Lecture 12

- NTIME and the classes NP.
Lecture 12

- NTIME and the classes NP.
- Examples of Problems in NP.
Lecture 12

- **NTIME** and the classes **NP**.
- Examples of Problems in **NP**.
- Verifiability.
Lecture 12

- NTIME and the classes NP.
- Examples of Problems in NP.
- Verifiability.
- Poly-Time Reductions
Lecture 12

- NTIME and the classes NP.
- Examples of Problems in NP.
- Verifiability.
- Poly-Time Reductions
- NP completeness
Non-Deterministic Time (reminder)

Let $N$ be a non-deterministic TM, and let $f : \mathcal{N} \rightarrow \mathcal{N}$.

We say that $N$ runs in time $f(n)$ if
- For every input $x$ of length $n$,
Non-Deterministic Time (reminder)

Let $N$ be a non-deterministic TM, and let

$$f : \mathcal{N} \rightarrow \mathcal{N}$$

We say that $N$ runs in time $f(n)$ if

- For every input $x$ of length $n$,
- the maximum number of steps that $N$ uses,
Non-Deterministic Time (reminder)

Let $N$ be a non-deterministic TM, and let

$$f : \mathcal{N} \rightarrow \mathcal{N}$$

We say that $N$ runs in time $f(n)$ if

- For every input $x$ of length $n$,
- the maximum number of steps that $N$ uses,
- on any branch of its computation tree on $x$, 

Non-Deterministic Time (reminder)

Let $N$ be a non-deterministic TM, and let

$$f : \mathcal{N} \rightarrow \mathcal{N}$$

We say that $N$ runs in time $f(n)$ if

- For every input $x$ of length $n$,
- the maximum number of steps that $N$ uses,
- on any branch of its computation tree on $x$,
- is at most $f(n)$. 
**NTime Classes Definition**

Let 

\[ f : \mathbb{N} \rightarrow \mathbb{N} \]

be a function.

**Definition:**

\[ \text{NTIME}(f(n)) = \{ L \mid L \text{ is a language, decided by an } O(f(n))-\text{time NTM} \} \]
The Class $NP$

**Definition:** $NP$ is the set of languages decidable in polynomial time on non-deterministic TMs.

$$NP = \bigcup_{c \geq 0} \text{NTIME}(n^c)$$

The class $NP$ is important because:
The Class \( NP \)

**Definition:** \( NP \) is the set of languages decidable in polynomial time on non-deterministic TMs.

\[
NP = \bigcup_{c \geq 0} \text{NTIME}(n^c)
\]

- The class \( NP \) is important because:
  - Invariant for all TMs with any number of tapes.
The Class $NP$

**Definition:** $NP$ is the set of languages decidable in polynomial time on non-deterministic TMs.

$$NP = \bigcup_{c \geq 0} \text{NTIME}(n^c)$$

- The class $NP$ is important because:
  - Invariant for all TMs with any number of tapes.
  - $NP$ is insensitive to choice of reasonable non-deterministic computational model.
The Class \( NP \)

**Definition:** \( NP \) is the set of languages decidable in polynomial time on non-deterministic TMs.

\[
NP = \bigcup_{c \geq 0} \text{NTIME}(n^c)
\]

- The class \( NP \) is important because:
  - Invariant for all TMs with any number of tapes.
  - \( NP \) is insensitive to choice of reasonable non-deterministic computational model.
  - Roughly corresponds to problems whose positive solutions cannot be efficiently generated (\( \Rightarrow \) intractable), but can be efficiently checked.
A Hamiltonian path in a directed $\mathcal{G}$ visits each node exactly once.
Hamiltonian Path

$$\text{HAMPATH} = \{\langle G, s, t \rangle | G \text{ has Hamiltonian path from } s \text{ to } t \}$$

**Question:** How hard is it to decide this language?
Hamiltonian Path

\[
\text{HAMPATH} = \{ \langle G, s, t \rangle | G \text{ has Hamiltonian path from } s \text{ to } t \}\]

Easy to obtain exponential time algorithm:
- generate each potential path
- check whether it is Hamiltonian
The Class $NP$

Here is an NTM that decides HAMPATH in poly time.

On input $\langle G, s, t \rangle$,

1. Guess and write down a list of numbers $p_1, \ldots, p_m$, where $m$ is number of nodes in $G$, and $1 \leq p_i \leq m$.

2. Check for repetitions in list. If any found, reject.

3. Check whether $p_1 = s$ and $p_m = t$. If either does not hold, reject.

4. For $i, 1 \leq i \leq m - 1$, check whether $(p_i, p_{i+1})$ is an edge in $G$. If any is not, reject. Otherwise accept.
**NP**

On input \( \langle G, s, t \rangle \),

1. Guess and write down a list of numbers \( p_1, \ldots, p_m \ldots \)
2. Check for repetitions . . .
3. Check whether \( p_1 = s \) and \( p_m = t \) . . .
4. Check whether \( (p_i, p_{i+1}) \) is an edge in \( G \) . . .

