Part VII. Selected Topics
Chapter 33.9 Exhaustive Search (covered!)
Part VII. Selected Topics

Chapter 33.9 Exhaustive Search (covered!)

Chapter 34 NP-Completeness
Part VII. Selected Topics

Chapter 33.9 Exhaustive Search (covered!)
Chapter 34 NP-Completeness
Chapter 33.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

- systematic examining all solutions
- without repeating solutions that have been examined
- stop when a satisfactory solution is found
Chapter 33.9. Exhaustive Search

To enumerate all possible solutions to the problem instance
Chapter 33.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

How?
Chapter 33.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

How?

- systematic examining all solutions
Chapter 33.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

How?

• systematic examining all solutions
• without repeating solutions that have been examined
Chapter 33.9. Exhaustive Search

To enumerate all possible solutions to the problem instance

How?

• systematic examining all solutions
• without repeating solutions that have been examined
• stop when a satisfactory solution is found
Chapter 33.9. Exhaustive Search

First, we need to be able to count total number of “things” to be enumerated.
Chapter 33.9. Exhaustive Search

First, we need to be able to count total number of “things” to be enumerated.

• without missing one (correctness)
Chapter 33.9. Exhaustive Search

First, we need to be able to count total number of “things” to be enumerated.

- without missing one (correctness)
- without over-counting (efficiency)
First, we need to be able to count total number of “things” to be enumerated.

- without missing one (correctness)
- without over-counting (efficiency)
- A sophisticated counting often has recursive solution.
Chapter 33.9. Exhaustive Search

Examples of counting:

1. Total number of permutations of \(1, 2, \ldots, n\) is \(P(n) = n \times P(n-1)\) with base case \(P(1) = 1\).

2. Total number of ways to choose \(k\) from \(n\) items is \(\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!} = \frac{n!}{(n-k)!k!}\) or, alternatively, \(\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}\) with base cases: (??)
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = \]

\[ n \times P(n-1) \times \cdots \times \frac{n}{2} \times 1 = n! \]
Examples of counting:

1. total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \(1, 2, \ldots, n\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) =
\]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1)
\]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2)
\]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is 

\[ P(n) = n \times P(n-1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n-1) = n \times (n-1) \times P(n-2) = n \times (n-1) \times \ldots \times 2 \times 1 = n! \]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots \times 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k}
\]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} = n(n - 1) \ldots (n - k + 1) \]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[
P(n) = n \times P(n - 1)
\]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!}
\]
Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!}
\]

or, alternatively,

\[
\binom{n}{k}
\]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!} = \frac{n!}{(n-k)! \times k!}
\]

or, alternatively,

\[
\binom{n}{k} = \binom{n-1}{k}
\]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[ P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n! \]

(2) total number of ways to choose \(k\) from \(n\) items is

\[ \binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!} \]

or, alternatively,

\[ \binom{n}{k} = \binom{n - 1}{k} + \binom{n - 1}{k - 1} \]
Chapter 33.9. Exhaustive Search

Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!}
\]

or, alternatively,

\[
\binom{n}{k} = \binom{n - 1}{k} + \binom{n - 1}{k - 1}
\]
Examples of counting:

(1) total number of permutations of \((1, 2, \ldots, n)\) is

\[ P(n) = n \times P(n - 1) \]

with base case \(P(1) = 1\).

\[
P(n) = n \times P(n - 1) = n \times (n - 1) \times P(n - 2) = n \times (n - 1) \times \ldots 2 \times 1 = n!
\]

(2) total number of ways to choose \(k\) from \(n\) items is

\[
\binom{n}{k} = \frac{n(n - 1) \ldots (n - k + 1)}{k!} = \frac{n!}{(n - k)! \times k!}
\]

or, alternatively,

\[
\binom{n}{k} = \binom{n - 1}{k} + \binom{n - 1}{k - 1}
\]

with base cases: (?)
Chapter 33.9. Exhaustive Search

Example: Boolean Formula Satisfiability problem (SAT)
Example: Boolean Formula Satisfiability problem (SAT)

**Input:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**Output:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**OUTPUT:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

\( f(x_1, x_2, \ldots, x_n) \) is **satisfiable** if there is an **assignment** to boolean variables

\[ x_i \in \{T, F\}, \ i = 1, 2, \ldots, n, \]
Chapter 33.9. Exhaustive Search

Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**OUTPUT:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

$f(x_1, x_2, \ldots, x_n)$ is satisfiable if there is an assignment to boolean variables $x_i \in \{T, F\}$, $i = 1, 2, \ldots, n$, such that $f$ is evaluated to $T$. 

For example, $f(x_1, x_2, x_3) = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3)$ is satisfiable.
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**OUTPUT:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

\( f(x_1, x_2, \ldots, x_n) \) is **satisfiable** if there is an **assignment** to boolean variables \( x_i \in \{T, F\}, \ i = 1, 2, \ldots, n \), such that \( f \) is evaluated to \( T \).

e.g,
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**OUTPUT:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

\( f(x_1, x_2, \ldots, x_n) \) is satisfiable if there is an assignment to boolean variables \( x_i \in \{T, F\}, i = 1, 2, \ldots, n \), such that \( f \) is evaluated to \( T \).

e.g., \( f(x_1, x_2, x_3) = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3) \) is satisfiable
Example: Boolean Formula Satisfiability problem (SAT)

**INPUT:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**OUTPUT:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

\( f(x_1, x_2, \ldots, x_n) \) is **satisfiable** if there is an **assignment** to boolean variables \( x_i \in \{T, F\}, i = 1, 2, \ldots, n \), such that \( f \) is evaluated to T.

**e.g.,**

\( f(x_1, x_2, x_3) = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3) \) is **satisfiable**

\( g(x_1, x_2) = (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) is **not**!
Chapter 33.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.
Use exhaustive search to solve the SAT problem.

**INPUT:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**OUTPUT:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.
Use exhaustive search to solve the SAT problem.

\textbf{Input:} boolean formula $f(x_1, x_2, \ldots, x_n)$,

\textbf{Output:} "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

How?
Chapter 33.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**Input:** boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**Output:** "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

How? What will you exhaustively search on?
Chapter 33.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**Input**: boolean formula \( f(x_1, x_2, \ldots, x_n) \),

**Output**: "yes" if and only if \( f(x_1, x_2, \ldots, x_n) \) is satisfiable.

How? What will you exhaustively search on?

- **Enumerate all combinations of T and F for** \( x_1, \ldots, x_n \).
Chapter 33.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**Input:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**Output:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

How? What will you exhaustively search on?

- **Enumerate all combinations of T and F for $x_1, \ldots, x_n$.**
- Can you solve it with a recursive algorithm?
Chapter 33.9. Exhaustive Search

Use exhaustive search to solve the SAT problem.

**Input:** boolean formula $f(x_1, x_2, \ldots, x_n)$,

**Output:** "yes" if and only if $f(x_1, x_2, \ldots, x_n)$ is satisfiable.

How? What will you exhaustively search on?

- **Enumerate all combinations of T and F for** $x_1, \ldots, x_n$.
- Can you solve it with a recursive algorithm?
- Can you solve it with an iterative algorithm?
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  - boolean formula \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  - \( n=0, \) formula without variables

- what is the recursive case?
  - \( f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F) \)

\[ f(x_1, \ldots, x_{n-1}, T) = \Rightarrow g(x_1, \ldots, x_{n-1}) \]
\[ f(x_1, \ldots, x_{n-1}, F) = \Rightarrow h(x_1, \ldots, x_{n-1}) \]
Chapter 33.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?

  \[ f(x_1, \ldots, x_n) \]

  - terminating (base) case:
    \[ n = 0, \text{ formula without variables} \]

  - recursive case:
    \[ f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F) \]

    \[ f(x_1, \ldots, x_{n-1}, T) = \Rightarrow g(x_1, \ldots, x_{n-1}) \]

    \[ f(x_1, \ldots, x_{n-1}, F) = \Rightarrow h(x_1, \ldots, x_{n-1}) \]
Chapter 33.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula \( f(x_1, \ldots, x_n) \)
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula \( f(x_1, \ldots, x_n) \)
  
- what is the terminating (base) case?
Chapter 33.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  boolean formula $f(x_1, \ldots, x_n)$

- what is the terminating (base) case?
  $n=0$, formula without variables
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  
  \( n=0 \), formula without variables

- what is the recursive case?
  
  \[
  f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F) \]
  
  \( g(x_1, \ldots, x_{n-1}) \Rightarrow f(x_1, \ldots, x_{n-1}, F) \Rightarrow h(x_1, \ldots, x_{n-1}) \)
Chapter 33.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula $f(x_1, \ldots, x_n)$

- what is the terminating (base) case?
  
  $n=0$, formula without variables

- what is the recursive case?

  $$f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)$$
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  
  \( n=0 \), formula without variables

- what is the recursive case?
  
  \[
  f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F) \\
  f(x_1, \ldots, x_{n-1}, T)
  \]
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula \( f(x_1, \ldots, x_n) \)

- what is the terminating (base) case?
  
  \( n=0 \), formula without variables

- what is the recursive case?

  \[
  f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)
  \]

  \[
  f(x_1, \ldots, x_{n-1}, T) \implies g(x_1, \ldots, x_{n-1})
  \]
Chapter 33.9. Exhaustive Search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  
  boolean formula $f(x_1, \ldots, x_n)$

- what is the terminating (base) case?
  
  $n=0$, formula without variables

- what is the recursive case?

  
  $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)$

  $f(x_1, \ldots, x_{n-1}, T) \implies g(x_1, \ldots, x_{n-1})$

  $f(x_1, \ldots, x_{n-1}, F) \implies h(x_1, \ldots, x_{n-1})$
Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  \[
  \text{boolean formula } f(x_1, \ldots, x_n)
  \]

- what is the terminating (base) case?
  \[
  n=0, \text{ formula without variables}
  \]

- what is the recursive case?
  \[
  f(x_1, \ldots, x_{n-1}, x_n) = f(x_1, \ldots, x_{n-1}, T) \lor f(x_1, \ldots, x_{n-1}, F)
  \]
  \[
  f(x_1, \ldots, x_{n-1}, T) \implies g(x_1, \ldots, x_{n-1})
  \]
  \[
  f(x_1, \ldots, x_{n-1}, F) \implies h(x_1, \ldots, x_{n-1})
  \]
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver($f(x_1, \ldots, x_{n-1}, x_n)$)
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver\(f(x_1, \ldots, x_{n-1}, x_n)\)

1. if \(n = 0\), return \(f\);
Chapter 33.9. Exhaustive Search

Algorithm $\text{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))$

1. if $n = 0$, return $(f)$;
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$

Does this algorithm exhaustively search all assignments to the variables?

- Draw a search tree based on the algorithm.
- What does the tree look like?
- What does each path mean?
- How many paths?
- Time? $T(n) = 2T(n-1) + cn$, $T(0) = c = \Theta(2^n)$
Chapter 33.9. Exhaustive Search

Algorithm \texttt{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{if} \( n = 0 \), \textbf{return} \( f \);
2. \textbf{else} \( g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \text{T}) \)
3. \( h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \text{F}) \)
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver\( (f(x_1, \ldots, x_{n-1}, x_n)) \)

1. if \( n = 0 \), return \( f \);
2. else \( g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T) \)
3. \( h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F) \)
4. return (SAT Solver\( (g(x_1, \ldots, x_{n-1})) \) \( \lor \) SAT Solver\( (h(x_1, \ldots, x_{n-1})) \))

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
- what does each path mean?
- how many paths?
- time? \( T(n) = 2T(n-1) + cn \), \( T(0) = c \), \( ⇒ T(n) = Θ(2^n) \)
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver($f(x_1, \ldots, x_{n-1}, x_n)$)

1. if $n = 0$, return ($f$);
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
4. return (SAT Solver($g(x_1, \ldots, x_{n-1})$) $\lor$ SAT Solver($h(x_1, \ldots, x_{n-1})$))

Does this algorithm exhaustively search all assignments to the variables?
Chapter 33.9. Exhaustive Search

Algorithm \textsc{Sat Solver}(f(x_1,\ldots,x_{n-1},x_n))

1. \textbf{if} \ n = 0, \textbf{return} \ (f);
2. \textbf{else} \ g(x_1,\ldots,x_{n-1}) = f(x_1,\ldots,x_{n-1},T)
3. \quad h(x_1,\ldots,x_{n-1}) = f(x_1,\ldots,x_{n-1},F)
4. \textbf{return} \ \textsc{Sat Solver}(g(x_1,\ldots,x_{n-1})) \lor \textsc{Sat Solver}(h(x_1,\ldots,x_{n-1}))

Does this algorithm exhaustively search all assignments to the variables?

- draw a \textit{search tree} based on the algorithm.
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver($f(x_1, \ldots, x_{n-1}, x_n)$)

1. if $n = 0$, return ($f$);
2. else $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
4. return (SAT Solver($g(x_1, \ldots, x_{n-1})$) $\lor$ SAT Solver($h(x_1, \ldots, x_{n-1})$))

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
Chapter 33.9. Exhaustive Search

Algorithm \textsc{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{if} \ n = 0, \textbf{return} (f);
2. \textbf{else} \ \ g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)
3. \quad h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)
4. \textbf{return} \ (\textsc{SAT Solver}(g(x_1, \ldots, x_{n-1})) \lor \textsc{SAT Solver}(h(x_1, \ldots, x_{n-1})))

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
- what does each path mean?
Chapter 33.9. Exhaustive Search

Algorithm \textsc{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))

1. if \( n = 0 \), return \((f)\);
2. else \( g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T) \)
3. \( h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F) \)
4. return \((\text{SAT Solver}(g(x_1, \ldots, x_{n-1})) \lor \text{SAT Solver}(h(x_1, \ldots, x_{n-1})))\)

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
Chapter 33.9. Exhaustive Search

Algorithm \texttt{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{if} \ n = 0, \textbf{return} (f);
2. \textbf{else} \ g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \text{T})
3. \hspace{1em} h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, \text{F})
4. \textbf{return} (\texttt{SAT Solver}(g(x_1, \ldots, x_{n-1})) \lor \texttt{SAT Solver}(h(x_1, \ldots, x_{n-1})))

Does this algorithm exhaustively search all assignments to the variables?

