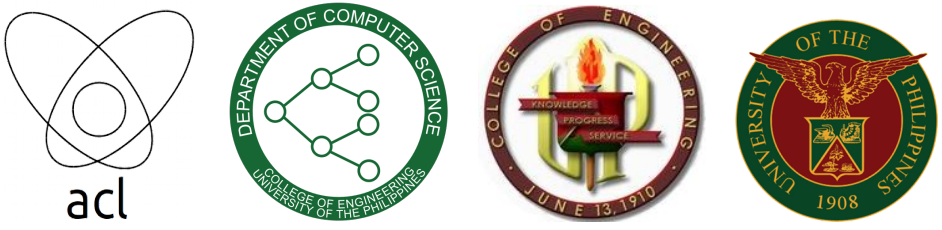
CS 133 : Automata Theory and Computability

Lecture Slides

Time Complexity

Francis George C. Cabarle
fccabarle@up.edu.ph



Algorithms and Complexity Laboratory
Department of Computer Science
College of Engineering
University of the Philippines Diliman

Day 25

Classes **P** and **NP**

NP-completeness

Classes **P** and **NP**

NP-completeness

Class P

or problems solvable in polynomial time with a 1-tape DTM. More formally

Definition

Language L is in class **P** if there is a DTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

Some problems in **P**:

Class P

or problems solvable in polynomial time with a 1-tape DTM. More formally

Definition

Language L is in class **P** if there is a DTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

Some problems in **P**:

$MWST \in \mathbf{P}$. (Use Kruskal's algorithm)

$PATH \in \mathbf{P}$. (Use BFS)

$RELPRIME \in \mathbf{P}$. (Use Eulerian algorithm for gcd)

Class P

or problems solvable in polynomial time with a 1-tape DTM. More formally

Definition

Language L is in class **P** if there is a DTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

Some problems in **P**:

$MWST \in \mathbf{P}$. (Use Kruskal's algorithm)

$PATH \in \mathbf{P}$. (Use BFS)

$RELPRIME \in \mathbf{P}$. (Use Eulerian algorithm for gcd)

Problems in **P** have "fast" or "efficient" algo.

Class NP

A problem in **NP**:
Hamiltonian path in a directed graph (i.e. digraph) is a directed path that goes through each node exactly once.

Class NP

A problem in **NP**:
Hamiltonian path in a directed graph (i.e. digraph) is a directed path that goes through each node exactly once.
The problem (or language) of testing if a given digraph G has a Hamiltonian path given two nodes s and t in G:

Class NP

A problem in **NP**:
Hamiltonian path in a directed graph (i.e. digraph) is a directed path that goes through each node exactly once.
The problem (or language) of testing if a given digraph G has a Hamiltonian path given two nodes s and t in G:

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

Class NP

A problem in **NP**:
Hamiltonian path in a directed graph (i.e. digraph) is a directed path that goes through each node exactly once.
The problem (or language) of testing if a given digraph G has a Hamiltonian path given two nodes s and t in G:

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

Exp time algo? Easy:

Class NP

A problem in **NP**:
Hamiltonian path in a directed graph (i.e. digraph) is a directed path that goes through each node exactly once.
The problem (or language) of testing if a given digraph G has a Hamiltonian path given two nodes s and t in G:

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

Exp time algo? Easy: Use and modify brute-force algo for $PATH$ and check if path is Hamiltonian.

Class NP

A problem in **NP**:
Hamiltonian path in a directed graph (i.e. digraph) is a directed path that goes through each node exactly once.
The problem (or language) of testing if a given digraph G has a Hamiltonian path given two nodes s and t in G:

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

Exp time algo? Easy: Use and modify brute-force algo for $PATH$ and check if path is Hamiltonian.

Hamiltonian cycle problem also popularly known as the *traveling salesman* (or *salesperson*) *problem*.

Class NP

A problem in **NP**:
Hamiltonian path in a directed graph (i.e. digraph) is a directed path that goes through each node exactly once.
The problem (or language) of testing if a given digraph G has a Hamiltonian path given two nodes s and t in G:

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

Exp time algo? Easy: Use and modify brute-force algo for $PATH$ and check if path is Hamiltonian.