- Stage 1 polynomial time
- Stages 2 and 3 simple checks.
- Stage 4 simple poly-time too.
Hamiltonian Path

This problem has one very interesting feature: polynomial verifiability.

we don’t know a fast way to find a Hamiltonian path
Hamiltonian Path

This problem has one very interesting feature: **polynomial verifiability**.

- we don’t know a fast way to **find** a Hamiltonian path
- but we can **check** whether a **given path** is Hamiltonian in polynomial time.

In other words,

- **verifying** correctness of a path is much **easier**
- than **determining** whether one exists
Composite Numbers

A natural number is composite if it is the product of two integers greater than one.

\[
\text{COMPOSITES} = \{x \mid x = pq \text{ for integers } p, q > 1\}
\]

- we don’t know a polynomial-time algorithm for deciding this problem*
- But we can easily verify that a number is composite (how?)

*Actually, in summer 2002, two Indian undergrads and their advisor found how to do this. However, let us pretend we’re still in 1/1/2002...
Verifiability

Not all problems are polynomially verifiable.

There is no known way to verify \( \text{HAMPATH} \) in polynomial time.

In fact, we will see many examples where \( L \) is polynomially verifiable, but its complement, \( \overline{L} \), is not known to be polynomially verifiable.
Verifiability

A verifier for a language $A$ is an algorithm $V$ where

$$A = \{ w \mid V \text{ accepts} \langle w, c \rangle \text{ for some string } c \}$$

The verifier uses the additional information $c$ to verify $w \in A$. 
Verifiability

A verifier for a language $\mathcal{A}$ is an algorithm $\mathcal{V}$ where

$$\mathcal{A} = \{ w \mid \mathcal{V} \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$

- The verifier uses the additional information $c$ to verify $w \in \mathcal{A}$.
- We measure verifier run time by length of $w$. 
Verifiability

A verifier for a language $A$ is an algorithm $V$ where

$$A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$

- The verifier uses the additional information $c$ to verify $w \in A$.
- We measure verifier run time by length of $w$.
- The string $c$ is called a certificate (or proof) for $w$ if $V$ accepts $\langle w, c \rangle$. 
Verifiability

A verifier for a language $A$ is an algorithm $\mathcal{V}$ where

$$A = \{ w \mid \mathcal{V} \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$

- The verifier uses the additional information $c$ to verify $w \in A$.
- We measure verifier run time by length of $w$.
- The string $c$ is called a certificate (or proof) for $w$ if $\mathcal{V}$ accepts $\langle w, c \rangle$.
- A polynomial verifier runs in polynomial time in $|w|$ (so $|c| \leq |w|^{O(1)}$).
Verifiability

A verifier for a language $A$ is an algorithm $V$ where

$$A = \{ w | V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$

- The verifier uses the additional information $c$ to verify $w \in A$.
- We measure verifier run time by length of $w$.
- The string $c$ is called a certificate (or proof) for $w$ if $V$ accepts $\langle w, c \rangle$.
- A polynomial verifier runs in polynomial time in $|w|$ (so $|c| \leq |w|^{O(1)}$).
- A language $A$ is polynomially verifiable if it has a polynomial verifier.
Examples

For HAMPATH, a certificate for

\[ \langle G, s, t \rangle \in \text{HAMPATH} \]

is simply the Hamiltonian path from \( s \) to \( t \).

Can verify in time polynomial in \(|\langle G \rangle|\) whether given path is Hamiltonian.
Examples

For **COMPOSITES**, a certificate for

\[ x \in \text{COMPOSITES} \]

is simply one of its divisors.

Can **verify** in time polynomial in \( |x| \) if given divisor indeed divides \( x \).
The Class $\mathcal{NP}$, Again

$\mathcal{NP}$ is important because it includes many problems of practical interest.

- Hamiltonian path
The Class $\mathcal{NP}$, Again

$\mathcal{NP}$ is important because it includes many problems of practical interest.

- Hamiltonian path
- Travelling salesman (salesperson, that is)
The Class $\mathcal{NP}$, Again

$\mathcal{NP}$ is important because it includes many problems of practical interest.

- Hamiltonian path
- Travelling salesman (salesperson, that is)
- Scheduling (operations research)
The Class $\mathcal{NP}$, Again

$\mathcal{NP}$ is important because it includes many problems of practical interest.

- Hamiltonian path
- Travelling salesman (salesperson, that is)
- Scheduling (operations research)
- Placement and routing (VLSI design)
The Class $\mathcal{NP}$, Again

$\mathcal{NP}$ is important because it includes many problems of practical interest.