- draw a \textit{search tree} based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
- \textit{time}?
Algorithm SAT Solver($f(x_1, \ldots, x_{n-1}, x_n)$)

1. \textbf{if} $n = 0$, \textbf{return} ($f$);
2. \textbf{else} $g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)$
3. \hspace{1em} $h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)$
4. \textbf{return} (SAT Solver($g(x_1, \ldots, x_{n-1})$) $\lor$
\hspace{1em} SAT Solver($h(x_1, \ldots, x_{n-1})$))

Does this algorithm exhaustively search all assignments to the variables?

- draw a \textit{search tree} based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
- time? $T(n) = 2T(n - 1) + cn$, $T(0) = c$, 

Chapter 33.9. Exhaustive Search
Chapter 33.9. Exhaustive Search

Algorithm \textsc{SAT Solver}(f(x_1, \ldots, x_{n-1}, x_n))

1. \hspace{5pt} \textbf{if} \ n = 0, \ \textbf{return} \ (f);
2. \hspace{5pt} \textbf{else} \ g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, T)
3. \hspace{5pt} h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, F)
4. \hspace{5pt} \textbf{return} \ (\textsc{SAT Solver}(g(x_1, \ldots, x_{n-1})) \lor \textsc{SAT Solver}(h(x_1, \ldots, x_{n-1})))

Does this algorithm exhaustively search all assignments to the variables?

- draw a search tree based on the algorithm.
- what does the tree look like?
- what does each path mean? how many paths?
- time? \( T(n) = 2T(n-1) + cn, \ T(0) = c, \implies T(n) = \Theta(2^n) \)
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  - assignments
- What is the initial value?
  - $x_1 = F, x_2 = F, \ldots, x_n = F$
  - or simply $(F, F, \ldots, F)$
- What to increment
  - $(\ldots, F, T, \ldots, T) \rightarrow (\ldots, T, F, \ldots, F)$
  - always flip the last bit.
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

\[ \text{assignments} \]
\[ x_1 = F, x_2 = F, \ldots, x_n = F, \]
or simply \((F, F, \ldots, F)\)

- What to increment \((\ldots, F, T, \ldots, T) \rightarrow (\ldots, T, F, \ldots, F)\)

always flip the last bit.
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

assignments
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

• How? what to iterate on?
  assignments

• what is the initial value?
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  assignments

- what is the initial value?
  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \]
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  - assignments
- what is the initial value?
  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  assignments

- what is the initial value?
  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]

- what to increment

always flip the last bit.
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?

  assignments

- what is the initial value?

  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]

- what to increment

  \((\ldots, F, T, \ldots, T)\)
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  assignments
- what is the initial value?
  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]
- what to increment
  \[ (\ldots, F, T, \ldots, T) \rightarrow (\ldots, T, F, \ldots, F) \]
Chapter 33.9. Exhaustive Search

Solve SAT problem with iterative algorithms

- How? what to iterate on?
  assignments

- what is the initial value?
  \[ x_1 = F, x_2 = F, \ldots, x_n = F, \text{ or simply } (F, F, \ldots, F) \]

- what to increment
  \[
  (\ldots, F, T, \ldots, T) \longrightarrow (\ldots, T, F, \ldots, F)
  \]
  always flip the last bit.
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver-Enum

1. for \( \langle x_1, \ldots, x_{n-1}, x_n \rangle \) = \( \langle F, \ldots, F \rangle \) to \( \langle T, \ldots, T \rangle \)
2. \( V = \text{Evaluate}(f, x_1, \ldots, x_n) \)
3. if \( V = T \), return \( (T) \)
4. return \( (F) \)

• for loop can be implemented by encoding vectors \( \langle F, \ldots, F \rangle, \ldots, \langle T, \ldots, T \rangle \) with binary numbers then further with integers
• a decoding process is needed to converting integers back to vectors
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver-Enum($f(x_1, \ldots, x_{n-1}, x_n)$)

Chapter 33.9. Exhaustive Search

Algorithm SAT Solver-Enum\( (f(x_1, \ldots, x_{n-1}, x_n)) \)

1. \textbf{for} \( \langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle \) \textbf{to} \( \langle T, \ldots, T \rangle \)
Algorithm SAT Solver-Enum\( (f(x_1, \ldots, x_{n-1}, x_n)) \)

1. for \( \langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle \) to \( \langle T, \ldots, T \rangle \)
2. \( V = Evaluate(f, x_1, \ldots, x_n) \)
Chapter 33.9. Exhaustive Search

Algorithm $\text{SAT Solver-Enum}(f(x_1, \ldots, x_{n-1}, x_n))$

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. \hspace{1em} $V = \text{Evaluate}(f, x_1, \ldots, x_n)$
3. \hspace{1em} if $V = T$, return $(T)$
Algorithm SAT Solver-Enum$(f(x_1, \ldots, x_{n-1}, x_n))$

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. $V = \text{Evaluate}(f, x_1, \ldots, x_n)$
3. if $V = T$, return (T)
4. return (F)

*• for loop can be implemented by encoding vectors $\langle F, \ldots, F \rangle$, ... $\langle T, \ldots, T \rangle$ with binary numbers then further with integers
• a decoding process is needed to converting integers back to vectors*
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver-Enum($f(x_1, \ldots, x_{n-1}, x_n)$)

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. \hspace{1em} $V = Evaluate(f, x_1, \ldots, x_n)$
3. \hspace{1em} if $V = T$, return (T)
4. \hspace{1em} return (F)

- for loop can be implemented by encoding vectors $\langle F, \ldots, F \rangle$, $\ldots$, $\langle T, \ldots, T \rangle$ with
Chapter 33.9. Exhaustive Search

Algorithm \textsc{SAT Solver-Enum}(f(x_1, \ldots, x_{n-1}, x_n))

1. \textbf{for} \langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle \textbf{ to } \langle T, \ldots, T \rangle
2. \hspace{1em} V = \text{Evaluate}(f, x_1, \ldots, x_n)
3. \hspace{1em} \textbf{if} V = T, \textbf{return} (T)
4. \hspace{1em} \textbf{return} (F)

- \textbf{for} loop can be implemented by encoding vectors \langle F, \ldots, F \rangle, 
  \ldots, \langle T, \ldots, T \rangle \text{ with binary numbers then}
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver-Enum($f(x_1, \ldots, x_{n-1}, x_n)$)

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. $V = \text{Evaluate}(f, x_1, \ldots, x_n)$
3. if $V = T$, return (T)
4. return (F)

• for loop can be implemented by encoding vectors $\langle F, \ldots, F \rangle$, $\ldots$, $\langle T, \ldots, T \rangle$ with binary numbers then further with integers
Chapter 33.9. Exhaustive Search

Algorithm SAT Solver-Enum($f(x_1, \ldots, x_{n-1}, x_n)$)

1. for $\langle x_1, \ldots, x_n \rangle = \langle F, \ldots, F \rangle$ to $\langle T, \ldots, T \rangle$
2. $V = Evaluate(f, x_1, \ldots, x_n)$
3. if $V = T$, return (T)
4. return (F)

• for loop can be implemented by encoding vectors $\langle F, \ldots, F \rangle$, $\ldots$, $\langle T, \ldots, T \rangle$ with binary numbers then further with integers
• a decoding process is needed to converting integers back to vectors
Iterative exhaustive search seems to be more convenient.

Another example: Travel Salesman Problem (TSP).

Related problem: Hamiltonian Cycle.

Input: a graph \( G = (V,E) \).

Output: yes if and only if \( G \) contains a Hamiltonian cycle (Hamiltonian path is a cycle going through every vertex exactly once).

How to enumerate all cycles and validate?

- enumerate all permutations of \((1, 2, \ldots, n)\).
- how to encode these permutations as integers?
Iterative exhaustive search seems to be more convenient
Chapter 33.9. Exhaustive Search

Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input**: a graph $G = (V, E)$

**Output**: yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input:** a graph $G = (V, E)$

**Output:** yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)

How to enumerate all cycles and validate?
Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input:** a graph $G = (V, E)$

**Output:** yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)

How to enumerate all cycles and validate?

- enumerate all permutations of $(12 \ldots n)$
Chapter 33.9. Exhaustive Search

Iterative exhaustive search seems to be more convenient

Another example: Travel Salesman Problem (TSP)

Related problem: Hamiltonian Cycle

**Input:** a graph $G = (V, E)$

**Output:** yes if and only if $G$ contains a Hamiltonian cycle

(Hamiltonian path is a cycle going through every vertex exactly once.)

How to enumerate all cycles and validate?

- enumerate all permutations of $(12 \ldots n)$
- how to encode these permutations as integers?
Chapter 33.9. Exhaustive Search

Exhaustive search could be non-trivial
Chapter 33.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**INPUT**: a graph $G = (V, E)$

**OUTPUT**: a subset $I \subseteq V$ such that

1. $\forall u, v \in I, (u, v) \notin E$, and
2. $|I|$ is the maximum.

- **trivial exhaustive search**: check every subset of $V$ and verify
- **non-trivial**: use a search tree, achieving a better time upper bound.

Taking advantage of the independent set
Chapter 33.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**Input:** a graph $G = (V, E)$

**Output:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I, (u, v) \notin E$,
Chapter 33.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**INPUT**: a graph \( G = (V, E) \)

**OUTPUT**: a subset \( I \subseteq V \) such that

1. \( \forall u, v \in I, (u, v) \notin E \), and
2. \( |I| \) is the maximum.
Exhaustive search could be non-trivial

Maximum Independent Set

**Input**: a graph \( G = (V, E) \)

**Output**: a subset \( I \subseteq V \) such that

1. \( \forall u, v \in I, (u, v) \notin E \), and
2. \( |I| \) is the maximum.

- trivial exhaustive search: check every subset of \( V \) and verify
Exhaustive search could be non-trivial

Maximum Independent Set

**Input:** a graph $G = (V, E)$

**Output:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I$, $(u, v) \notin E$, and
2. $|I|$ is the maximum.

- trivial exhaustive search: check every subset of $V$ and verify
  use $n$-binary bits to encode a subset; totally $2^n$ subsets
Chapter 33.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**Input**: a graph $G = (V, E)$

**Output**: a subset $I \subseteq V$ such that

1. $\forall u, v \in I, (u, v) \not\in E$, and
2. $|I|$ is the maximum.

- trivial exhaustive search: check every subset of $V$ and verify
  - use $n$-binary bits to encode a subset; totally $2^n$ subsets
- non-trivial: use a search tree, achieving a better time upper bound.
Chapter 33.9. Exhaustive Search

Exhaustive search could be non-trivial

Maximum Independent Set

**INPUT:** a graph $G = (V, E)$

**OUTPUT:** a subset $I \subseteq V$ such that

1. $\forall u, v \in I$, $(u, v) \not\in E$, and
2. $|I|$ is the maximum.

- trivial exhaustive search: check every subset of $V$ and verify
  - use $n$-binary bits to encode a subset; totally $2^n$ subsets

- non-trivial: use a search tree, achieving a better time upper bound.
  - taking advantage of the independent set
The algorithm follows a logical search tree

Chapter 33.9. Exhaustive Search
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  2. $G_2$ is the result of $G$ after $v$ is removed.
- the algorithm terminates when the considered graph is empty.
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  (1) to include $v$ in the independent set;
  (2) to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  (1) $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  (2) $G_2$ is the result of $G$ after $v$ is removed.
- the algorithm terminates when the considered graph is empty.
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  (1) to include $v$ in the independent set;
  (2) to exclude $v$ from the independent set;
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  (1) to include $v$ in the independent set;
  (2) to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  (1) $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;
- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;
- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  2. $G_2$ is the result of $G$ after $v$ is removed.
Chapter 33.9. Exhaustive Search

The algorithm follows a logical search tree

- given a graph $G$, it picks an arbitrary vertex $v$ from $G$;

- exhaustively, there are two cases to consider:
  1. to include $v$ in the independent set;
  2. to exclude $v$ from the independent set;

- resulting in two subgraphs $G_1$ and $G_2$ to be recursively considered,
  1. $G_1$ is the result of $G$ after $v$ and all its neighbors are removed;
  2. $G_2$ is the result of $G$ after $v$ is removed.