Hamiltonian cycle problem also popularly known as the *traveling salesman* (or *salesperson*) *problem*.

Applications: planning, logistics, microchip manufacturing, DNA sequencing, etc.

Class NP

or problems solvable in polynomial time with a 1-tape NTM. More formally

Class NP

or problems solvable in polynomial time with a 1-tape NTM. More formally

Definition

Language L is in class **NP** if there is a NTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

Class NP

or problems solvable in polynomial time with a 1-tape NTM. More formally

Definition

Language L is in class **NP** if there is a NTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

In some sources (e.g. book by M. Sipser) class **NP** is defined as class of problems with poly time **verifiers**:

Class NP

or problems solvable in polynomial time with a 1-tape NTM. More formally

Definition

Language L is in class **NP** if there is a NTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

In some sources (e.g. book by M. Sipser) class **NP** is defined as class of problems with poly time **verifiers**: a verifier is an algo with poly run time and verifies if a given *witness* or *certificate* is a solution to a problem in **NP**.

Class NP

or problems solvable in polynomial time with a 1-tape NTM. More formally

Definition

Language L is in class **NP** if there is a NTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

In some sources (e.g. book by M. Sipser) class **NP** is defined as class of problems with poly time **verifiers**: a verifier is an algo with poly run time and verifies if a given *witness* or *certificate* is a solution to a problem in **NP**.

e.g. witness of $HAMPATH$ is a Hamiltonian path from s to t.

Class NP

or problems solvable in polynomial time with a 1-tape NTM. More formally

Definition

Language L is in class **NP** if there is a NTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

In some sources (e.g. book by M. Sipser) class **NP** is defined as class of problems with poly time **verifiers**: a verifier is an algo with poly run time and verifies if a given *witness* or *certificate* is a solution to a problem in **NP**.

e.g. witness of $HAMPATH$ is a Hamiltonian path from s to t.

No one knows if $HAMPATH$ is solvable in poly time by an algo (i.e. by a DTM)!

Class NP

or problems solvable in polynomial time with a 1-tape NTM. More formally

Definition

Language L is in class **NP** if there is a NTM M such that $L = L(M)$ and M has polynomial run time $t(n)$.

In some sources (e.g. book by M. Sipser) class **NP** is defined as class of problems with poly time **verifiers**: a verifier is an algo with poly run time and verifies if a given *witness* or *certificate* is a solution to a problem in **NP**.

e.g. witness of $HAMPATH$ is a Hamiltonian path from s to t.

No one knows if $HAMPATH$ is solvable in poly time by an algo (i.e. by a DTM)!

Problems in **NP** have "slow" or "inefficient" algo.

Deciding $HAMPATH$

The following NTM decides $HAMPATH$ in nondeterministic polynomial time.

N_1 = "On input $\langle G, s, t \rangle$ where G is a digraph with nodes s and t:

1. Nondeterministically choose a list of m numbers $p_1, \ldots, p_m$, where m is the number of nodes in G and $1 \leq p_i \leq m$.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. $\forall i, 1 \leq i \leq m-1$, check if (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have passed, so accept."

Deciding $HAMPATH$

The following NTM decides $HAMPATH$ in nondeterministic polynomial time.

N_1 = "On input $\langle G, s, t \rangle$ where G is a digraph with nodes s and t:

1. Nondeterministically choose a list of m numbers $p_1, \ldots, p_m$, where m is the number of nodes in G and $1 \leq p_i \leq m$.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. $\forall i, 1 \leq i \leq m-1$, check if (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have passed, so accept."

Poly run time?

Deciding $HAMPATH$

The following NTM decides $HAMPATH$ in nondeterministic polynomial time.

N_1 = "On input $\langle G, s, t \rangle$ where G is a digraph with nodes s and t:

1. Nondeterministically choose a list of m numbers $p_1, \ldots, p_m$, where m is the number of nodes in G and $1 \leq p_i \leq m$.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. $\forall i, 1 \leq i \leq m-1$, check if (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have passed, so accept."

Poly run time? Yup.
Step 1: poly time nondeterministic selection of m numbers.

Deciding $HAMPATH$

The following NTM decides $HAMPATH$ in nondeterministic polynomial time.