- Hamiltonian path
- Travelling salesman (salesperson, that is)
- Scheduling (operations research)
- Placement and routing (VLSI design)
- Composites (factoring/cryptography)
NP and Verifiability

**Theorem:** A language is in \( NP \) if and only if it has a polynomial time verifiers.

**Basic Idea:**
NP and Verifiability

**Theorem:** A language is in \( \mathcal{NP} \) if and only if it has a polynomial time verifiers.

**Basic Idea:**

- NTM simulates verifier by guessing the certificate.
NP and Verifiability

**Theorem:** A language is in \( \mathcal{NP} \) if and only if it has a polynomial time verifiers.

**Basic Idea:**

- NTM simulates verifier by guessing the certificate.
- Verifier simulates NTM by using accepting branch as certificate.
NP

Claim: If $A$ has a poly-time verifier, then is decided by some polynomial-time NTM.

Let $V$ be poly-time verifier for $A$.

- single-tape TM
- runs in time $n^k$

$N$: on input $w$ of length $n$

- Nondeterministically select string $c$ of length $n^k$.
- Run $V$ on $\langle w, c \rangle$
- If $V$ accepts, accept; otherwise reject.
Claim: If $A$ is decided by a polynomial-time NTM $N$, then $A$ has a poly-time verifier.

Construct polynomial-time verifier $V$ as follows.

$V$: on input $w$ of length $n$

- Simulate $N$ on input $w$, treating each symbol of $c$ as a description of each step’s non-deterministic choice.
- If this branch accepts, accept, otherwise reject.
Examples: Clique

A **clique** in a graph is a subgraph where every two nodes are connected by an edge.

A **$k$-clique** is a clique of size $k$. 
Examples: Clique

Define the language

\[ \text{CLIQUE} = \{ \langle G, k \rangle | G \text{ is an undirected graph with a } k\text{-clique} \} \]
Examples: Clique

Theorem:

CLIQUE ∈ \textit{NP}

The clique is the certificate.

Here is a verifier \( V \): on input \((G, k), c\)
Examples: Clique

Theorem:

$$\text{CLIQUE} \in \mathcal{NP}$$

The clique is the certificate.

Here is a verifier $\mathcal{V}$: on input $(\langle G, k \rangle, c)$

- if $c$ is not a $k$-clique, reject
Examples: Clique

Theorem:

\[ \text{CLIQUE} \in \mathcal{NP} \]

The clique is the certificate.

Here is a verifier \( V \): on input \( \langle G, k, c \rangle \)

- if \( c \) is not a \( k \)-clique, reject
- if \( G \) does not contain all vertices of \( c \), reject
Examples: Clique

Theorem: 

$$\text{CLIQUE} \in \mathcal{NP}$$

The clique is the certificate.

Here is a verifier $\mathcal{V}$: on input $(\langle G, k \rangle, c)$

- if $c$ is not a $k$-clique, $\text{reject}$
- if $G$ does not contain all vertices of $c$, $\text{reject}$
- $\text{accept}$
Examples: SUBSET-SUM

An instance of the problem

- A collection of numbers $x_1, \ldots, x_k$
- Target number $t$
- Question: does some subcollection add up to $t$?

\[
\text{SUBSET-SUM} = \{ \langle S, t \rangle \mid S = \{ x_1, \ldots, x_k \} \exists \{ y_1, \ldots, y_\ell \} \subseteq \{ x_1, \ldots, x_k \}, \sum_{y_j} = t \}\]
Examples: SUBSET-SUM

We have

\[(\{4, 11, 16, 21, 27\}, 25) \in \text{SUBSET-SUM}\]

because \(4 + 21 = 25\).

Collections are multisets: repetitions allowed.
Examples: SUBSET-SUM

Theorem:

\[ \text{SUBSET-SUM} \in NP \]

The subset is the certificate.

Here is a verifier:

\( \mathcal{V} \): on input \((\langle S, t \rangle, c)\)

- test whether \( c \) is a collection of numbers summing to \( t \).
- test whether \( c \) is a subset of \( S \)
- if either fail, reject, otherwise accept.
Complementary Problems

**CLIQUE** and **SUBSET-SUM** seem **not** to be members of NP.
It is harder to efficiently verify that something **does not** exist than to efficiently verify that something **does** exist..
Complementary Problems

**CLIQUE** and **SUBSET-SUM** seem *not* to be members of NP.

It is harder to efficiently verify that something *does not* exist than to efficiently verify that something *does* exist.

**Definition:** The class **coNP**: 
\[ L \in \text{coNP} \text{ if } \overline{L} \in \text{NP} \].
Complementary Problems

**CLIQUE** and **SUBSET-SUM** seem **not** to be members of NP.

It is harder to efficiently verify that something **not** exist than to efficiently verify that something **does** exist..