- the algorithm terminates when the considered graph is empty.
Algorithm **MaxIndSet** \((G)\)

1. if \(G = \emptyset\) return \((\emptyset)\)
2. else pick an arbitrary vertex \(v\) in \(G\)
3. let \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \text{MaxIndSet}(G_1)\)
5. let \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \text{MaxIndSet}(G_2)\)
7. if \(|I_1| \geq |I_2|\) return \((I_1)\)
8. else return \((I_2)\)

- the algorithm is a search tree
- the time complexity:

\[ T(n) = \sum_{m=0}^{n-1} T(n-m) + cn^2 + T(n-1) \]

where \(m\) is the number of neighbors of \(v\)'s
Chapter 33.9. Exhaustive Search

Algorithm \textbf{MaxIndSet} (\(G\))

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
Chapter 33.9. Exhaustive Search

Algorithm \texttt{MaxIndSet} \((G)\)

1. \textbf{if} \quad G = \emptyset \quad \textbf{return} \quad (\emptyset)
2. \textbf{else} \quad \text{pick an arbitrary vertex} \ v \ \text{in} \ G
Chapter 33.9. Exhaustive Search

Algorithm `MaxIndSet (G)`

1. **if** \( G = \emptyset \) **return** \( (\emptyset) \)
2. **else** pick an arbitrary vertex \( v \) in \( G \)
3. let \( G_1 \) be \( G \) with \( v \) and all its neighbors removed

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

where \( m \) is the number of neighbors of \( v \)'s
Chapter 33.9. Exhaustive Search

Algorithm \textbf{MaxIndSet} \((G)\)

1. \begin{itemize} 
   \item \textbf{if} \( G = \emptyset \) \textbf{return} \((\emptyset)\)
   \end{itemize}

2. \begin{itemize} 
   \item \textbf{else} pick an arbitrary vertex \( v \) in \( G \)
   \end{itemize}

3. \begin{itemize} 
   \item let \( G_1 \) be \( G \) with \( v \) and all its neighbors removed
   \end{itemize}

4. \begin{itemize} 
   \item \( I_1 = \{v\} \cup \text{MaxIndSet} \ (G_1) \)
   \end{itemize}

• the algorithm is a search tree
• the time complexity:
  \[ T(n) = T(n-1-m) + T(n-1) + cn^2 \]
  where \( m \) is the number of neighbors of \( v \)'s
Chapter 33.9. Exhaustive Search

Algorithm $\text{MaxIndSet} \ (G)$

1. if $G = \emptyset$ return $(\emptyset)$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} \ (G_1)$
5. let $G_2$ be $G$ with $v$ removed

$T(n) = T(n-1-m) + T(n-1) + cn^2$ where $m$ is the number of neighbors of $v$'s
Algorithm $\text{MaxIndSet} \ (G)$

1. if $G = \emptyset$ return $(\emptyset)$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} \ (G_1)$
5. let $G_2$ be $G$ with $v$ removed
6. $I_2 = \text{MaxIndSet} \ (G_2)$
Chapter 33.9. Exhaustive Search

Algorithm $\text{MaxIndSet} (G)$

1. if $G = \emptyset$ return $(\emptyset)$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} (G_1)$
5. let $G_2$ be $G$ with $v$ removed
6. $I_2 = \text{MaxIndSet} (G_2)$
7. if $|I_1| \geq |I_2|$ return $(I_1)$

$T(n) = T(n-1-m) + T(n-1) + cn^2$ where $m$ is the number of neighbors of $v$'s
Algorithm \textbf{MaxIndSet} \((G)\)

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \(v\) in \(G\)
3. \textbf{let} \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \text{MaxIndSet} \((G_1)\)\)
5. \textbf{let} \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \text{MaxIndSet} \((G_2)\)\)
7. \textbf{if} \(|I_1| \geq |I_2|\) \textbf{return} \((I_1)\)
8. \textbf{else} \textbf{return} \((I_2)\)
Chapter 33.9. Exhaustive Search

Algorithm MaxIndSet \((G)\)

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \(v\) in \(G\)
3. let \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \text{MaxIndSet} \((G_1)\)\)
5. let \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \text{MaxIndSet} \((G_2)\)\)
7. \textbf{if} \(|I_1| \geq |I_2|\) \textbf{return} \((I_1)\)
8. \textbf{else} \textbf{return} \((I_2)\)

- the algorithm is a search tree
Chapter 33.9. Exhaustive Search

Algorithm \textsc{MaxIndSet} \((G)\)

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \(v\) in \(G\)
3. let \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \textsc{MaxIndSet}(G_1)\)
5. let \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \textsc{MaxIndSet}(G_2)\)
7. \textbf{if} \(|I_1| \geq |I_2|\) \textbf{return} \((I_1)\)
8. \textbf{else} \textbf{return} \((I_2)\)

- the algorithm is a search tree
- the time complexity:

\(T(|G|) = cn^2 + T(|G_1|) + T(|G_2|)\)

where \(m\) is the number of neighbors of \(v\) ’s
Algorithm MaxIndSet \((G)\)

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \(v\) in \(G\)
3. let \(G_1\) be \(G\) with \(v\) and all its neighbors removed
4. \(I_1 = \{v\} \cup \text{MaxIndSet}(G_1)\)
5. let \(G_2\) be \(G\) with \(v\) removed
6. \(I_2 = \text{MaxIndSet}(G_2)\)
7. \textbf{if} \(|I_1| \geq |I_2|\) \textbf{return} \((I_1)\)
8. \textbf{else} \textbf{return} \((I_2)\)

- the algorithm is a search tree
- the time complexity: \(T(|G|) = cn^2 + T(|G_1|) + T(|G_2|)\)
Chapter 33.9. Exhaustive Search

Algorithm $\text{MaxIndSet} \ (G)$

1. if $G = \emptyset$ return $(\emptyset)$
2. else pick an arbitrary vertex $v$ in $G$
3. let $G_1$ be $G$ with $v$ and all its neighbors removed
4. $I_1 = \{v\} \cup \text{MaxIndSet} \ (G_1)$
5. let $G_2$ be $G$ with $v$ removed
6. $I_2 = \text{MaxIndSet} \ (G_2)$
7. if $|I_1| \geq |I_2|$ return $(I_1)$
8. else return $(I_2)$

- the algorithm is a search tree
- the time complexity: $T(|G|) = cn^2 + T(|G_1|) + T(|G_2|)$

\[
T(n) = T(n - 1 - m) + T(n - 1) + cn^2
\]

where $m$ is the number of neighbors of $v$’s
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0 \),
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2 , \)
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \) \( T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \)
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \]
- Or even better, to guarantee \( m \geq 2 \) if we can,
  \[ T(n) \leq T(n - 3) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \]
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)

- Can we guarantee \( m \geq 1 \) so we have
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \]
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \]
- Or even better, to guarantee \( m \geq 2 \)?
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0 \), \( T(n) \leq T(n - 1) + T(n - 1) + cn^2 \), \( \implies T(n) = O(2^n) \)

- Can we guarantee \( m \geq 1 \) so we have
  \( T(n) \leq T(n - 2) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.6181^n) \)

- Or even better, to guarantee \( m \geq 2 \)? if we can,
  \( T(n) \leq T(n - 3) + T(n - 1) + cn^2 \),
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0 \), \( T(n) \leq T(n - 1) + T(n - 1) + cn^2 \), \( \implies T(n) = O(2^n) \)

- Can we guarantee \( m \geq 1 \) so we have
  \( T(n) \leq T(n - 2) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.6181^n) \)

- Or even better, to guarantee \( m \geq 2 \) if we can,
  \( T(n) \leq T(n - 3) + T(n - 1) + cn^2 \), \( \implies T(n) = O(1.5^n) \)
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)
- Can we guarantee \( m \geq 1 \) so we have
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \]
- Or even better, to guarantee \( m \geq 2 \)? if we can,
  \[ T(n) \leq T(n - 3) + T(n - 1) + cn^2, \implies T(n) = O(1.5^n) \]

use the substitution method to prove \( T(n) = O(1.5^n) \).
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)

- Can we guarantee \( m \geq 1 \) so we have
  \( T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \)

- Or even better, to guarantee \( m \geq 2 \) if we can,
  \( T(n) \leq T(n - 3) + T(n - 1) + cn^2, \implies T(n) = O(1.5^n) \)

use the substitution method to prove \( T(n) = O(1.5^n) \).

- Can we guarantee \( m \geq 3 \)?
Chapter 33.9. Exhaustive Search

\[ T(n) = T(n - 1 - m) + T(n - 1) + cn^2 \]

- \( m \geq 0, \ T(n) \leq T(n - 1) + T(n - 1) + cn^2, \implies T(n) = O(2^n) \)

- Can we guarantee \( m \geq 1 \) so we have
  \[ T(n) \leq T(n - 2) + T(n - 1) + cn^2, \implies T(n) = O(1.6181^n) \]

- Or even better, to guarantee \( m \geq 2 \)? if we can,
  \[ T(n) \leq T(n - 3) + T(n - 1) + cn^2, \implies T(n) = O(1.5^n) \]

  use the substitution method to prove \( T(n) = O(1.5^n) \).

- Can we guarantee \( m \geq 3 \)? possible but a little more complicated.
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. 

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2$$
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)
Let $T(n) \leq T(n-3) + T(n-1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n-3) + T(n-1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}})$$
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1)$$
Let $T(n) \leq T(n-3) + T(n-1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof (use the substitution method)**

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n-3) + T(n-1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$
Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

**Claim:** $T(n) = O(1.5^n)$

**Proof** (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35$$
Let $T(n) \leq T(n-3) + T(n-1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n-3) + T(n-1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375$$
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3$$
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n-3) + T(n-1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n-3) + T(n-1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n$$
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n$$

Now we decide $n_0$:

$$\frac{n^2}{1.5^{n-3}} \leq 0.1$$
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n$$

Now we decide $n_0$:

$$\frac{n^2}{1.5^{n-3}} \leq 0.1 \implies n^2 \leq 0.1 \times 1.5^{n-3}$$
Chapter 33.9. Exhaustive Search

Let $T(n) \leq T(n - 3) + T(n - 1) + cn^2$, with $T(1) = O(1)$

Claim: $T(n) = O(1.5^n)$

Proof (use the substitution method)

Assume that $T(k) \leq 1.5^k$ for all $k < n$. Then

$$T(n) \leq T(n - 3) + T(n - 1) + n^2 \leq 1.5^{n-3} + 1.5^{n-1} + n^2$$

when $n > n_0$ ($n_0$ to be determined)

$$\leq 1.5^{n-3}(1 + 1.5^2 + \frac{n^2}{1.5^{n-3}}) \leq 1.5^{n-3}(1 + 1.5^2 + 0.1) = 1.5^{n-3}(1 + 2.25 + 0.1)$$

$$= 1.5^{n-3} \times 3.35 \leq 1.5^{n-3} \times 3.375 = 1.5^{n-3} \times 1.5^3 = 1.5^n$$

Now we decide $n_0$:

$$\frac{n^2}{1.5^{n-3}} \leq 0.1 \implies n^2 \leq 0.1 \times 1.5^{n-3} \text{ holds when roughly } n \geq n_0 = 29$$
Chapter 33.9. Exhaustive Search

- Algorithms for SAT and MAXINDSET run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$
Chapter 33.9. Exhaustive Search

- Algorithms for SAT and MaxIndSet run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$
- search tree (solution search space) is large, inherently large
• Algorithms for SAT and \textsc{MaxIndSet} run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$

• search tree (solution search space) is large, \textit{inherently} large

• search tree does not have obvious overlapping subproblems,
Chapter 33.9. Exhaustive Search

- Algorithms for SAT and MaxIndSet run in exponential time $O(2^n)$ or $O(\gamma^n)$ for $1 < \gamma < 2$

- Search tree (solution search space) is large, inherently large

- Search tree does not have obvious overlapping subproblems, which otherwise would incur dynamic programming approaches.
Chapter 34. NP-Completeness

1. Intractable problems
   - investigating decision problems suffice

2. NP model
   - how to show a problem is in class NP

3. NP-completeness framework
   - polynomial-time reduction and NP-completeness proof.
Chapter 34. NP-Completeness

Chapter 34 NP-Completeness
Chapter 34. NP-Completeness

1. Intractable problems
Chapter 34. NP-Completeness

Chapter 34 NP-Completeness

1. Intractable problems
   • investigating decision problems suffice
Chapter 34. NP-Completeness

Chapter 34 NP-Completeness

1. Intractable problems
   - investigating decision problems suffice

2. NP model
Chapter 34. NP-Completeness

Chapter 34 NP-Completeness

1. Intractable problems
   • investigating decision problems suffice

2. NP model
   • how to show a problem is in class NP
Chapter 34 NP-Completeness

1. Intractable problems
   - investigating decision problems suffice

2. NP model
   - how to show a problem is in class NP

3. NP-completeness framework
Chapter 34. NP-Completeness

Chapter 34 NP-Completeness

1. Intractable problems
   • investigating decision problems suffice

2. NP model
   • how to show a problem is in class NP

3. NP-completeness framework
   • polynomial-time reduction and NP-completeness proof.
1. Intractable problems
Chapter 34. NP-Completeness

1. Intractable problems
   • we have seen many problems solvable in polynomial time
Chapter 34. NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
1. Intractable problems

- we have seen many problems solvable in polynomial time
e.g., sorting, SCC, MST

- there are problems that do not seem to have polynomial time algorithms
Chapter 34. NP-Completeness

1. Intractable problems
   • we have seen many problems solvable in polynomial time
     e.g., sorting, SCC, MST
   • there are problems that do not seem to have polynomial time algorithms
     i.e., not solvable in time $O(n \log n)$,
1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST

- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or
1. **Intractable problems**

- we have seen many problems solvable in polynomial time
e.g., sorting, SCC, MST

- there are problems that do **not** seem to have polynomial time algorithms
  i.e., **not solvable** in time $O(n \log n)$, $O(n^3)$, or $O(n^{100})$. 
Chapter 34. NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
e.g., sorting, SCC, MST

- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or $O(n^{100})$.

- why would a time $O(n^{100})$-time algorithm be attractive?
1. **Intractable problems**

- we have seen many problems solvable in polynomial time
e.g., sorting, SCC, MST

- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time \(O(n \log n)\), \(O(n^3)\), or \(O(n^{100})\).

- why would a time \(O(n^{100})\)-time algorithm be attractive?
  only theoretical?
Chapter 34. NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
e.g., sorting, SCC, MST

- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or $O(n^{100})$.

- why would a time $O(n^{100})$-time algorithm be attractive?
  only theoretical? practical significance as well
Chapter 34. NP-Completeness

Define: a Hamiltonian cycle in a graph is a circular path going through every vertex exactly once. Different from Eulerian cycle that goes through every edge exactly once.
Define: a Hamiltonian cycle in a graph is a circular path going through every vertex exactly once.
Chapter 34. NP-Completeness

**Define:** a Hamiltonian cycle in a graph is a circular path going through every vertex exactly once.
Chapter 34. NP-Completeness

**Define:** a Hamiltonian cycle in a graph is a circular path going through every vertex exactly once.

**Different** from Eulerian cycle that goes through every edge exactly once.
Define: a Hamiltonian cycle in a graph is a circular path going through every vertex exactly once.