N_1 = "On input $\langle G, s, t \rangle$ where G is a digraph with nodes s and t:

1. Nondeterministically choose a list of m numbers $p_1, \ldots, p_m$, where m is the number of nodes in G and $1 \leq p_i \leq m$.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. $\forall i, 1 \leq i \leq m-1$, check if (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have passed, so accept."

Poly run time? Yup.
Step 1: poly time nondeterministic selection of m numbers.
Step 2 to 4: Easy to check! So, poly time!

Deciding $HAMPATH$

The following NTM decides $HAMPATH$ in nondeterministic polynomial time.

N_1 = "On input $\langle G, s, t \rangle$ where G is a digraph with nodes s and t:

1. Nondeterministically choose a list of m numbers $p_1, \ldots, p_m$, where m is the number of nodes in G and $1 \leq p_i \leq m$.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. $\forall i, 1 \leq i \leq m-1$, check if (p_i, p_{i+1}) is an edge of G. If any are not, reject. Otherwise, all tests have passed, so accept."

Poly run time? Yup.
Step 1: poly time nondeterministic selection of m numbers.
Step 2 to 4: Easy to check! So, poly time!
What about $\overline{HAMPATH}$?

Other Problems in NP

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$$

Other Problems in NP

$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

NTM that decides $CLIQUE$:
$N =$ "On input string $\langle G, k\rangle$, where G is a graph:

1. Nondeterministically select a subset C of k nodes of G.
2. Test whether G contains all edges connecting nodes in C.
3. If yes, accept. Otherwise, reject."

Other Problems in NP

$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

NTM that decides $CLIQUE$:
$N =$ "On input string $\langle G, k\rangle$, where G is a graph:

1. Nondeterministically select a subset C of k nodes of G.
2. Test whether G contains all edges connecting nodes in C.
3. If yes, accept. Otherwise, reject."

Step 1 here is basically the "choose m numbers..." step (Step 1) of the NTM for $HAMPATH$, i.e.

Other Problems in NP

$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

NTM that decides $CLIQUE$:
$N =$ "On input string $\langle G, k\rangle$, where G is a graph:

1. Nondeterministically select a subset C of k nodes of G.
2. Test whether G contains all edges connecting nodes in C.
3. If yes, accept. Otherwise, reject."

Step 1 here is basically the "choose m numbers..." step (Step 1) of the NTM for $HAMPATH$, i.e.
We use nondeterminism to *guess* a witness or a possibly correct solution to the problem in **NP**.

Other Problems in NP

$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $B \subseteq S$, we have $\sum_{b \in B} b = t\}$.

Other Problems in NP

$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $B \subseteq S$, we have $\sum_{b \in B} b = t\}$.

Note: Collections S and B are **multisets**, so repetition is allowed.

Other Problems in NP

$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $B \subseteq S$, we have $\sum_{b \in B} b = t\}$.

Note: Collections S and B are **multisets**, so repetition is allowed.

NTM that decides $SUBSETSUM$:
$N =$ "On input string $\langle S, t\rangle$:

1. Nondeterministically select a subset B of the numbers in S.
2. Test whether B is a collection of numbers that sum to t.
3. If yes, accept. Otherwise, reject."

Other Problems in NP

$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $B \subseteq S$, we have $\sum_{b \in B} b = t\}$.

Note: Collections S and B are **multisets**, so repetition is allowed.

NTM that decides $SUBSETSUM$:
$N =$ "On input string $\langle S, t\rangle$:

1. Nondeterministically select a subset B of the numbers in S.
2. Test whether B is a collection of numbers that sum to t.
3. If yes, accept. Otherwise, reject."

Again, step 1 here is basically the "choose m numbers..." step (Step 1) of the NTM for $HAMPATH$ ...

Other Problems in NP

$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $B \subseteq S$, we have $\sum_{b \in B} b = t\}$.

Note: Collections S and B are **multisets**, so repetition is allowed.

NTM that decides $SUBSETSUM$:
$N =$ "On input string $\langle S, t\rangle$:

1. Nondeterministically select a subset B of the numbers in S.
2. Test whether B is a collection of numbers that sum to t.
3. If yes, accept. Otherwise, reject."