**Definition:** The class **coNP**:

$L \in \text{coNP}$ if $\overline{L} \in \text{NP}$.

So far, no one knows if **coNP** is distinct from **NP**.
The question $P = NP$? is one of the great unsolved mysteries in contemporary mathematics.

- most computer scientists believe the two classes are not equal
- most bogus proofs show them equal (why?)
Observations

If $\mathcal{P}$ differs from $\mathcal{NP}$, then the distinction between $\mathcal{P}$ and $\mathcal{NP} - \mathcal{P}$ is meaningful and important.

languages in $\mathcal{P}$ tractable
Observations

If $\mathcal{P}$ differs from $\mathcal{NP}$, then the distinction between $\mathcal{P}$ and $\mathcal{NP} - \mathcal{P}$ is meaningful and important.

- languages in $\mathcal{P}$ tractable
- languages in $\mathcal{NP} - \mathcal{P}$ intractable

Until we can prove that $\mathcal{P} \neq \mathcal{NP}$, there is no hope of proving that a specific language lies in $\mathcal{NP} - \mathcal{P}$. 
Observations

If $\mathcal{P}$ differs from $\mathcal{NP}$, then the distinction between $\mathcal{P}$ and $\mathcal{NP} - \mathcal{P}$ is meaningful and important.

- languages in $\mathcal{P}$ tractable
- languages in $\mathcal{NP} - \mathcal{P}$ intractable

Until we can prove that $\mathcal{P} \neq \mathcal{NP}$, there is no hope of proving that a specific language lies in $\mathcal{NP} - \mathcal{P}$. Nevertheless, we can prove statements of the form “If $\mathcal{P} \neq \mathcal{NP}$ then $A \in \mathcal{NP} - \mathcal{P}$.”
The class of **NP-complete** languages are

- “hardest” languages in $NP$
- “least likely” to be in $P$
- If any NP-complete $A \in P$, then $NP = P$. 
Cook–Levin (1971-1973)

**Theorem:** There is a language $S \in NP$ such that $S \in P$ if and only if $P = NP$. 

This theorem establishes the class of NP-complete languages. Such a language, like Frodo Baggins, “carries on its back” all burden of $NP$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Cook–Levin (1971-1973)

**Theorem:** There is a language $S \in \mathcal{NP}$ such that $S \in \mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.

This theorem establishes the class of NP-complete languages.

Such language, like Frodo Baggins, “carries on its back” all burden of $\mathcal{NP}$. 
Poly-Time Computable Functions

**Definition:** A function

\[ f : \Sigma^* \rightarrow \Sigma^* \]

is **polynomial-time computable** if there is a poly-time deterministic TM that

- starts with input \( w \), and
- halts with \( f(w) \) on tape.
Poly-Time Reducibility

Definition: We say that a language $A$ is polynomial time mapping reducible to $B$, written

$$A \leq_P B,$$

if there is a poly-time computable function

$$f : \Sigma^* \longrightarrow \Sigma^*$$

such that, for every $w$,

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

The function $f$ is called a polynomial-time reduction from $A$ to $B$. 
Computable Functions

Converts questions about membership in $A$ to membership in $B$, and does it efficiently.
Computable Functions

**Theorem:** If $A \leq_P B$ and $B \in P$ then $A \in P$.

**Proof:** Let

$f$ the reduction from $A$ to $B$, computed by TM $M_f$. 
Computable Functions

**Theorem:** If $A \leq_P B$ and $B \in P$ then $A \in P$.

**Proof:** Let

- $f$ the reduction from $A$ to $B$, computed by TM $M_f$.
- On input $x$ of length $n$, $M_f$ takes at most $c_1 n^{a_1}$ steps.
Computable Functions

**Theorem:** If $A \leq_P B$ and $B \in P$ then $A \in P$.

**Proof:** Let

- $f$ the reduction from $A$ to $B$, computed by TM $M_f$.
- On input $x$ of length $n$, $M_f$ takes at most $c_1 n^{a_1}$ steps.
- $M$ be the poly-time decider for $B$. 

Computable Functions

**Theorem:** If $A \leq_P B$ and $B \in P$ then $A \in P$.

**Proof:** Let

- $f$ the reduction from $A$ to $B$, computed by TM $M_f$.
- On input $x$ of length $n$, $M_f$ takes at most $c_1 n^{a_1}$ steps.
- $M$ be the poly-time decider for $B$.
- On input $y$ of length $m$, $M$ takes at most $c_2 m^{a_2}$ steps.
Computable Functions

Define $N$: on input $x$

1. compute $f(x)$
2. run $M$ on input $f(x)$ and output whatever $M$ outputs.