Different from Eulerian cycle that goes through every edge exactly once.
Chapter 34. NP-Completeness

Travel Salesman Problem (TSP)

Input: an edge-weighted graph \( G = (V,E) \);

Output: a Hamiltonian cycle of the minimum weight sum.

- intuitively, a circular path is a permutation of \((v_1,v_2,...,v_n)\) or simply a permutation of \((1,2,...,n)\), where \(|V| = n\).

so the problem has time upper bound \( O(n!|E|) \), exponential time.

\[
n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \geq n \times (n-1) \cdots \times 2 \times 1 = (n^2)
\]

- all known algorithms (solving TSP) are of exponential-time.
Travel Salesman Problem (TSP)

Input: an edge-weighted graph $G = (V, E)$;
Chapter 34. NP-Completeness

Travel Salesman Problem (TSP)

**Input:** an edge-weighted graph \( G = (V, E) \);

**Output:** a Hamiltonian cycle of the minimum weight sum.
Chapter 34. NP-Completeness

**Travel Salesman Problem (TSP)**

**Input:** an edge-weighted graph $G = (V, E)$;

**Output:** a Hamiltonian cycle of the minimum weight sum.

- intuitively, a circular path is a permutation of $(v_1, v_2, \ldots, v_n)$ or simply a permutation of $(1, 2, \ldots, n)$, where $|V| = n$. 
Travel Salesman Problem (TSP)

- **Input**: an edge-weighted graph $G = (V, E)$;
- **Output**: a Hamiltonian cycle of the minimum weight sum.

- intuitively, a circular path is a permutation of $(v_1, v_2, \ldots, v_n)$ or simply a permutation of $(1, 2, \ldots, n)$, where $|V| = n$. so the problem has time upper bound $O(n!|E|)$, exponential time.
Chapter 34. NP-Completeness

**Travel Salesman Problem (TSP)**

**Input:** an edge-weighted graph \( G = (V, E) \);

**Output:** a Hamiltonian cycle of the minimum weight sum.

- intuitively, a circular path is a permutation of \((v_1, v_2, \ldots, v_n)\) or simply a permutation of \((1, 2, \ldots, n)\), where \(|V| = n\). so the problem has time upper bound \(O(n! |E|)\), exponential time.

\[
n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \geq n \times (n-1) \cdots \times \frac{n}{2} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}
\]

- all known algorithms (solving TSP) are of exponential-time.
Chapter 34. NP-Completeness

Instead of considering Travel Salesman Problem (TSP) Input: an edge-weighted graph $G = (V, E)$;
Output: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem: H-Cycle Weight Decision (HCW) Input: an edge-weighted graph $G = (V, E)$, a weight value $K$;
Output: "YES" if and only if there is a Hamiltonian cycle of weight $\leq K$ in $G$.

• HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
• However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.
Instead of considering

The Travel Salesman Problem (TSP) involves an edge-weighted graph $G = (V,E)$; the output is a Hamiltonian cycle of the minimum weight sum. We might consider a related problem:

**H-Cycle Weight Decision (HCW)**

- **Input**: an edge-weighted graph $G = (V,E)$, a weight value $K$.
- **Output**: "YES" if and only if there is a Hamiltonian cycle of weight $\leq K$ in $G$.

- **HCW** appears to "easier" than **TSP** as an H-cycle is not produced in the answer.

- However, **HCW** may not be "easier".

**Theorem 1**: **HCW** is solvable in P-time if and only if **TSP** is solvable in P-time.
Chapter 34. NP-Completeness

Instead of considering

**Travel Salesman Problem (TSP)**

**Input:** an edge-weighted graph $G = (V, E)$;

**Output:** a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

**H-Cycle Weight Decision (HCW)**

**Input:** an edge-weighted graph $G = (V, E)$, a weight value $K$;

**Output:** “YES” if and only if there is a Hamiltonian cycle of weight $\leq K$ in $G$.

- **HCW** appears to “easier” than **TSP** as an H-cycle is not produced in the answer.

- However, **HCW** may not be “easier”

**Theorem 1:**

**HCW** is solvable in P-time if and only if **TSP** is solvable in P-time.
Instead of considering

**Travel Salesman Problem (TSP)**

**Input:** an edge-weighted graph $G = (V, E)$;

**Output:** a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:
Chapter 34. NP-Completeness

Instead of considering

**Travel Salesman Problem (TSP)**

**Input:** an edge-weighted graph \( G = (V, E) \);

**Output:** a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

**H-Cycle Weight Decision (HCW)**

**Input:** an edge-weighted graph \( G = (V, E) \), a weight value \( K \);
Instead of considering

**Travel Salesman Problem (TSP)**
- **Input**: an edge-weighted graph \( G = (V, E) \);
- **Output**: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

**H-Cycle Weight Decision (HCW)**
- **Input**: an edge-weighted graph \( G = (V, E) \), a weight value \( K \);
- **Output**: “YES” if and only there is a Hamiltonian cycle of weight \( \leq K \) in \( G \).
Instead of considering

**Travel Salesman Problem (TSP)**

- **Input**: an edge-weighted graph $G = (V, E)$;
- **Output**: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

**H-Cycle Weight Decision (HCW)**

- **Input**: an edge-weighted graph $G = (V, E)$, a weight value $K$;
- **Output**: “YES” if and only there is a Hamiltonian cycle of weight $\leq K$ in $G$.

- HCW appears to “easier” than TSP as an H-cycle is not produced in the answer.
Chapter 34. NP-Completeness

Instead of considering

Travel Salesman Problem (TSP)
   \textbf{Input}: an edge-weighted graph } G = (V, E);  
   \textbf{Output}: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-Cycle Weight Decision (HCW)
   \textbf{Input}: an edge-weighted graph } G = (V, E), a weight value \( K \);  
   \textbf{Output}: “YES” if and only there is a Hamiltonian cycle of weight \( \leq K \) in } G.

- HCW appears to “easier” than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be “easier”
Instead of considering

**Travel Salesman Problem (TSP)**
- **Input**: an edge-weighted graph $G = (V, E)$;
- **Output**: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

**H-Cycle Weight Decision (HCW)**
- **Input**: an edge-weighted graph $G = (V, E)$, a weight value $K$;
- **Output**: “YES” if and only there is a Hamiltonian cycle of weight $\leq K$ in $G$.

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

- HCW appears to “easier” than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be “easier”
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.
Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Trivially,
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Trivially, P-time algorithms for TSP
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\implies$ P-time algorithms for HCW,
Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\implies$ P-time algorithms for HCW, why?
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\implies$ P-time algorithms for HCW, why?

How to prove:
Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\implies$ P-time algorithms for HCW, why?

How to prove: P-time algorithms for TSP
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\implies$ P-time algorithms for HCW, why?

How to prove: P-time algorithms for TSP $\iff$ P-time algorithms for HCW?
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof**: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof:** (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) =$ “YES/NO”
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof**: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) = \text{"YES/"NO}$
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof**: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) =$ “YES/“NO”
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

  1. on input $G$, for every possible values of $K$, call $A(G, K)$;
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof**: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) = \text{"YES/NO"}$
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G, K)$;
   remember the smallest $k_{min}$ such that $A(G, k_{min}) = \text{"YES"}$. 

How to make Step 1 P-time?
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof:** (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- Assume P-time algorithm $A$ for HCW such that $A(G, K) =$ “YES/NO”
- Construct a P-time algorithm $B(G)$ for TSP to behave as follows:

  1. on input $G$, for every possible values of $K$, call $A(G, K)$; remember the smallest $k_{\text{min}}$ such that $A(G, k_{\text{min}}) =$ “YES”.

  How to make Step 1 P-time?
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof**: (P-time algorithms for TSP \(\iff\) P-time algorithms for HCW)

- assume P-time algorithm \(A\) for HCW such that \(A(G, K) = \text{"YES/"NO}\)
- construct a P-time algorithm \(B(G)\) for TSP to behave as follows:

  1. on input \(G\), for every possible values of \(K\), call \(A(G, K)\);
     remember the smallest \(k_{min}\) such that \(A(G, k_{min}) = \text{"YES"}\).
  2. mark all edges in \(G\) as “unvisited”;

How to make Step 1 P-time?
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof:** (P-time algorithms for TSP ⇐ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) = \text{"YES/" NO}$
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G, K)$;
   remember the smallest $k_{min}$ such that $A(G, k_{min}) = \text{"YES"}$.

2. mark all edges in $G$ as “unvisited”;
   while there are “unvisited” edges in $G$
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof**: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G,K) = \text{"YES/"NO}$
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G,K)$;
   remember the smallest $k_{\text{min}}$ such that $A(G,k_{\text{min}}) = \text{"YES"}$.

2. mark all edges in $G$ as “unvisited”;
   while there are “unvisited” edges in $G$
   pick an “unvisited” edge $(u,v)$, mark it “visited”;

How to make Step 1 P-time?

Theorem 1 says problems HCW and TSP are “polynomially equivalent.”
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof:** (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) =$ “YES/“NO”
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

  1. on input $G$, for every possible values of $K$, call $A(G, K)$; remember the smallest $k_{min}$ such that $A(G, k_{min}) =$ “YES”.
  2. mark all edges in $G$ as “unvisited”; while there are “unvisited” edges in $G$
     - pick an “unvisited” edge $(u, v)$, mark it “visited”;
     - let $G' = G - \{(u, v)\}$;
Chapter 34. NP-Completeness

**Theorem 1**: HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof**: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- Assume P-time algorithm $A$ for HCW such that $A(G, k) =$ “YES/“NO”
- Construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. On input $G$, for every possible values of $K$, call $A(G, K)$; remember the smallest $k_{\text{min}}$ such that $A(G, k_{\text{min}}) =$ “YES”.

2. Mark all edges in $G$ as “unvisited”;
   - While there are “unvisited” edges in $G$
     - Pick an “unvisited” edge $(u, v)$, mark it “visited”;
     - Let $G' = G - \{(u, v)\}$;
     - If $A(G', k_{\text{min}}) =$ “YES”
Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) =$ “YES/NO”
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G, K)$;
   remember the smallest $k_{min}$ such that $A(G, k_{min}) =$ “YES”.
2. mark all edges in $G$ as “unvisited”;
   while there are “unvisited” edges in $G$
     pick an “unvisited” edge $(u, v)$, mark it “visited”;
     let $G' = G - \{(u, v)\}$;
     if $A(G', k_{min}) =$ “YES”
       then $G = G'$;
Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) =$ “YES/NO”
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G, K)$;
   remember the smallest $k_{min}$ such that $A(G, k_{min}) =$ “YES”.

2. mark all edges in $G$ as “unvisited”;
   while there are “unvisited” edges in $G$
     pick an “unvisited” edge $(u, v)$, mark it “visited”;
     let $G' = G - \{(u, v)\}$;
     if $A(G', k_{min}) =$ “YES”
       then $G = G'$;
     else mark $(u, v)$ “critical”;

How to make Step 1 P-time?

Theorem 1 says problems HCW and TSP are “polynomially equivalent.”
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof:** (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) = \text{"YES/"NO}$
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G, K)$; remember the smallest $k_{min}$ such that $A(G, k_{min}) = \text{"YES"}$.

2. mark all edges in $G$ as “unvisited”;
   while there are “unvisited” edges in $G$
   pick an “unvisited” edge $(u, v)$, mark it “visited”;
   let $G' = G - \{(u, v)\}$;
   if $A(G', k_{min}) = \text{"YES"}$
   then $G = G'$;
   else mark $(u, v)$ “critical”;
return (all “critical” edges)
Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- assume P-time algorithm $A$ for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G, K)$; remember the smallest $k_{min}$ such that $A(G, k_{min}) =$"YES".

2. mark all edges in $G$ as "unvisited"; while there are "unvisited" edges in $G$
   - pick an "unvisited" edge $(u, v)$, mark it "visited";
   - let $G' = G - \{(u, v)\}$;
   - if $A(G', k_{min}) =$ "YES"
     - then $G = G'$;
   - else mark $(u, v)$ "critical";
   - return (all "critical" edges)

- show algorithm $B$ runs in P-time.
Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time if and only if TSP is solvable in P-time.

Proof: (P-time algorithms for TSP \iff P-time algorithms for HCW)

• assume P-time algorithm $A$ for HCW such that $A(G, K) = \text{"YES"/\"NO"}$

• construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input $G$, for every possible values of $K$, call $A(G, K)$;
   remember the smallest $k_{\text{min}}$ such that $A(G, k_{\text{min}}) = \text{"YES"}$.

2. mark all edges in $G$ as “unvisited”;
   while there are “unvisited” edges in $G$
      pick an “unvisited” edge $(u, v)$, mark it “visited”;
      let $G' = G - \{ (u, v) \}$;
      if $A(G', k_{\text{min}}) = \text{"YES"}$
         then $G = G'$;
      else mark $(u, v)$ “critical”;
   return (all “critical” edges)

• show algorithm $B$ runs in P-time. How to make Step 1 P-time?
Chapter 34. NP-Completeness

**Theorem 1:** HCW is solvable in P-time if and only if TSP is solvable in P-time.

**Proof:** (P-time algorithms for TSP $\iff$ P-time algorithms for HCW)

- Assume P-time algorithm $A$ for HCW such that $A(G, K) = \text{"YES"/"NO"}$
- Construct a P-time algorithm $B(G)$ for TSP to behave as follows:

  1. On input $G$, for every possible values of $K$, call $A(G, K)$; remember the smallest $k_{\text{min}}$ such that $A(G, k_{\text{min}}) = \text{"YES"}$.
  2. Mark all edges in $G$ as “unvisited”; while there are “unvisited” edges in $G$
     - Pick an “unvisited” edge $(u, v)$, mark it “visited”;
     - Let $G' = G - \{(u, v)\}$;
     - If $A(G', k_{\text{min}}) = \text{"YES"}$
        - Then $G = G'$;
     - Else mark $(u, v)$ “critical”;
     - Return (all “critical” edges)

- Show algorithm $B$ runs in P-time. How to make Step 1 P-time?

