accepts, *accept*, otherwise, *reject*."

If we instead have a NTM N deciding L, we could make a verifier V which simulated N using the certificate to chose which branch to simulate at each split. In this case, the certificate would be the accepting branch of the computation of N on w. Since N uses polynomial time, c would also be polynomial.

Examples of languages in *NP*

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

Examples of languages in *NP*

- $SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$.
- $SUBSET - SUM = \{\langle S, t \rangle \mid S \text{ is a multiset of integers, such that there exists a subset of } S \text{ summing to } t\}$.

Examples of languages in *NP*

- $SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$.
- $SUBSET - SUM = \{\langle S, t \rangle \mid S \text{ is a multiset of integers, such that there exists a subset of } S \text{ summing to } t\}$.
- $CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a clique of size } k\}$

Examples of languages in NP

- $SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$.
- $SUBSET - SUM = \{\langle S, t \rangle \mid S \text{ is a multiset of integers, such that there exists a subset of } S \text{ summing to } t\}$.
- $CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a clique of size } k\}$
- $PARTITION = \{\langle S \rangle \mid S \text{ is a multiset of natural numbers, such that there exists a subset } S' \text{ of } S \text{ that sums to exactly half the sum of } S\}$
  (Can S be partitioned into two parts summing to the same?)

Theorem

$P \subseteq NP$.

Theorem

$P \subseteq NP$.

Proof

Since every DTM can be viewed as a NTM which does not branch, every language L in P, has a NTM which decides L in polynomial time. □

P vs NP

So we know that every member of P is a member of NP, but is the opposite true?

P vs NP

So we know that every member of P is a member of NP, but is the opposite true?
We can simulate a NTM with a DTM, but our best approach gives an exponential growth in running time. Intuition: We need to simulate each branch, but even with a branching-factor of 2, the whole tree would have size $2^{\text{running time of the NTM}}$.

P vs NP

So we know that every member of P is a member of NP, but is the opposite true?
We can simulate a NTM with a DTM, but our best approach gives an exponential growth in running time. Intuition: We need to simulate each branch, but even with a branching-factor of 2, the whole tree would have size $2^{\text{running time of the NTM}}$.

The Millennium Prize Problems (from Wikipedia)

- Hodge conjecture
- Riemann hypothesis
- **P versus NP**
- Poincaré conjecture **(solved)**
- Yang–Mills existence and mass gap
- Navier–Stokes existence and smoothness
- Birch and Swinnerton-Dyer conjecture