Analysis:

On input $x$ of length $n$, computing $y = f(x)$ takes at most $c_1 n^{a_1}$ steps.
Computable Functions

Define $\mathcal{N}$: on input $x$

1. compute $f(x)$
2. run $\mathcal{M}$ on input $f(x)$ and output whatever $\mathcal{M}$ outputs.

Analysis:

- On input $x$ of length $n$, computing $y = f(x)$ takes at most $c_1 n^{a_1}$ steps.
- On input $y$ of length $m = c_1 n^{a_1}$, $\mathcal{M}$ takes at most $c_2 m^{a_2} = c_2 (c_1 n^{a_1})^{a_2} = (c_2 c_1^{a_2}) n^{a_1 a_2}$ steps.
Computable Functions

Define $\mathcal{N}$: on input $x$

1. compute $f(x)$
2. run $\mathcal{M}$ on input $f(x)$ and output whatever $\mathcal{M}$ outputs.

Analysis:

- On input $x$ of length $n$, computing $y = f(x)$ takes at most $c_1 n^{a_1}$ steps.
- On input $y$ of length $m = c_1 n^{a_1}$, $\mathcal{M}$ takes at most $c_2 m^{a_2} = c_2 (c_1 n^{a_1})^{a_2} = (c_2 c_1^{a_2}) n^{a_1 \cdot a_2}$ steps.
- Summing both stages, we got a polynomial in $n$. 

Correctness is clear, so $A \in \mathcal{P}$. 

♣

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Computable Functions

Define $\mathcal{N}$: on input $x$

1. compute $f(x)$
2. run $\mathcal{M}$ on input $f(x)$ and output whatever $\mathcal{M}$ outputs.

Analysis:

- On input $x$ of length $n$, computing $y = f(x)$ takes at most $c_1 n^{a_1}$ steps.
- On input $y$ of length $m = c_1 n^{a_1}$, $\mathcal{M}$ takes at most $c_2 m^{a_2} = c_2 (c_1 n^{a_1})^{a_2} = (c_2 c_1^{a_2}) n^{a_1 \cdot a_2}$ steps.
- Summing both stages, we got a polynomial in $n$.
- Correctness is clear, so $\mathcal{A} \in P$. ♣
Satisfiability

A boolean variable assumes values
Satisfiability

A boolean variable assumes values true (written 1), and false (written 0).
Satisfiability

- A boolean variable assumes values
  - true (written 1), and false (written 0).
- Boolean operations:
Satisfiability

- A boolean variable assumes values true (written 1), and false (written 0).
- Boolean operations:
  - and: $\land$
Satisfiability

A boolean variable assumes values true (written 1), and false (written 0).

Boolean operations:
- and: ∧
- or: ∨
Satisfiability

A boolean variable assumes values true (written 1), and false (written 0).

Boolean operations:

- and: $\land$
- or: $\lor$
- not: $\neg$

Examples:

- $0 \land 1 = 0$
- $0 \lor 1 = 1$
- $0 = 1$
Satisfiability

- A boolean variable assumes values true (written 1), and false (written 0).

- Boolean operations:
  - and: $\land$
  - or: $\lor$
  - not: $\neg$

- Examples:

  $0 \land 1 = 0$
  $0 \lor 1 = 1$
  $\overline{0} = 1$
Satisfiability

A boolean formula is an expression involving boolean variables and operations.

\[ \phi = (\overline{x} \land y) \lor (x \land \overline{z}) \]
Satisfiability

A boolean formula is an expression involving boolean variables and operations.

\[ \phi = (\overline{x} \land y) \lor (x \land \overline{z}) \]

Definition: A formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.
Satisfiability

\[ \phi = (\overline{x} \land y) \lor (x \land \overline{z}) \]

is satisfiable by

\[ x = 0 \]
\[ y = 1 \]
\[ z = 0 \]

This assignment satisfies \( \phi \).
Satisfiability

Define

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

It is useful to consider special version:
Satisfiability

It is useful to consider special version:

- A **literal** is a variable or negated variable: $x$ or $\overline{x}$. 
Satisfiability

It is useful to consider special version:

- A **literal** is a variable or negated variable: \( x \) or \( \overline{x} \).
- A **term** is several literals joined by \( \lor \)s:
  \[
  (x_1 \lor \overline{x_2} \lor \overline{x_3})
  \]
Satisfiability

It is useful to consider special version:

- A **literal** is a variable or negated variable: \( x \) or \( \overline{x} \).
- A **term** is several literals joined by \( \lor \)'s:
  \[
  (x_1 \lor \overline{x_2} \lor \overline{x_3})
  \]
- A Boolean formula is in **conjunctive normal form** (CNF) if it consists of terms, connected with \( \land \)'s.
Satisfiability

It is useful to consider special version:

- A **literal** is a variable or negated variable: $x$ or $\overline{x}$.
- A **term** is several literals joined by $\lor$ s:
  \[(x_1 \lor \overline{x_2} \lor \overline{x_3})\]
- A Boolean formula is in **conjunctive normal form (CNF)** if it consists of **terms**, connected with $\land$ s.
- For example
  \[(x_1 \lor \overline{x_2} \lor \overline{x_3} \lor x_4) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6})\]
Satisfiability

**Definition:** A Boolean formula is in 3CNF form if it is a CNF formula, and all terms have three literals.

\[(x_1 \lor \overline{x}_2 \lor \overline{x}_3) \land (x_3 \lor \overline{x}_5 \lor x_6) \land (x_3 \lor \overline{x}_6 \lor x_4)\]
Satisfiability

**Definition:** A Boolean formula is in 3CNF form if it is a CNF formula, and all terms have three literals.

\[(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \land (x_3 \vee \bar{x}_5 \vee x_6) \land (x_3 \vee \bar{x}_6 \vee x_4)\]

Define

\[3\text{SAT} = \{ \langle \phi \rangle \mid \phi \text{ is satisfiable 3CNF formula} \}\]
Satisfiability

**Definition:** A Boolean formula is in 3CNF form if it is a CNF formula, and all terms have three literals.

\[(x_1 \lor \overline{x_2} \lor \overline{x_3}) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6} \lor x_4)\]

Define

\[3SAT = \{\langle \phi \rangle \mid \phi \text{ is satisfiable 3CNF formula}\}\]

Clearly, if \(\phi\) is a satisfiable 3CNF formula, then for any satisfying assignment of \(\phi\), every clause must contain at least one literal assigned 1.
Reductions

**Claim:** There is a poly time reduction from 3SAT to CLIQUE. In other words,

\[ 3\text{SAT} \leq^P \text{CLIQUE} . \]
Reductions

Claim: There is a poly time reduction from 3SAT to CLIQUE. In other words,

\[ 3\text{SAT} \leq_P \text{CLIQUE}. \]

We’ll construct a poly time reduction \( f \) that maps 3CNF formulae \( \phi \) to graphs and numbers, \( \langle G, k \rangle \). The function \( f \) will have the property that \( \phi \) is satisfiable if and only if \( G \) has a clique of size \( k \).
Examples: Clique

Reminder: A **clique** in a graph is a subgraph where every two nodes are connected by an edge.

A **$k$-clique** is a clique of size $k$. For example, the graph above has a **5-clique**.
3SAT $\leq_P$ CLIQUE

Let $\phi$ be a 3CNF formula with $k$ clauses.

$$(x_1 \lor \overline{x_2} \lor \overline{x_3}) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6} \lor x_4)$$

We define a graph $G$ as follows:
3SAT $\leq_P$ CLIQUE

We define a graph $G$ as follows:

- nodes in $G$ are organized into triples $t_1, \ldots, t_k$. 

3SAT $\leq_P$ CLIQUE

We define a graph $G$ as follows:

- nodes in $G$ are organized into triples $t_1, \ldots, t_k$.
- each triple corresponds to a term of $\phi$. 

3SAT $\leq_P$ CLIQUE

We define a graph $G$ as follows:

- nodes in $G$ are organized into triples $t_1, \ldots, t_k$.
- each triple corresponds to a term of $\phi$
- each node in a triple corresponds to a literal.
$\text{3SAT} \leq_P \text{CLIQUE}$

$$(x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_3} \lor x_5 \lor x_6) \land (x_3 \lor x_4 \lor \overline{x_6})$$
3SAT vs. CLIQUE

\[(x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_3} \lor x_5 \lor x_6) \land (x_3 \lor x_4 \lor \overline{x_6})\]

Add edges between all vertex pairs, except
- within same triple
- between contradictory literals
3SAT $\leq_P$ CLIQUE

**Claim:** If $\phi$ is satisfiable, $G$ has a $k$-clique.

Suppose $\phi$ is satisfiable.

- at least one literal is true in every term
Claim: If $\phi$ is satisfiable, $G$ has a $k$-clique.

Suppose $\phi$ is satisfiable.

- at least one literal is true in every term
- in every tuple, select one true literal
3SAT \leq_P CLIQUE

Claim: If $\phi$ is satisfiable, $G$ has a $k$-clique.

Suppose $\phi$ is satisfiable.

- at least one literal is true in every term
- in every tuple, select one true literal
- they can be joined by edges
3SAT $\leq_P$ CLIQUE

**Claim:** If $\phi$ is satisfiable, $G$ has a $k$-clique.

Suppose $\phi$ is satisfiable.
- at least one literal is true in every term
- in every tuple, select one true literal
- they can be joined by edges
- yielding a $k$-clique
3SAT \leq_P CLIQUE

**Claim:** If $\phi$ is satisfiable, $G$ has a $k$-clique.

$$(x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_3} \lor x_5 \lor x_6) \land (x_3 \lor x_4 \lor \overline{x_6})$$

![Diagram showing a graph with vertices labeled $x_1, x_2, \ldots, x_6$ connected by edges to form a 5-clique.](image-url)
3SAT $\leq_P$ CLIQUE

Claim: If $G$ has a $k$-clique, $\phi$ is satisfiable.