Again, step 1 here is basically the "choose m numbers..." step (Step 1) of the NTM for $HAMPATH$ ... are you sensing any pattern here for NTMs of problems in **NP**?

Other Problems in NP

$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_k\}$ and for some $B \subseteq S$, we have $\sum_{b \in B} b = t\}$.

Note: Collections S and B are **multisets**, so repetition is allowed.

NTM that decides $SUBSETSUM$:
$N =$ "On input string $\langle S, t\rangle$:

1. Nondeterministically select a subset B of the numbers in S.
2. Test whether B is a collection of numbers that sum to t.
3. If yes, accept. Otherwise, reject."

Again, step 1 here is basically the "choose m numbers..." step (Step 1) of the NTM for $HAMPATH$ ... are you sensing any pattern here for NTMs of problems in **NP**?

The problem with problems in NP

The first steps in solving problems in **NP** involve guessing (via nondeterminism)!

The problem with problems in NP

The first steps in solving problems in **NP** involve guessing (via nondeterminism)! Why???

The problem with problems in NP

The first steps in solving problems in **NP** involve guessing (via nondeterminism)! Why???
Again, because nobody has found fast/efficient/poly run time algo for problems in **NP** so

The problem with problems in NP

The first steps in solving problems in **NP** involve guessing (via nondeterminism)! Why???
Again, because nobody has found fast/efficient/poly run time algo for problems in **NP** so all we can do is *exhaustive search* (i.e. brute-force) with exp time algos.

The problem with problems in NP

The first steps in solving problems in **NP** involve guessing (via nondeterminism)! Why???
Again, because nobody has found fast/efficient/poly run time algo for problems in **NP** so all we can do is *exhaustive search* (i.e. brute-force) with exp time algos.
One popular way to go around this exp time is via approximation using *heuristic algorithms*:

The problem with problems in NP

The first steps in solving problems in **NP** involve guessing (via nondeterminism)! Why???
Again, because nobody has found fast/efficient/poly run time algo for problems in **NP** so all we can do is *exhaustive search* (i.e. brute-force) with exp time algos.
One popular way to go around this exp time is via approximation using *heuristic algorithms*: such algos run relatively fast (compared to *exact algorithms*), and give "good enough" (i.e. not necessarily exact) solution.

The problem with problems in NP

The first steps in solving problems in **NP** involve guessing (via nondeterminism)! Why???
Again, because nobody has found fast/efficient/poly run time algo for problems in **NP** so all we can do is *exhaustive search* (i.e. brute-force) with exp time algos.
One popular way to go around this exp time is via approximation using *heuristic algorithms*: such algos run relatively fast (compared to *exact algorithms*), and give "good enough" (i.e. not necessarily exact) solution.
Despite this, exact algos and heuristic algos can still take inordinate amount of run times in general.

P vs NP

P vs **NP** problem restated:[1]

[1] More philosophically this time.

P vs NP

P vs **NP** problem restated:[1] Is it really harder to verify a candidate solution than to solve (find a solution to) the problem?

[1] More philosophically this time.

P vs NP

P vs **NP** problem restated:[1] Is it really harder to verify a candidate solution than to solve (find a solution to) the problem?

If $\mathbf{P} = \mathbf{NP}$ then solving a problem is just as easy (in poly time) as checking a candidate solution for the problem.

[1] More philosophically this time.

P vs NP

P vs **NP** problem restated:[1] Is it really harder to verify a candidate solution than to solve (find a solution to) the problem?

If $\mathbf{P} = \mathbf{NP}$ then solving a problem is just as easy (in poly time) as checking a candidate solution for the problem.

If $\mathbf{P} \neq \mathbf{NP}$, this means proving *no fast algorithm exists* to replace brute-force search .

[1] More philosophically this time.

Classes **P** and **NP**

NP-completeness

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e.

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e. If a poly time algo exists for any one of these problems, then all problems in **NP** are poly time solvable!

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e. If a poly time algo exists for any one of these problems, then all problems in **NP** are poly time solvable!
Such problems are called **NP**-*complete* problems.

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e. If a poly time algo exists for any one of these problems, then all problems in **NP** are poly time solvable!
Such problems are called **NP**-*complete* problems. Practical and theoretical importance of this phenomena?