Theorem 1 says problems HCW and TSP are “polynomially equivalent”.
Consider another related problem:

**H-Cycle Decision (HC)**

**Input:** an edge-weighted graph $G = (V,E)$;

**Output:** "YES" if and only there is a Hamiltonian cycle in $G$.

**H-Cycle Weight Decision (HCW)**

**Input:** an edge-weighted graph $G = (V,E)$, a weight value $K$;

**Output:** "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in $G$.

- Which problem is seemingly "easier"?
- **Theorem 2:** $HCW$ is P-time solvable if and only if $HC$ is P-time solvable.
- Can you prove it?
- Theorem 2 says problems $HCW$ and $HC$ are “polynomially equivalent”.
Consider another related problem:

**H-Cycle Decision (HC)**
- **Input:** an edge-weighted graph \( G = (V, E) \);
- **Output:** "YES" if and only there is a Hamiltonian cycle in \( G \).

Compared with **H-Cycle Weight Decision (HCW)**
- **Input:** an edge-weighted graph \( G = (V, E) \), a weight value \( K \);
- **Output:** "YES" if and only there is a Hamiltonian cycle of weight \( \leq K \) in \( G \).

• Which problem is seemingly "easier"?

**Theorem 2:** \( HCW \) is P-time solvable if and only if \( HC \) is P-time solvable.

Can you prove it?

**Theorem 2** says problems \( HCW \) and \( HC \) are "polynomially equivalent".
Chapter 34. NP-Completeness

Consider another related problem:

**H-Cycle Decision (HC)**
- **Input**: an edge-weighted graph \( G = (V, E) \);
- **Output**: “YES” if and only there is a Hamiltonian cycle in \( G \).
Consider another related problem:

**H-Cycle Decision (HC)**

**Input:** an edge-weighted graph $G = (V, E)$;

**Output:** “YES” if and only there is a Hamiltonian cycle in $G$.

Compared with
Consider another related problem:

**H-Cycle Decision (HC)**
- **Input**: an edge-weighted graph $G = (V, E)$;
- **Output**: “YES” if and only if there is a Hamiltonian cycle in $G$.

Compared with

**H-Cycle Weight Decision (HCW)**
- **Input**: an edge-weighted graph $G = (V, E)$, a weight value $K$;
- **Output**: “YES” if and only if there is a Hamiltonian cycle of weight $\leq K$ in $G$.

---

**Theorem 2**: $HCW$ is P-time solvable if and only if $HC$ is P-time solvable.

Can you prove it?

Theorem 2 says problems $HCW$ and $HC$ are “polynomially equivalent.”
Consider another related problem:

**H-Cycle Decision (HC)**
- **Input:** an edge-weighted graph $G = (V, E)$;
- **Output:** “YES” if and only there is a Hamiltonian cycle in $G$.

Compared with

**H-Cycle Weight Decision (HCW)**
- **Input:** an edge-weighted graph $G = (V, E)$, a weight value $K$;
- **Output:** “YES” if and only there is a Hamiltonian cycle of weight $\leq K$ in $G$.

- Which problem is seemingly “easier”?
Consider another related problem:

**H-Cycle Decision (HC)**
- **Input**: an edge-weighted graph $G = (V, E)$;
- **Output**: “YES” if and only there is a Hamiltonian cycle in $G$.

Compared with

**H-Cycle Weight Decision (HCW)**
- **Input**: an edge-weighted graph $G = (V, E)$, a weight value $K$;
- **Output**: “YES” if and only there is a Hamiltonian cycle of weight $\leq K$ in $G$.

- Which problem is seemingly “easier”?

**Theorem 2**: HCW is P-time solvable if and only if HC is P-time solvable.
Consider another related problem:

**H-Cycle Decision (HC)**
- **Input**: an edge-weighted graph $G = (V, E)$;
- **Output**: “YES” if and only there is a Hamiltonian cycle in $G$.

Compared with

**H-Cycle Weight Decision (HCW)**
- **Input**: an edge-weighted graph $G = (V, E)$, a weight value $K$;
- **Output**: “YES” if and only there is a Hamiltonian cycle of weight $\leq K$ in $G$.

- Which problem is seemingly “easier”?

**Theorem 2**: HCW is P-time solvable if and only if HC is P-time solvable.

Can you prove it?
Consider another related problem:

**H-Cycle Decision (HC)**
- **Input**: an edge-weighted graph \( G = (V, E) \);
- **Output**: “YES” if and only there is a Hamiltonian cycle in \( G \).

Compared with

**H-Cycle Weight Decision (HCW)**
- **Input**: an edge-weighted graph \( G = (V, E) \), a weight value \( K \);
- **Output**: “YES” if and only there is a Hamiltonian cycle of weight \( \leq K \) in \( G \).

- Which problem is seemingly “easier”?

**Theorem 2**: HCW is P-time solvable if and only if HC is P-time solvable.

Can you prove it?

Theorem 2 says problems HCW and HC are “polynomially equivalent”.
Chapter 34. NP-Completeness

**Corollary 3**: Problems TSP, HCW, and HC are all “polynomially equivalent”.
Chapter 34. NP-Completeness

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.
Corollary 3: Problems TSP, HCW, and HC are all “polynomially equivalent”.

There are other problems that have the similar situation.

Max Independent Set (MaxIS)
Input: graph $G = (V, E)$;
Corollary 3: Problems TSP, HCW, and HC are all “polynomially equivalent”.

There are other problems that have the similar situation.

Max Independent Set (MaxIS)
Input: graph $G = (V, E)$;
Output: an independent set of vertices of the maximum size;
Chapter 34. NP-Completeness

**Corollary 3**: Problems TSP, HCW, and HC are all “polynomially equivalent”.

There are other problems that have the similar situation.

**Max Independent Set (MaxIS)**
- **Input**: graph $G = (V, E)$;
- **Output**: an independent set of vertices of the maximum size;

**Independent Set (IS)**
- **Input**: graph $G = (V, E)$, integer $k$;
Corollary 3: Problems TSP, HCW, and HC are all “polynomially equivalent”.

There are other problems that have the similar situation.

Max Independent Set (MaxIS)
Input: graph $G = (V, E)$;
Output: an independent set of vertices of the maximum size;

Independent Set (IS)
Input: graph $G = (V, E)$, integer $k$;
Output: “YES” if and only if $G$ has an independent set of size $\geq k$. 
Chapter 34. NP-Completeness

**Corollary 3**: Problems TSP, HCW, and HC are all “polynomially equivalent”.

There are other problems that have the similar situation.

**Max Independent Set (MaxIS)**

**Input**: graph $G = (V, E)$;
**Output**: an independent set of vertices of the maximum size;

**Independent Set (IS)**

**Input**: graph $G = (V, E)$, integer $k$;
**Output**: “YES” if and only if $G$ has an independent set of size $\geq k$.

**Theorem 4**: MaxIS is P-time solvable if and only if IS is P-time solvable.
Chapter 34. NP-Completeness

**Corollary 3**: Problems TSP, HCW, and HC are all “polynomially equivalent”.

There are other problems that have the similar situation.

**Max Independent Set (MaxIS)**
- **Input**: graph $G = (V, E)$;
- **Output**: an independent set of vertices of the maximum size;

**Independent Set (IS)**
- **Input**: graph $G = (V, E)$, integer $k$;
- **Output**: “YES” if and only if $G$ has an independent set of size $\geq k$.

**Theorem 4**: MaxIS is P-time solvable if and only if IS is P-time solvable.

Can you prove the theorem?
Similarly,

Min Vertex Cover (MinVC)

Input: graph \( G = (V,E) \);
Output: a vertex cover set of vertices of the minimum size;

Vertex Cover (VC)

Input: graph \( G = (V,E) \), integer \( k \);
Output: "YES" if and only if \( G \) has a vertex cover of size \( \leq k \).

Theorem 5: MinVC is P-time solvable if and only if VC is P-time solvable.
Can you prove the theorem?
Similarly,

**Min Vertex Cover (MinVC)**

**Input:** graph $G = (V, E)$;
Chapter 34. NP-Completeness

Similarly,

**Min Vertex Cover (MinVC)**
*Input*: graph $G = (V, E)$;
*Output*: a vertex cover set of vertices of the minimum size;
Similarly,

**Min Vertex Cover (MinVC)**
**Input:** graph $G = (V, E)$;
**Output:** a vertex cover set of vertices of the minimum size;

**Vertex Cover (VC)**
**Input:** graph $G = (V, E)$, integer $k$;

Theorem 5: MinVC is P-time solvable if and only if VC is P-time solvable.
Can you prove the theorem?
Similarly,

**Min Vertex Cover (MinVC)**
- **Input**: graph $G = (V, E)$;
- **Output**: a vertex cover set of vertices of the minimum size;

**Vertex Cover (VC)**
- **Input**: graph $G = (V, E)$, integer $k$;
- **Output**: “YES” if and only if $G$ has a vertex cover of size $\leq k$. 

Theorem 5: MinVC is P-time solvable if and only if VC is P-time solvable.
Chapter 34. NP-Completeness

Similarly,

**Min Vertex Cover (MinVC)**
- **Input:** graph $G = (V, E)$;
- **Output:** a vertex cover set of vertices of the minimum size;

**Vertex Cover (VC)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has a vertex cover of size $\leq k$.

**Theorem 5:** MinVC is P-time solvable if and only if VC is P-time solvable.
Similarly,

**MIN VERTEX COVER (MinVC)**
- **Input**: graph $G = (V, E)$;
- **Output**: a vertex cover set of vertices of the minimum size;

**VERTEX COVER (VC)**
- **Input**: graph $G = (V, E)$, integer $k$;
- **Output**: “YES” if and only if $G$ has a vertex cover of size $\leq k$.

**Theorem 5**: MinVC is P-time solvable if and only if VC is P-time solvable.

Can you prove the theorem?
Chapter 34. NP-Completeness

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems. To study tractability of optimization problems, often it suffices to investigate decision problems. (Decision problems are also called languages.)

2. "Polynomial equivalency" can also be established between different decision problems, e.g., Corollary 6: $VC$ is P-time solvable if and only if $IS$ is P-time solvable.

3. However, "Polynomial equivalency" does not tell us the tractability of the problems.

4. We need a rigorous framework to study tractability via the notion "Polynomial equivalency".
Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.
Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.

To study tractability of optimization problems, often it suffices to investigate decision problems.
(Decision problems are also called languages.)
Chapter 34. NP-Completeness

Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.
   
   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called languages.)

2. “Polynomial equivalency” can also be established
Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.
   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called languages.)

2. “Polynomial equivalency” can also be established between different decision problems,
Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called languages.)

2. “Polynomial equivalency” can also be established between different decision problems, e.g.,
Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.
   
   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called languages.)

2. “Polynomial equivalency” can also be established between different decision problems, e.g.,
   
   **Corollary 6**: VC is P-time solvable if and only if IS is P-time solvable.
Chapter 34. NP-Completeness

Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called languages.)

2. “Polynomial equivalency” can also be established between different decision problems, e.g.,

   **Corollary 6**: VC is P-time solvable if and only if IS is P-time solvable.

3. However, “Polynomial equivalency” does not tell us the tractability of the problems.
Conclusions:

1. “Polynomial equivalency” can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called languages.)

2. “Polynomial equivalency” can also be established between different decision problems, e.g.,

   **Corollary 6**: VC is P-time solvable if and only if IS is P-time solvable.

3. However, “Polynomial equivalency” does not tell us the tractability of the problems.

4. We need a rigorous framework to study tractability via the notion “Polynomial equivalency”.


Chapter 34. NP-Completeness

2. Nondeterministic algorithms
2. Nondeterministic algorithms

Deterministic algorithms
2. Nondeterministic algorithms

Deterministic algorithms

- Given input data, a deterministic algorithm has its every step completely determined by the algorithm and data.
Chapter 34. NP-Completeness

2. Nondeterministic algorithms

Deterministic algorithms

• Given input data, a deterministic algorithm has its every step completely determined by the algorithm and data.

• All algorithms we have seen so far are deterministic.
Chapter 34. NP-Completeness

2. Nondeterministic algorithms

Deterministic algorithms

• Given input data, a deterministic algorithm has its every step completely determined by the algorithm and data.

• All algorithms we have seen so far are deterministic.

• Every deterministic algorithm can be unfolded into a linear sequence of steps (when the input is given).

```
M = -\infty
n = 3
i = 1
check 1 \leq 3
check -\infty < 10
M = 10
i = 2
check 2 \leq 3
check 10 < 30
M = 30
i = 3
check 3 \leq 3
check 30 < 20
i = 4
check 4 \leq 3
return (30)
```

MaxOfList(L)
1. \( M = -\infty \)
2. \( n = \text{length}(L) \)
3. \( \text{for } i = 1 \text{ to } n \)
4. \( \text{if } M < L[i] \)
5. \( M = L[i] \)
6. \( \text{return } (M) \)

Unfolded when input \( L = (10, 30, 20) \)
A deterministic algorithm can be thought of a linear path of steps;
A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.
Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.
Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.
Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a tree of steps.
Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a tree of steps.
- each step has more than one nondeterministic choice.
A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a tree of steps.
- each step has more than one nondeterministic choice.
- a path from root to a leaf is a sequence of nondeterministic choices;
A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one **nondeterministic** choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a tree of steps.
- each step has more than one nondeterministic choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers “YES” if one execution path leads to “YES”.
A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a tree of steps.
- each step has more than one nondeterministic choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers “YES” if one execution path leads to “YES”.
- the running time is the number of steps on a longest path.
Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a tree of steps.
- each step has more than one nondeterministic choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers “YES” if one execution path leads to “YES”.
- the running time is the number of steps on a longest path.
- if running time is $n$, there may be $\geq 2^n$ paths.
Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a linear path of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a tree of steps.
- each step has more than one nondeterministic choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers “YES” if one execution path leads to “YES”.
- the running time is the number of steps on a longest path.
- if running time is \( n \), there may be \( \geq 2^n \) paths.