- No two of the cliques nodes are in the same triple.
3SAT \leq_P CLIQUE

Claim: If \( G \) has a \( k \)-clique, \( \phi \) is satisfiable.

- No two of the cliques nodes are in the same triple.
- Have \( k \) vertexes and \( k \) terms, so
3SAT $\leq_P$ CLIQUE

**Claim:** If $G$ has a $k$-clique, $\phi$ is satisfiable.

- No two of the cliques nodes are in the same triple.
- Have $k$ vertexes and $k$ terms, so
- each triple has exactly one clique node.
3SAT $\leq_P$ CLIQUE

Claim: If $G$ has a $k$-clique, $\phi$ is satisfiable.

- No two of the cliques nodes are in the same triple.
- Have $k$ vertexes and $k$ terms, so
- each triple has exactly one clique node.
- Assign 1 to each node in clique
3SAT $\leq_P$ CLIQUE

**Claim:** If $G$ has a $k$-clique, $\phi$ is satisfiable.

- No two of the cliques nodes are in the same triple.
- Have $k$ vertexes and $k$ terms, so
each triple has exactly one clique node.
- Assign 1 to each node in clique
- no contradictions.
3SAT $\leq_P$ CLIQUE

We’ve constructed a poly time computable function $f$. 
3SAT $\leq_P$ CLIQUE

- We’ve constructed a poly time computable function $f$.
- We saw that the function $f$ has the property that $\phi \in 3\text{SAT}$ if and only if $f(\phi) \in \text{CLIQUE}$.
3SAT $\leq_P$ CLIQUE

- We’ve constructed a poly time computable function $f$.
- We saw that the function $f$ has the property that $\phi \in 3\text{SAT}$ if and only if $f(\phi) \in \text{CLIQUE}$.
- Therefore $f$ is a reduction from 3SAT to CLIQUE, so $3\text{SAT} \leq_P \text{CLIQUE}$.

♣
Independent Set

An independent set in a graph is a set of vertexes, no two of which are linked by an edge.

The independent set problem asks whether there exists an independent set of size $k$. 
Independent Set

Define

\[ \text{INDEPENDENT-SET} = \{ \langle G, k \rangle | G \text{ contains an independent set of size } k \} \]
Independent Set

Define

\[ \text{INDEPENDENT-SET} = \{ \langle G, k \rangle \mid G \text{ contains an independent set of size } k \} \]

**Claim:** INDEPENDENT-SET is polynomial time reducible to CLIQUE,

\[ \text{INDEPENDENT-SET} \leq_P \text{CLIQUE} \]

and vice-versa,

\[ \text{CLIQUE} \leq_P \text{INDEPENDENT-SET} \]
Independent Set

**Definition:** The complement of a graph of \( G = (V, E) \) is a graph \( G^c = (V, E^c) \), where

\[
E^c = \{(v_1, v_2) \mid v_1, v_2 \in V \text{ and } (v_1, v_2) \notin E\}.
\]
Independent Set

**Definition:** The complement a graph of $G = (V, E)$ is a graph $G^c = (V, E^c)$, where

$$E^c = \{(v_1, v_2) | v_1, v_2 \in V \text{ and } (v_1, v_2) \notin E\}.$$

**Claim:** If $V$ is an independent set in $G$, then $V$ is a clique in $G^c$.

’nuff said.
Independent Set
A Hamiltonian path in a directed $G$ visits each note once.
Hamiltonian Path

\[ \text{HAMPATH} = \{ \langle G, s, t \rangle | G \text{ has Hamiltonian path from } s \text{ to } t \} \]
Hamiltonian Circuit

visits each note once.
Hamiltonian Circuit

- visits each node once.
- ends up **where it started**
Hamiltonian Circuit

\[ \text{HAMCIRCUIT} = \{ \langle G \rangle \mid G \text{ has Hamiltonian circuit} \} \]
Hamiltonian Circuit

\[
\text{HAMCIRCUIT} = \{ \langle G \rangle \mid G \text{ has Hamiltonian circuit} \}
\]