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e. If a poly time algo exists for any one of these problems, then all problems in **NP** are poly time solvable! Such problems are called **NP**-*complete* problems. Practical and theoretical importance of this phenomena?

- theoretical: attempting to prove that $\mathbf{P} = \mathbf{NP}$ only needs to find one poly time algo for any **NP**-complete problem to achieve this goal.

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e. If a poly time algo exists for any one of these problems, then all problems in **NP** are poly time solvable! Such problems are called **NP**-*complete* problems. Practical and theoretical importance of this phenomena?

- theoretical: attempting to prove that $\mathbf{P} = \mathbf{NP}$ only needs to find one poly time algo for any **NP**-complete problem to achieve this goal.
- practical: May prevent wasting time searching for a nonexistent poly time algo to solve a general problem.

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e. If a poly time algo exists for any one of these problems, then all problems in **NP** are poly time solvable! Such problems are called **NP**-*complete* problems. Practical and theoretical importance of this phenomena?

- theoretical: attempting to prove that $\mathbf{P} = \mathbf{NP}$ only needs to find one poly time algo for any **NP**-complete problem to achieve this goal.
- practical: May prevent wasting time searching for a nonexistent poly time algo to solve a general problem.

The phenomenon known as NP-completeness

Important advance on the **P** vs **NP** question was in 1971, with work from Stephen Cook and Leonid Levin:
Discovery of certain **NP** problems whose individual complexity is related to the entire class **NP**, i.e. If a poly time algo exists for any one of these problems, then all problems in **NP** are poly time solvable! Such problems are called **NP**-*complete* problems. Practical and theoretical importance of this phenomena?

- theoretical: attempting to prove that $\mathbf{P} = \mathbf{NP}$ only needs to find one poly time algo for any **NP**-complete problem to achieve this goal.
- practical: May prevent wasting time searching for a nonexistent poly time algo to solve a general problem.
  Note: book author Michael Sipser believes that $\mathbf{P} \neq \mathbf{NP}$*, so proving a problem is* ***NP****-complete is evidence of its nonpolynomiality.*

Satisfiability Problem

A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

Satisfiability Problem

A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

Example

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

Satisfiability Problem

A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

Example

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

The satisfiability problem is to test whether a Boolean formula is satisfiable.

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

Satisfiability Problem

A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

Example

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

The satisfiability problem is to test whether a Boolean formula is satisfiable.

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

Theorem (Cook-Levin Theorem)

$$SAT \in \mathbf{P} \ \textit{iff} \ \mathbf{P} = \mathbf{NP}.$$

Satisfiability Problem

A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

Example

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

The satisfiability problem is to test whether a Boolean formula is satisfiable.

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

Theorem (Cook-Levin Theorem)

$$SAT \in \mathbf{P} \ \textit{iff} \ \mathbf{P} = \mathbf{NP}.$$

SAT was the first problem to be identified to be **NP**-complete.
Method to prove **NP**-completeness?

Polynomial Time Reduction

In computability theory we used reduction to show undecidability (without considering efficiency of reduction).

Polynomial Time Reduction

In computability theory we used reduction to show undecidability (without considering efficiency of reduction).
Considering efficiency of reduction: If problem A is **efficiently reducible** to problem B, an efficient solution to B can be used to solve A efficiently.

Polynomial Time Reduction

In computability theory we used reduction to show undecidability (without considering efficiency of reduction).
Considering efficiency of reduction: If problem A is **efficiently reducible** to problem B, an efficient solution to B can be used to solve A efficiently.

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a **poly time computable function** if some poly time TM M exists and halts with just $f(w)$ on its tape, given any input w.

Polynomial Time Reduction

In computability theory we used reduction to show undecidability (without considering efficiency of reduction).
Considering efficiency of reduction: If problem A is **efficiently reducible** to problem B, an efficient solution to B can be used to solve A efficiently.

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a **poly time computable function** if some poly time TM M exists and halts with just $f(w)$ on its tape, given any input w.

i.e. M transforms w to $f(w)$ in poly time.

Polynomial Time Reduction

In computability theory we used reduction to show undecidability (without considering efficiency of reduction).
Considering efficiency of reduction: If problem A is **efficiently reducible** to problem B, an efficient solution to B can be used to solve A efficiently.