Let us call this tree model of nondeterministic algorithms.
Chapter 34. NP-Completeness
Chapter 34. NP-Completeness

Use nondeterministic algorithms to solve problem SAT
Chapter 34. NP-Completeness

Use nondeterministic algorithms to solve problem SAT in polynomial time.
Use nondeterministic algorithms to solve problem SAT in polynomial time.

Algorithm **NonDetSAT-Solver**

**Input:** $\phi(x_1, \ldots, x_n)$

1. Let $\phi_0 = \phi(x_1, \ldots, x_n)$
2. for $i = 1$ to $n$
3. nondeterministically let $a_i = 0$ or $a_i = 1$
4. $\phi_i = \phi_{i-1}(x_i = a_i)$
5. if ($\phi_n == 1$)
6. return YES
7. else
8. return NO
Use **nondeterministic algorithms** to solve problem SAT in polynomial time.

**Algorithm** NonDetSAT-Solver

**Input:** $\phi(x_1, \ldots, x_n)$

1. Let $\phi_0 = \phi(x_1, \ldots, x_n)$
2. **for** $i = 1$ **to** $n$
3. nondeterministically let $a_i = 0$ or $a_i = 1$;
4. $\phi_i = \phi_{i-1}(x_i = a_i)$
5. **if** ($\phi_n == 1$)
6. **return** Yes
7. else
8. **return** No

**Algorithm** NonDetSAT-Solver-1

**Input:** $\phi(x_1, \ldots, x_n)$

1. **for** $i = 1$ **to** $n$
2. nondeterministically let $a_i = 0$ or $a_i = 1$;
3. **if** ($\phi(a_1, \ldots, a_n) == 1$)
4. **return** Yes
5. else
6. **return** No
Algorithm **NonDetSAT-Solver-1**

**Input:** $\phi(x_1, \ldots, x_n)$

1. for $i = 1$ to $n$
2. nondeterministically let $a_i = 0$ or $a_i = 1$;
3. if ($\phi(a_1, \ldots, a_n) == 1$)
4. return **YES**
5. else
6. return **NO**
Algorithm **NonDetSAT-Solver-1**

**Input:** $\phi(x_1, \ldots, x_n)$

1. **for** $i = 1$ **to** $n$
2.  nondeterministically let $a_i = 0$ or $a_i = 1$;
3.  **if** ($\phi(a_1, \ldots, a_n) == 1$)
4.  **return** YES
5.  **else**
6.  **return** NO
Chapter 34. NP-Completeness

Algorithm **NonDetSAT-Solver-1**

**Input:** $\phi(x_1, \ldots, x_n)$

1. for $i = 1$ to $n$
2.    nondeterministically let $a_i = 0$ or $a_i = 1$;
3. if $(\phi(a_1, \ldots, a_n) == 1)$
4.    return *Yes*
5. else
6.    return *No*

Answer is *Yes* iff $\exists A = (a_1, \ldots, a_n)$
Chapter 34. NP-Completeness

Algorithm NonDetSAT-Solver-1

Input: $\phi(x_1, \ldots, x_n)$

1. for $i = 1$ to $n$
2.    nondeterministically let $a_i = 0$ or $a_i = 1$;
3. if ($\phi(a_1, \ldots, a_n) == 1$)
4.    return YES
5. else
6.    return NO

Answer is YES iff $\exists A = (a_1, \ldots, a_n)$ $\phi(A) = 1$. 
Chapter 34. NP-Completeness

1. \( \text{Answer is } \text{Yes} \iff \exists A = (a_1,\ldots,a_n), \varphi(A) = 1. \)

2. \( \varphi(x_1,\ldots,x_n) \) is satisfiable \( \iff \exists \text{witness } A = (a_1,\ldots,a_n), \varphi(A) = 1 \) can be verified.

3. \( \varphi(x_1,\ldots,x_n) \) is satisfiable \( \iff \exists \text{witness } A, V(\varphi,A) \) can be verified to true in P-time.
(1) Answer is \textbf{YES}
(1) Answer is Yes iff $\exists A = (a_1, \ldots, a_n), \phi(A) = 1$. 

(2) $\phi(x_1, \ldots, x_n)$ is satisfiable iff $\exists$ witness $A = (a_1, \ldots, a_n), \phi(A) = 1$ can be verified.

(3) $\phi(x_1, \ldots, x_n)$ is satisfiable iff $\exists$ witness $A, V(\phi, A)$ can be verified to true in P-time.
(1) Answer is \textbf{YES} \textbf{iff} \( \exists A = (a_1, \ldots, a_n), \phi(A) = 1 \).

(2) \( \phi(x_1, \ldots, x_n) \) is satisfiable \textbf{iff}
Chapter 34. NP-Completeness

(1) Answer is **YES** iff \( \exists A = (a_1, \ldots, a_n), \phi(A) = 1 \).

(2) \( \phi(x_1, \ldots, x_n) \) is satisfiable

iff \( \exists \text{witness} \ A = (a_1, \ldots, a_n), \phi(A) = 1 \) can be verified
(1) Answer is \textbf{YES} iff $\exists A = (a_1, \ldots, a_n), \phi(A) = 1$.

(2) $\phi(x_1, \ldots, x_n)$ is satisfiable iff $\exists$ witness $A = (a_1, \ldots, a_n), \phi(A) = 1$ can be verified

(3) $\phi(x_1, \ldots, x_n)$ is satisfiable iff
(1) Answer is **YES** iff \( \exists A = (a_1, \ldots, a_n), \phi(A) = 1 \).

(2) \( \phi(x_1, \ldots, x_n) \) is satisfiable
   iff \( \exists \text{witness} A = (a_1, \ldots, a_n), \phi(A) = 1 \) can be verified

(3) \( \phi(x_1, \ldots, x_n) \) is satisfiable
   iff \( \exists \text{witness} A, V(\phi, A) \) can be verified to true in \( P \)-time
Use nondeterministic algorithms to solve problem Hamiltonian Cycle.
Chapter 34. NP-Completeness

Use nondeterministic algorithms to solve problem Hamiltonian Cycle in polynomial time.

![Diagram of a graph with vertices a, b, c, d and edges labeled with weights 1, 30, 99, 1, 30, 1]
Use nondeterministic algorithms to solve problem Hamiltonian Cycle in polynomial time.

(1) starting from any vertex $v$ in the graph;
Chapter 34. NP-Completeness

Use **nondeterministic algorithms** to solve problem **Hamiltonian Cycle** in polynomial time.

(1) starting from any vertex \( v \) in the graph;
(2) nondeterministically choose one of its (at most \( n - 1 \)) neighbors which has not been chosen;
Chapter 34. NP-Completeness

Use **nondeterministic algorithms** to solve problem **Hamiltonian Cycle** in polynomial time.

![Graph Diagram]

(1) starting from any vertex $v$ in the graph;
(2) nondeterministically choose one of its (at most $n - 1$) neighbors which has not been chosen;
   let the newly picked vertex be $v$, **go to** step (2)
Use nondeterministic algorithms to solve problem **Hamiltonian Cycle** in polynomial time.

1. starting from any vertex $v$ in the graph;
2. nondeterministically choose one of its (at most $n - 1$) neighbors which has not been chosen;
   - let the newly picked vertex be $v$, **go to** step (2)
3. if all vertices have been chosen,
   - return “YES” if their edges form an H-cycle;
   - return “NO” if their edges do NOT form an H-cycle;
Chapter 34. NP-Completeness

Use **nondeterministic algorithms** to solve problem **Hamiltonian Cycle** in polynomial time.

(1) starting from any vertex $v$ in the graph;
(2) nondeterministically choose one of its (at most $n - 1$) neighbors which has not been chosen;
   let the newly picked vertex be $v$, go to step (2)
(3) if all vertices have been chosen,
   return “YES” if their edges form an H-cycle;
   return “NO” if their edges do NOT form an H-cycle;

• The algorithm will answer “YES” iff there is a H-cycle in $G$. 
Use nondeterministic algorithms to solve problem Hamiltonian Cycle in polynomial time.

(1) starting from any vertex \( v \) in the graph;
(2) nondeterministically choose one of its (at most \( n - 1 \)) neighbors which has not been chosen;
   let the newly picked vertex be \( v \), go to step (2)
(3) if all vertices have been chosen,
   return “YES” if their edges form an H-cycle;
   return “NO” if their edges do NOT form an H-cycle;

• The algorithm will answer “YES” iff there is a H-cycle in \( G \).
  Because each path try one permutation of vertices.
Use nondeterministic algorithms to solve problem Hamiltonian Cycle in polynomial time.

(1) starting from any vertex \( v \) in the graph;
(2) nondeterministically choose one of its (at most \( n - 1 \)) neighbors which has not been chosen;
   let the newly picked vertex be \( v \), go to step (2)
(3) if all vertices have been chosen,
   return “YES” if their edges form an H-cycle;
   return “NO” if their edges do NOT form an H-cycle;

- The algorithm will answer “YES” iff there is a H-cycle in \( G \).
  Because each path try one permutation of vertices.
- The algorithm runs in polynomial time as each path takes \( O(n) \) steps.
Problems like **Independent Set**, **Vertex Cover**, **HCW** can all be solved with **nondeterministic algorithms** in **polynomial time**.
Problems like Independent Set, Vertex Cover, HCW can all be solved with nondeterministic algorithms in polynomial time.

Can you prove the claim?
Chapter 34. NP-Completeness

**Definition:** \(\mathcal{P}\) is the class of languages (i.e., decision problems) that can be solved by **deterministic polynomial-time algorithms**.
Chapter 34. NP-Completeness

**Definition:** $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like **Reachability**
Chapter 34. NP-Completeness

**Definition:** \( \mathcal{P} \) is the class of languages (i.e., decision problems) that can be solved by *deterministic polynomial-time algorithms*.

- class \( \mathcal{P} \) contains problems like \textsc{Reachability} and many others.
Chapter 34. NP-Completeness

**Definition:** $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like \textsc{Reachability} and many others.

**Definition:** $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $\mathcal{NP}$ contains problems like \textsc{VC}, \textsc{HC}, \textsc{IS} and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm, $\mathcal{P} \subseteq \mathcal{NP}$.
Chapter 34. NP-Completeness

**Definition:** \( \mathcal{P} \) is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class \( \mathcal{P} \) contains problems like Reachability and many others.

**Definition:** \( \mathcal{NP} \) is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class \( \mathcal{NP} \) contains problems like VC, HC, IS and many others.
Chapter 34. NP-Completeness

**Definition:** $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by **deterministic polynomial-time algorithms**.

- class $\mathcal{P}$ contains problems like **Reachability** and many others.

**Definition:** $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by **nondeterministic polynomial-time algorithms**.

- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

$$\mathcal{P} \subseteq \mathcal{NP}$$
Chapter 34. NP-Completeness

**Definition**: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like \textsc{Reachability} and many others.

**Definition**: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $\mathcal{NP}$ contains problems like \textsc{VC}, \textsc{HC}, \textsc{IS} and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

\[
\mathcal{P} \subseteq \mathcal{NP}
\]
Chapter 34. NP-Completeness

**Definition:** $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like Reachability and many others.

**Definition:** $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

$$\mathcal{P} \subseteq \mathcal{NP}$$

\[\text{WANTED} \quad \mathcal{P} \neq \mathcal{NP} \quad \text{REWARD} \]$1,000,000
Definition: \( \mathcal{P} \) is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class \( \mathcal{P} \) contains problems like Reachability and many others.

Definition: \( \mathcal{NP} \) is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class \( \mathcal{NP} \) contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

\[ \mathcal{P} \subseteq \mathcal{NP} \]
Chapter 34. NP-Completeness

**Definition:** $P$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $P$ contains problems like Reachability and many others.

**Definition:** $NP$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $NP$ contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

\[ P \subseteq NP \]
Chapter 34. NP-Completeness
Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

• we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
• each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.
• we can assume the algorithm does all nondeterministic choices before other operations.

So we can model the computation as

1. first choose a binary string nondeterministically, and
2. follow the specified path deterministically

The binary string is called **certificate** or **witness**; The deterministic computation part is called **verification**. Deterministic algorithms are when the certificate is empty.
We consider the tree model of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices
Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string:
Chapter 34. NP-Completeness

We consider the tree model of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices
  (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string:
  0 for branching left, 1 for right.
We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.
- we can assume the algorithm does all nondeterministic choices **before** other operations.

The binary string is called **certificate** or **witness**; the deterministic computation part is called **verification**.
We consider the tree model of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)

- each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.

- we can assume the algorithm does all nondeterministic choices before other operations. So we can model the computation as
Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.
- we can assume the algorithm does all nondeterministic choices **before** other operations. So we can model the computation as
We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices  
  (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string:  
  0 for branching left, 1 for right.
- we can assume the algorithm does **all** nondeterministic choices **before**  
  other operations. So we can model the computation as

  (1) first choose a binary string nondeterministically, and
We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.
- we can assume the algorithm does **all** nondeterministic choices **before** other operations. So we can model the computation as

  (1) first choose a binary string nondeterministically, and
  (2) follow the specified path deterministically
Chapter 34. NP-Completeness

We consider the tree model of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices
  (5 choices can be simulated with 4 nondeterministic steps)

- each nondeterministic path can be represented with a binary string:
  0 for branching left, 1 for right.

- we can assume the algorithm does all nondeterministic choices before
  other operations. So we can model the computation as

  (1) first choose a binary string nondeterministically, and
  (2) follow the specified path deterministically

The binary string is called certificate or witness;
The deterministic computation part is called verification.
We consider the tree model of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)

- each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.