**Theorem:** HAMPATH is polynomial-time reducible to HAMCIRCUIT,

\[
\text{HAMPATH} \leq_P \text{HAMCIRCUIT}.
\]
Reduction

**Theorem:** HAMPATH is polynomial-time reducible to HAMCIRCUIT.
Reduction

**Theorem:** HAMCIRCUIT is polynomial-time reducible to HAMPATH.

**Proof:** Left as an easy (recommended) exercise.
Definition

A language $\mathcal{B}$ is NP-complete if it satisfies
Definition

A language $\mathcal{B}$ is NP-complete if it satisfies

$\mathcal{B} \in NP$, and
Definition

A language $\mathcal{B}$ is **NP-complete** if it satisfies

- $\mathcal{B} \in \text{NP}$, and
- Every $\mathcal{A}$ in NP is polynomial time reducible to $\mathcal{B}$
Theorem

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

To show $P = NP$, suffices find a polynomial-time algorithm for some NP-complete problem.
Theorem

**Theorem:** If $\mathcal{B}$ is NP-complete and $\mathcal{B} \leq_P \mathcal{C}$, for $\mathcal{C} \in \text{NP}$, then $\mathcal{C}$ is NP-complete.

- We know that $\mathcal{C} \in \text{NP}$,
Theorem

**Theorem:** If $\mathcal{B}$ is NP-complete and $\mathcal{B} \leq_P \mathcal{C}$, for $\mathcal{C} \in \text{NP}$, then $\mathcal{C}$ is NP-complete.

- We know that $\mathcal{C} \in \text{NP}$,
- must show that every $\mathcal{A}$ in NP is poly-time reducible to $\mathcal{C}$. 
Theorem

**Theorem:** If $\mathcal{B}$ is NP-complete and $\mathcal{B} \leq_P \mathcal{C}$, for $\mathcal{C} \in NP$, then $\mathcal{C}$ is NP-complete.

- We know that $\mathcal{C} \in NP$,
- must show that every $\mathcal{A}$ in NP is poly-time reducible to $\mathcal{C}$.
- Because $\mathcal{B}$ is NP-complete,
Theorem

**Theorem:** If $\mathcal{B}$ is NP-complete and $\mathcal{B} \leq_P \mathcal{C}$, for $\mathcal{C} \in \mathcal{NP}$, then $\mathcal{C}$ is NP-complete.

- We know that $\mathcal{C} \in \mathcal{NP}$,
- must show that every $\mathcal{A}$ in NP is poly-time reducible to $\mathcal{C}$.
- Because $\mathcal{B}$ is NP-complete,
- every language in NP is poly-time reducible to $\mathcal{B}$.
Theorem

*Theorem:* If $B$ is NP-complete and $B \leq_P C$, for $C \in NP$, then $C$ is NP-complete.

- We know that $C \in NP$,
- must show that every $A$ in NP is poly-time reducible to $C$.
- Because $B$ is NP-complete,
- every language in NP is poly-time reducible to $B$.
- $B$ is poly-time reducible to $C$
Theorem

**Theorem:** If $\mathcal{B}$ is NP-complete and $\mathcal{B} \leq_P \mathcal{C}$, for $\mathcal{C} \in NP$, then $\mathcal{C}$ is NP-complete.

- We know that $\mathcal{C} \in NP$,
- must show that every $\mathcal{A}$ in NP is poly-time reducible to $\mathcal{C}$.
- Because $\mathcal{B}$ is NP-complete,
- every language in NP is poly-time reducible to $\mathcal{B}$.
- $\mathcal{B}$ is poly-time reducible to $\mathcal{C}$
- Can compose poly-time reductions (why?), so
Theorem

**Theorem:** If $\mathcal{B}$ is NP-complete and $\mathcal{B} \leq_P \mathcal{C}$, for $\mathcal{C} \in NP$, then $\mathcal{C}$ is NP-complete.

- We know that $\mathcal{C} \in NP$,
- must show that every $\mathcal{A}$ in NP is poly-time reducible to $\mathcal{C}$.
- Because $\mathcal{B}$ is NP-complete,
- every language in NP is poly-time reducible to $\mathcal{B}$.
- $\mathcal{B}$ is poly-time reducible to $\mathcal{C}$
- Can compose poly-time reductions (why?), so $\mathcal{A}$ is poly-time reducible to $\mathcal{C}$.
Strategy

Once we have one “structured” NP-complete problem, we can generate more by poly-time reduction.
Strategy

- Once we have one “structured” NP-complete problem, we can generate more by poly-time reduction.
- Getting the first one requires some work.
Strategy

- Once we have one “structured” NP-complete problem, we can generate more by poly-time reduction.

- Getting the first one requires some work.

- This is what Steve Cook (then in Berkeley, now in Toronto) and Leonid Levin (then in Moscow, now in Boston) did in the early seventies.
Traveling Salesman

Parameters:
- set of cities $C$
- set of inter-city distances $D$
- goal $k$

(not drawn to scale)
Traveling Salesman

Define

\[
\text{TRAVELING-SALESMAN} = \{ \langle C, D, k \rangle | (C, D, k) \text{ there is a tour of total distance } \leq k \}.
\]