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a **poly time computable function** if some poly time TM M exists and halts with just $f(w)$ on its tape, given any input w.

i.e. M transforms w to $f(w)$ in poly time.

Definition

Language A is **poly time mapping reducible**, or simply **poly time reducible**, to language B, written $A \leq_P B$, if a poly time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w, $w \in A \iff f(w) \in B$.

Polynomial Time Reduction

In computability theory we used reduction to show undecidability (without considering efficiency of reduction).
Considering efficiency of reduction: If problem A is **efficiently reducible** to problem B, an efficient solution to B can be used to solve A efficiently.

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a **poly time computable function** if some poly time TM M exists and halts with just $f(w)$ on its tape, given any input w.

i.e. M transforms w to $f(w)$ in poly time.

Definition

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

Polynomial Time Reduction

In computability theory we used reduction to show undecidability (without considering efficiency of reduction).
Considering efficiency of reduction: If problem A is **efficiently reducible** to problem B, an efficient solution to B can be used to solve A efficiently.

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a **poly time computable function** if some poly time TM M exists and halts with just $f(w)$ on its tape, given any input w.

i.e. M transforms w to $f(w)$ in poly time.

Definition

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

If a new problem A is poly time reducible to a known problem B with poly time solution, we obtain a poly time solution to A!

More precisely

Theorem

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

More precisely

Theorem

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

Proof

Let M be the poly time algo for B and f be the poly time reduction from A to B. A poly time algo N for A:

More precisely

Theorem

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

Proof

Let M be the poly time algo for B and f be the poly time reduction from A to B. A poly time algo N for A:
$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

More precisely

Theorem

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

Proof

Let M be the poly time algo for B and f be the poly time reduction from A to B. A poly time algo N for A:
$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Accounting (analysis):

- $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B

More precisely

Theorem

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

Proof

Let M be the poly time algo for B and f be the poly time reduction from A to B. A poly time algo N for A:
$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Accounting (analysis):

- $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B
- M accepts $f(w)$ whenever $w \in A$

More precisely

Theorem

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

Proof

Let M be the poly time algo for B and f be the poly time reduction from A to B. A poly time algo N for A:
$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Accounting (analysis):

- $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B
- M accepts $f(w)$ whenever $w \in A$
- N runs in poly time: each of its two stages runs in poly time.

Poly time reduction: an example

$3SAT$:

Poly time reduction: an example

$3SAT$: Each clause has exactly 3 variables, and entire formula is in CNF (*conjunctive normal form*) i.e. clauses connected with $\wedge$s and variables in clauses with $\vee$s.

Poly time reduction: an example

$3SAT$: Each clause has exactly 3 variables, and entire formula is in CNF (*conjunctive normal form*) i.e. clauses connected with $\wedge$s and variables in clauses with $\vee$s.

$3SAT = \{\langle \phi, k \rangle \mid \phi$ is a satisfiable 3CNF formula with k clauses$\}$.

Poly time reduction: an example

$3SAT$: Each clause has exactly 3 variables, and entire formula is in CNF (*conjunctive normal form*) i.e. clauses connected with $\wedge$s and variables in clauses with $\vee$s.

$3SAT = \{\langle \phi, k \rangle \mid \phi$ is a satisfiable 3CNF formula with k clauses$\}$.

Theorem: $3SAT \leq_p CLIQUE$.

Poly time reduction: an example

$3SAT$: Each clause has exactly 3 variables, and entire formula is in CNF (*conjunctive normal form*) i.e. clauses connected with $\wedge$s and variables in clauses with $\vee$s.

$3SAT = \{\langle \phi, k \rangle \mid \phi$ is a satisfiable 3CNF formula with k clauses$\}$.

Theorem: $3SAT \leq_p CLIQUE$.

Goal: show if $\langle \phi, k \rangle \in 3SAT \iff f(\langle \phi, k \rangle) = \langle G, k \rangle \in CLIQUE$.

Poly time reduction: an example

$3SAT$: Each clause has exactly 3 variables, and entire formula is in CNF (*conjunctive normal form*) i.e. clauses connected with $\wedge$s and variables in clauses with $\vee$s.