- we can assume the algorithm does all nondeterministic choices before other operations. So we can model the computation as
  1. first choose a binary string nondeterministically, and
  2. follow the specified path deterministically

    The binary string is called certificate or witness; The deterministic computation part is called verification.

Deterministic algorithms are when the certificate is empty.
Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation
Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is
Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

• to nondeterministically choose a binary string of a polynomial length,
Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \text{NP}$. Then there is a deterministic algorithm $A_\Pi$, and a constant $c > 0$, such that

1. if $x$ is a positive instance of $\Pi$, there is a binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"YES"}$;
2. if $x$ is a negative instance of $\Pi$, for all binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"NO"}$;

and $A_\Pi$ runs in time $O(n^c)$.

We call $y$ a certificate/witness and $A_\Pi$ the verification algorithm.

$P$ is defined with certificate $y = \epsilon$, i.e., empty string.
Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a deterministic algorithm $A_\Pi$, and a constant $c > 0$, such that

1. if $x$ is a positive instance of $\Pi$, there is a binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"YES"}$;
2. if $x$ is a negative instance of $\Pi$, for all binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"NO"}$;

and $A_\Pi$ runs in time $O(n^c)$. We call $y$ a certificate/witness and $A_\Pi$ the verification algorithm.

$P$ is defined with certificate $y = \epsilon$, i.e., empty string.
Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let \( \Pi \in \mathcal{NP} \). Then there is a deterministic algorithm \( A_\Pi \), and a constant \( c > 0 \), such that

1. if \( x \) is a positive instance of \( \Pi \), there is a binary string \( y \) of length \( n^c \), \( A_\Pi(x, y) = \text{“YES”} \);
Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

• to nondeterministically choose a binary string of a polynomial length,
• then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a deterministic algorithm $A_{\Pi}$, and a constant $c > 0$, such that

(1) if $x$ is a positive instance of $\Pi$, there is a binary string $y$ of length $n^c$,
   $A_{\Pi}(x, y) = \text{"YES"}$;
(2) if $x$ is a negative instance of $\Pi$, for all binary string $y$ of length $n^c$,
   $A_{\Pi}(x, y) = \text{"NO"}$;

$P$ is defined with certificate $y = \epsilon$, i.e., empty string.
Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a deterministic algorithm $A_\Pi$, and a constant $c > 0$, such that

1. if $x$ is a positive instance of $\Pi$, there is a binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"YES"}$;
2. if $x$ is a negative instance of $\Pi$, for all binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"NO"}$;

and $A_\Pi$ runs in time $O(n^c)$. 
Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let \( \Pi \in \mathcal{NP} \). Then there is a deterministic algorithm \( A_\Pi \), and a constant \( c > 0 \), such that

1. if \( x \) is a positive instance of \( \Pi \), there is a binary string \( y \) of length \( n^c \),
   \( A_\Pi(x, y) = \text{"YES";} \)
2. if \( x \) is a negative instance of \( \Pi \), for all binary string \( y \) of length \( n^c \),
   \( A_\Pi(x, y) = \text{"NO";} \)
and \( A_\Pi \) runs in time \( O(n^c) \).

We call \( y \) a certificate/witness and \( A_\Pi \) the verification algorithm.
Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a deterministic algorithm $A_\Pi$, and a constant $c > 0$, such that

1. if $x$ is a positive instance of $\Pi$, there is a binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"YES"}$;
2. if $x$ is a negative instance of $\Pi$, for all binary string $y$ of length $n^c$, $A_\Pi(x, y) = \text{"NO"}$;

and $A_\Pi$ runs in time $O(n^c)$.

We call $y$ a certificate/witness and $A_\Pi$ the verification algorithm.

$\mathcal{P}$ is defined with certificate $y = \epsilon$, i.e., empty string.
Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$, $x \in L \iff \exists y, |y| \leq |x|^c, A_L(x,y) = 1$ and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what?
in $m = |x,y| = |x| + |y| \leq n + n^c$, so if $A_L$ runs in polynomial time $m^d \leq (n + n^c)^d = O(n^{dc})$, also polynomial time of $n = |x|$. 

Class $\mathcal{P}$ is defined with certificate $y = \epsilon$, i.e., empty string.
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. 

...
Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a 
**deterministic** algorithm $A_L$, 

$A_L$ runs in polynomial time of $m = |x,y| = |x| + |y| \leq n + nc$. So if $A_L$ runs in polynomial time $m^d \leq (2^nc)^d = O(n^dc)$, also polynomial time of $n = |x|$. 
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a \textbf{deterministic} algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,
Chapter 34. NP-Completeness

**Definition of \( \mathcal{NP} \) in terms of languages:**

Let \( L \subseteq \{0, 1\}^* \) be a language in the class \( \mathcal{NP} \). Then there is a **deterministic** algorithm \( A_L \), and a constant \( c > 0 \), such that, for every \( x \in \{0, 1\}^* \),

\[
    x \in L
\]
Definition of $NP$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $NP$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y,$$
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c,$$
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$
Definition of \( \mathcal{NP} \) in terms of languages:

Let \( L \subseteq \{0, 1\}^* \) be a language in the class \( \mathcal{NP} \). Then there is a **deterministic** algorithm \( A_L \), and a constant \( c > 0 \), such that, for every \( x \in \{0, 1\}^* \),

\[
x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1
\]

and \( A_L \) runs in polynomial time.
Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time
Definition of \( \mathcal{NP} \) in terms of languages:

Let \( L \subseteq \{0, 1\}^* \) be a language in the class \( \mathcal{NP} \). Then there is a deterministic algorithm \( A_L \), and a constant \( c > 0 \), such that, for every \( x \in \{0, 1\}^* \),

\[
 x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1
\]

and \( A_L \) runs in polynomial time.

\( A_L \) runs in polynomial time of what?

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y|$
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a \textbf{deterministic} algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y| = |x| + |y|$
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y| = |x| + |y| \leq n + n^c$. 

Definition of $NP$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $NP$. Then there is a \textbf{deterministic} algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y| = |x| + |y| \leq n + nc$.
So if $A_L$ runs in polynomial time $m^d$.
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y| = |x| + |y| \leq n + n^c$.

So if $A_L$ runs in polynomial time $m^d \leq (n + n^c)$
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y| = |x| + |y| \leq n + n^c$.

So if $A_L$ runs in polynomial time $m^d \leq (n + n^c)^d$. 
Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y| = |x| + |y| \leq n + n^c$.

So if $A_L$ runs in polynomial time $m^d \leq (n + n^c)^d \leq (2n^c)^d$.
Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0, 1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

$A_L$ runs in polynomial time of what? in $m = |x, y| = |x| + |y| \leq n + n^c$.

So if $A_L$ runs in polynomial time $m^d \leq (n + n^c)^d \leq (2n^c)^d = O(n^{dc})$,
Chapter 34. NP-Completeness

Definition of \( \mathcal{NP} \) in terms of languages:

Let \( L \subseteq \{0,1\}^* \) be a language in the class \( \mathcal{NP} \). Then there is a deterministic algorithm \( A_L \), and a constant \( c > 0 \), such that, for every \( x \in \{0,1\}^* \),

\[
x \in L \iff \exists y, |y| \leq |x|^c, A_L(x,y) = 1
\]

and \( A_L \) runs in polynomial time.

\( A_L \) runs in polynomial time of what? in \( m = |x,y| = |x| + |y| \leq n + nc \).

So if \( A_L \) runs in polynomial time \( m^d \leq (n + nc)^d \leq (2nc)^d = O(n^{dc}) \), also polynomial time of \( n = |x| \).
Chapter 34. NP-Completeness

Definition of \( \mathcal{NP} \) in terms of languages:

Let \( L \subseteq \{0, 1\}^* \) be a language in the class \( \mathcal{NP} \). Then there is a deterministic algorithm \( A_L \), and a constant \( c > 0 \), such that, for every \( x \in \{0, 1\}^* \),

\[
x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1
\]

and \( A_L \) runs in polynomial time.

\( A_L \) runs in polynomial time of what? in \( m = |x, y| = |x| + |y| \leq n + nc \).

So if \( A_L \) runs in polynomial time \( m^d \leq (n + nc)^d \leq (2nc)^d = O(nc^d) \),

also polynomial time of \( n = |x| \).

Class \( \mathcal{P} \) is defined with certificate \( y = \epsilon \), i.e., empty string.
Chapter 34. NP-Completeness

The diagram illustrates concepts related to nondeterministic and deterministic moves in computational graphs or automata.

- **nondet. moves**: Directed edges representing nondeterministic transitions, allowing multiple possible paths from a given state.
- **det. moves**: Directed edges representing deterministic transitions, showing a single, fixed path from a given state.

The arrows between nodes indicate the flow of transitions, with labeled states at the end of each node indicating the outcomes (Y for Yes, N for No) of the computational process.
Chapter 34. NP-Completeness
Chapter 34. NP-Completeness

Proof that $HC \in \mathcal{NP}$.

Certificate $y$ represents a sequence of ordered vertices; the algorithm $A$ is to verify that $y$ does form a $H$-cycle.

Details:
- $y = B_1 B_2 ... B_n$, where $B_i$ is a binary representation of some vertex in $G$.
- How many bits does $B_i$ need? $\lceil \log_2 n \rceil$.
- Whether $y$ forms a $H$-cycle can be verified in time $O(|E|)$. 


Proof that $HC \in NP$.

We need to show there is a deterministic algorithm $A$ and a constant $c > 0$, such that for any $G$, 

$$G \in HC \iff \exists y, |y| \leq |G|^c, A(G, y) = \text{"YES"}$$

We can design that

- certificate $y$ represents a sequence of ordered vertices;
Proof that $HC \in NP$.

We need to show there is a deterministic algorithm $A$ and a constant $c > 0$, such that for any $G$,

$$G \in HC \iff \exists y, |y| \leq |G|^c, A(G, y) = \text{"YES"}$$

We can design that

- certificate $y$ represents a sequence of ordered vertices;
- algorithm $A$ is to verify that $y$ does form a H-cycle.
Proof that $HC \in NP$.

We need to show there is a deterministic algorithm $A$ and a constant $c > 0$, such that for any $G$,

$$G \in HC \iff \exists y, |y| \leq |G|^c, A(G, y) = "YES"$$

We can design that
- certificate $y$ represents a sequence of ordered vertices;
- algorithm $A$ is to verify that $y$ does form a H-cycle.

Details:
- $y = B_1 B_2 \ldots B_n$, where $B_i$ is a binary representation of some vertex in $G$;
Proof that $HC \in \mathcal{NP}$.

We need to show there is a deterministic algorithm $A$ and a constant $c > 0$, such that for any $G$,

$$G \in HC \iff \exists y, |y| \leq |G|^c, A(G, y) = "YES"$$

We can design that

- certificate $y$ represents a sequence of ordered vertices;

- algorithm $A$ is to verify that $y$ does form a H-cycle.

Details:

- $y = B_1B_2\ldots B_n$, where $B_i$ is a binary representation of some vertex in $G$; How many bits does $B_i$ need?
Chapter 34. NP-Completeness

Proof that HC ∈ NP.

We need to show there is a deterministic algorithm A and a constant c > 0, such that for any G,

\[ G \in \text{HC} \iff \exists y, |y| \leq |G|^c, A(G, y) = \text{"YES"} \]

We can design that

• certificate y represents a sequence of ordered vertices;

• algorithm A is to verify that y does form a H-cycle.

Details:

• y = B_1 B_2 \ldots B_n, where B_i is a binary representation of some vertex in G; How many bits does B_i need? \lceil \log_2 n \rceil
Proof that $HC \in \mathcal{NP}$.

We need to show there is a deterministic algorithm $A$ and a constant $c > 0$, such that for any $G$,

$$G \in HC \iff \exists y, |y| \leq |G|^c, A(G, y) = "YES"$$

We can design that

- certificate $y$ represents a sequence of ordered vertices;

- algorithm $A$ is to verify that $y$ does form a H-cycle.

Details:

- $y = B_1 B_2 \ldots B_n$, where $B_i$ is a binary representation of some vertex in $G$; How many bits does $B_i$ need? $\lceil \log_2 n \rceil$

- whether $y$ forms a H-cycle can be verified in time
Proof that $HC \in \mathcal{NP}$.

We need to show there is a deterministic algorithm $A$ and a constant $c > 0$, such that for any $G$,

$$G \in HC \iff \exists y, |y| \leq |G|^c, A(G, y) = \text{"YES"}$$

We can design that

- certificate $y$ represents a sequence of ordered vertices;

- algorithm $A$ is to verify that $y$ does form a H-cycle.

Details:

- $y = B_1 B_2 \ldots B_n$, where $B_i$ is a binary representation of some vertex in $G$; How many bits does $B_i$ need? $\lceil \log_2 n \rceil$

- whether $y$ forms a H-cycle can be verified in time $O(|E|)$
Chapter 34. NP-Completeness

exercises:

1. to prove a language is in the class $NP$ by no mean to prove that the language can be solved in polynomial time. Instead, it only shows the language is in the class $NP$.

2. there is a difference between deciding $x \in L$ and checking $A_L(x, y) = 1$.

3. as between convicting a suspect vs checking an evidence against the suspect.
exercises:

Proof that **Independent Set** $\in \mathcal{NP}$.

Proof that **Vertex Cover** $\in \mathcal{NP}$.

Notes
exercises:

Proof that \textit{Independent Set} $\in \mathcal{NP}$.

Proof that \textit{Vertex Cover} $\in \mathcal{NP}$.

Notes

1. to prove a language is in the class $\mathcal{NP}$ by no mean to prove that the language can be solved in polynomial time. Instead, it only shows the language is in the class $\mathcal{NP}$. 
exercises:

Proof that Independent Set $\in \mathcal{NP}$.

Proof that Vertex Cover $\in \mathcal{NP}$.