$3SAT = \{\langle \phi, k \rangle \mid \phi$ is a satisfiable 3CNF formula with k clauses$\}$.

Theorem: $3SAT \leq_p CLIQUE$.

Goal: show if $\langle \phi, k \rangle \in 3SAT \iff f(\langle \phi, k \rangle) = \langle G, k \rangle \in CLIQUE$.

E.g. $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2 \vee x_2)$, where $k = 3$.

Poly time reduction: an example

$3SAT$: Each clause has exactly 3 variables, and entire formula is in CNF (*conjunctive normal form*) i.e. clauses connected with $\wedge$s and variables in clauses with $\vee$s.

$3SAT = \{\langle \phi, k \rangle \mid \phi$ is a satisfiable 3CNF formula with k clauses$\}$.

Theorem: $3SAT \leq_p CLIQUE$.

Goal: show if $\langle \phi, k \rangle \in 3SAT \iff f(\langle \phi, k \rangle) = \langle G, k \rangle \in CLIQUE$.

E.g. $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2 \vee x_2)$, where $k = 3$. Convert ϕ to a graph for $CLIQUE$.

Poly time reduction: an example

Theorem: $3SAT \leq_p CLIQUE$.

Proof:

Let $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$.

Poly time reduction: an example

Theorem: $3SAT \leq_p CLIQUE$.

Proof:

Let $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as:

Poly time reduction: an example

Theorem: $3SAT \leq_p CLIQUE$.

Proof:

Let $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as:

- The nodes in G are organized into k groups of three nodes each called the triples, $t_1, \ldots, t_k$.
  - Each triple corresponds to one of the clauses in ϕ, and
  - each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.
- The edges of G connect all but two types of pairs of nodes in G.
  - No edge is present between nodes in the same triple and
  - no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$.

Poly time reduction: an example

Theorem: $3SAT \leq_p CLIQUE$.

Proof:

Let $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$.
The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as:

- The nodes in G are organized into k groups of three nodes each called the triples, $t_1, \ldots, t_k$.
  - Each triple corresponds to one of the clauses in ϕ, and
  - each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.
- The edges of G connect all but two types of pairs of nodes in G.
  - No edge is present between nodes in the same triple and
  - no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$.

We show that ϕ is satisfiable iff G has k-clique.

Suppose that ϕ has a satisfying assignment ("if" part)

Suppose that ϕ has a satisfying assignment ("if" part)

- In that satisfying assignment, at least one literal is true in every clause.

Suppose that ϕ has a satisfying assignment ("if" part)

- In that satisfying assignment, at least one literal is true in every clause.
- In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily.

Suppose that ϕ has a satisfying assignment ("if" part)

- In that satisfying assignment, at least one literal is true in every clause.
- In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily.
- The nodes just selected form a k-clique.

Suppose that ϕ has a satisfying assignment ("if" part)

- In that satisfying assignment, at least one literal is true in every clause.
- In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily.
- The nodes just selected form a k-clique.

Therefore G contains a k-clique.

Suppose that G has a k-clique ("only if" part)

Suppose that G has a k-clique ("only if" part)

- No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes.

Suppose that G has a k-clique ("only if" part)

- No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes.
- We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique.

Suppose that G has a k-clique ("only if" part)

- No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes.
- We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique.
- Each triple contains a clique node and hence each clause contains a literal that is assigned TRUE.

Suppose that G has a k-clique ("only if" part)

- No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes.
- We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique.
- Each triple contains a clique node and hence each clause contains a literal that is assigned TRUE.

Therefore ϕ is satisfiable.

Fin (wakas)

Thanks for the attention.

Questions?

Reading assignment(s)

[Sipser 2005] Chapter 7.3, 7.4 or

[Hopcroft et al 2001] Chapter 10.2

References:

[Sipser 2005] M. Sipser. *Introduction to the Theory of Computation*: 2ed. PWS Publishing Company, 2005.

[Hopcroft, Ullman 1979] J. Hopcroft, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*: Addison-Wesley, 1979.

[Hopcroft et al 2001] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*: Addison-Wesley, 2001.

[Hernandez 2014] CS133 lecture slides of N.H.S. Hernandez, 2014