Notes

1. to prove a language is in the class $\mathcal{NP}$ by no mean to prove that the language can be solved in polynomial time. Instead, it only shows the language is in the class $\mathcal{NP}$.

2. there is a difference between deciding $x \in L$ and checking $A_L(x, y) = 1$. 
Chapter 34. NP-Completeness

exercises:

Proof that Independent Set $\in \mathcal{NP}$.

Proof that Vertex Cover $\in \mathcal{NP}$.

Notes

1. to prove a language is in the class $\mathcal{NP}$ by no mean to prove that the language can be solved in polynomial time. Instead, it only shows the language is in the class $\mathcal{NP}$.

2. there is a difference between deciding $x \in L$ and checking $A_L(x, y) = 1$.

3. as between convicting a suspect vs checking an evidence against the suspect.
3. NP-Completeness Framework
3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages
3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
Chapter 34. NP-Completeness

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

\[ \overline{L} = \{ x : x \notin L \} \]
Chapter 34. NP-Completeness

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

\[
\overline{L} = \{x : x \not\in L\}
\]

called complement of \(L\)
3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

\[ \overline{L} = \{ x : x \notin L \} \] called complement of \( L \)

\[ L \cup \overline{L} = \{0, 1\}^* \]
3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains **positive instances** of the corresponding decision problem.
- Define

\[ \overline{L} = \{ x : x \not\in L \} \] called *complement* of \( L \)

\[ L \cup \overline{L} = \{0, 1\}^* = \mathcal{U}, \]
3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define $\overline{L} = \{x : x \notin L\}$ called complement of $L$

$$L \cup \overline{L} = \{0, 1\}^* = \mathcal{U}, \text{ called universe}$$
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet \( \{0, 1\} \).

A reduction from $L_1$ to $L_2$, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$,
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.

A reduction from $L_1$ to $L_2$, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.

A reduction from $L_1$ to $L_2$, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.

A reduction from $L_1$ to $L_2$, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.

A reduction from $L_1$ to $L_2$, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.

A reduction from $L_1$ to $L_2$, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

That is,

$$x \in L_1 \implies f(x) \in L_2;$$
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.

A reduction from $L_1$ to $L_2$, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

That is,

$$x \in L_1 \implies f(x) \in L_2; \quad x \in \overline{L_1} \implies f(x) \in \overline{L_2};$$
Chapter 34. NP-Completeness

Let $L_1$ and $L_2$ are two languages over the alphabet $\{0, 1\}$.

A **reduction from $L_1$ to $L_2$**, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

That is,

$$x \in L_1 \implies f(x) \in L_2; \quad x \in \overline{L_1} \implies f(x) \in \overline{L_2};$$
Chapter 34. NP-Completeness

Two example problems:
Two example problems:

**Independent Set (IS)**

**Input:** graph $G = (V, E)$, integer $k$;
Two example problems:

**Independent Set (IS)**

**Input:** graph $G = (V, E)$, integer $k$;

**Output:** “YES” if and only if $G$ has an independent set of size $\geq k$. 
Chapter 34. NP-Completeness

Two example problems:

**INDEPENDENT SET (IS)**

**INPUT:** graph $G = (V, E)$, integer $k$;

**OUTPUT:** “YES” if and only if $G$ has an independent set of size $\geq k$.

and
Two example problems:

**Independent Set (IS)**
- **Input:** graph \( G = (V, E) \), integer \( k \);
- **Output:** “YES” if and only if \( G \) has an independent set of size \( \geq k \).

and

**Vertex Cover (VC)**
- **Input:** graph \( G = (V, E) \), integer \( k \);
Two example problems:

**INDEPENDENT SET (IS)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has an independent set of size $\geq k$.

and

**VERTEX COVER (VC)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has a vertex cover of size $\leq k$. 

Consider their corresponding languages:

$L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \}$

$L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \}$
Two example problems:

**INDEPENDENT SET (IS)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has an independent set of size $\geq k$.

and

**VERTEX COVER (VC)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has a vertex cover of size $\leq k$.

Consider their corresponding languages:
Chapter 34. NP-Completeness

Two example problems:

**INDEPENDENT SET (IS)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has an independent set of size $\geq k$.

and

**VERTEX COVER (VC)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has a vertex cover of size $\leq k$.

Consider their corresponding languages:

$$L_{IS} = \{\langle G, k \rangle : G \text{ has an independent set of size } \geq k\}$$
Chapter 34. NP-Completeness

Two example problems:

**Independent Set (IS)**

**Input:** graph $G = (V, E)$, integer $k$;

**Output:** “YES” if and only if $G$ has an independent set of size $\geq k$.

and

**Vertex Cover (VC)**

**Input:** graph $G = (V, E)$, integer $k$;

**Output:** “YES” if and only if $G$ has a vertex cover of size $\leq k$.

Consider their corresponding languages:

$$L_{IS} = \{\langle G, k \rangle : G \text{ has an independent set of size } \geq k \}$$

$$L_{VC} = \{\langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \}$$
Two example problems:

**INDEPENDENT SET (IS)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has an independent set of size $\geq k$.

and

**VERTEX COVER (VC)**
- **Input:** graph $G = (V, E)$, integer $k$;
- **Output:** “YES” if and only if $G$ has a vertex cover of size $\leq k$.

Consider their corresponding languages:

$$L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \}$$

$$L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \}$$
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]
Chapter 34. NP-Completeness

\[ L_{IS} = \{\langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{\langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

**Theorem:**

\[ L_{IS} \leq L_{VC} \]

**Proof:**

we use the fact that complement set of an independent set is a vertex cover in the same graph.

We construct a mapping \( f \) that maps instance \( \langle G, k \rangle \) to instance \( \langle G, |G| - k \rangle \), i.e.,

\[ f(\langle G, k \rangle) = \langle G, |G| - k \rangle \]

This is a reduction from \( L_{IS} \) to \( L_{VC} \).

**Claim:**

\( G \) has an i.s. of size \( \geq k \) \( \iff \) \( G \) has an v.c. of size \( \leq |G| - k \)

(Proof of the claim is on the next slide)

So \( L_{IS} \leq L_{VC} \).
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

\[ L_{IS} \leq L_{VC} \]
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

**Theorem:** \( L_{IS} \leq L_{VC} \)
Chapter 34. NP-Completeness

$L_{IS} = \{\langle G, k \rangle : G \text{ has an independent set of size } \geq k \}$

$L_{VC} = \{\langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \}$

Because the two problems are very relevant to each other, we have:

**Theorem:** $L_{IS} \leq L_{VC}$

**Proof:**
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

**Theorem:** \( L_{IS} \leq L_{VC} \)

**Proof:** we use the fact that complement set of an independent set is a vertex cover in the same graph
Chapter 34. NP-Completeness

$L_{IS} = \{\langle G, k \rangle : G \text{ has an independent set of size } \geq k\}$

$L_{VC} = \{\langle G, k \rangle : G \text{ has a vertex cover of size } \leq k\}$

Because the two problems are very relevant to each other, we have:

**Theorem:** $L_{IS} \leq L_{VC}$

**Proof:** we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping $f$ that maps instance $\langle G, k \rangle$ to instance $\langle G, |G| - k \rangle$,
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

**Theorem**: \( L_{IS} \leq L_{VC} \)

**Proof**: we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping \( f \) that maps instance \( \langle G, k \rangle \) to instance \( \langle G, |G| - k \rangle \), i.e.,

\[ f(\langle G, k \rangle) \]
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

**Theorem:** \( L_{IS} \leq L_{VC} \)

**Proof:** we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping \( f \) that maps instance \( \langle G, k \rangle \) to instance \( \langle G, |G| - k \rangle \), i.e.,

\[ f(\langle G, k \rangle) = \langle G, |G| - k \rangle \]

This is a reduction from \( L_{IS} \) to \( L_{VC} \).
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

**Theorem:** \( L_{IS} \leq L_{VC} \)

**Proof:** we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping \( f \) that maps instance \( \langle G, k \rangle \) to instance \( \langle G, |G| - k \rangle \), i.e.,

\[ f(\langle G, k \rangle) = \langle G, |G| - k \rangle \]

This is a reduction from \( L_{IS} \) to \( L_{VC} \)

**Claim:**

\( G \) has an i.s. of size \( \geq k \) \iff \( G \) has an v.c. of size \( \leq |G| - k \)
Chapter 34. NP-Completeness

\[ L_{IS} = \{ \langle G, k \rangle : G \text{ has an independent set of size } \geq k \} \]

\[ L_{VC} = \{ \langle G, k \rangle : G \text{ has a vertex cover of size } \leq k \} \]

Because the two problems are very relevant to each other, we have:

**Theorem:** \( L_{IS} \leq L_{VC} \)

**Proof:** we use the fact that complement set of an independent set is a vertex cover in the same graph.

We construct a mapping \( f \) that maps instance \( \langle G, k \rangle \) to instance \( \langle G, |G| - k \rangle \), i.e.,

\[ f(\langle G, k \rangle) = \langle G, |G| - k \rangle \]

This is a reduction from \( L_{IS} \) to \( L_{VC} \)

**Claim:**

G has an i.s. of size \( \geq k \) \( \iff \) G has an v.c. of size \( \leq |G| - k \)

(proof of the claim is on the next slide)

So \( L_{IS} \leq L_{VC} \).
Chapter 34. NP-Completeness

Proof: “⇒”
(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)

Let $G$ be such that has vertices $V = \{v_1,\ldots,v_n\}$. Assume that $G$ has a i.s. of size $k_0$ for some $k_0 \geq k$. We further assume, without loss of generality, the i.s include vertices $\{v_1,\ldots,v_{k_0}\}$. Then vertices $\{v_{k_0+1},\ldots,v_n\}$ form a v.c. for $G$.

Suppose otherwise, $\exists$ edge $(u,v)$ that is not covered, i.e., neither $u \in \{v_{k_0+1},\ldots,v_n\}$ nor $v \in \{v_{k_0+1},\ldots,v_n\}$. Thus, $u,v \in \{v_1,\ldots,v_{k_0}\}$, the independent set. But $(u,v)$ is an edge, contradicts that $\{v_1,\ldots,v_{k_0}\}$ is an i.s.

Can you prove “⇐”? 

Chapter 34. NP-Completeness

Proof: “⇒”
(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)
Let $G$ be such that has vertices $V = \{v_1, \ldots, v_n\}$. 

Proof: “⇐”
(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)
Let $G$ be such that has vertices $V = \{v_1, \ldots, v_n\}$. 

Chapter 34. NP-Completeness

**Proof:** “⇒”

(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)

Let $G$ be such that has vertices $V = \{v_1, \ldots, v_n\}$.

Assume that $G$ has a i.s. of size $k_0$ for some $k_0 \geq k$.

We further assume, without loss of generality, the i.s include vertices $\{v_1, \ldots, v_{k_0}\}$. 

Can you prove “⇐”? 
Proof: “⇒”
(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)

Let $G$ be such that has vertices $V = \{v_1, \ldots, v_n\}$.

Assume that $G$ has a i.s. of size $k_0$ for some $k_0 \geq k$.
We further assume, without loss of generality, the i.s include vertices
$\{v_1, \ldots, v_{k_0}\}$.

Then vertices $\{v_{k_0+1}, \ldots, v_n\}$ form a v.c. for $G$. 

Proof: “⇒”
(to prove that \( G \) has i.s. of size \( \geq k \) implies \( G \) has v.c of size \( \geq k \))

Let \( G \) be such that has vertices \( V = \{v_1, \ldots, v_n\} \).

Assume that \( G \) has a i.s. of size \( k_0 \) for some \( k_0 \geq k \).

We further assume, without loss of generality, the i.s include vertices \( \{v_1, \ldots, v_{k_0}\} \).

Then vertices \( \{v_{k_0+1}, \ldots, v_n\} \) form a v.c. for \( G \).

Suppose otherwise, \( \exists \) edge \((u, v)\) that is not covered, i.e.,

- neither \( u \in \{v_{k_0+1}, \ldots, v_n\} \)
- nor \( v \in \{v_{k_0+1}, \ldots, v_n\} \).
Proof: “⇒”
(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)

Let $G$ be such that has vertices $V = \{v_1, \ldots, v_n\}$.

Assume that $G$ has a i.s. of size $k_0$ for some $k_0 \geq k$.
We further assume, without loss of generality, the i.s include vertices
$\{v_1, \ldots, v_{k_0}\}$.

Then vertices $\{v_{k_0+1}, \ldots, v_n\}$ form a v.c. for $G$.

Suppose otherwise, $\exists$ edge $(u, v)$ that is not covered, i.e.,
neither $u \in \{v_{k_0+1}, \ldots, v_n\}$
nor $v \in \{v_{k_0+1}, \ldots, v_n\}$.

Thus, $u, v \in \{v_1, \ldots, v_{k_0}\}$, the independent set.
Proof: “⇒”
(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)

Let $G$ be such that has vertices $V = \{v_1, \ldots, v_n\}$.

Assume that $G$ has a i.s. of size $k_0$ for some $k_0 \geq k$.
We further assume, without loss of generality, the i.s include vertices $\{v_1, \ldots, v_{k_0}\}$.

Then vertices $\{v_{k_0+1}, \ldots, v_n\}$ form a v.c. for $G$.

Suppose otherwise, $\exists$ edge $(u, v)$ that is not covered, i.e.,
neither $u \in \{v_{k_0+1}, \ldots, v_n\}$
nor $v \in \{v_{k_0+1}, \ldots, v_n\}$.

Thus, $u, v \in \{v_1, \ldots, v_{k_0}\}$, the independent set.
But $(u, v)$ is an edge, contradicts that $\{v_1, \ldots, v_{k_0}\}$ is an i.s.
Chapter 34. NP-Completeness

Proof: “⇒”
(to prove that $G$ has i.s. of size $\geq k$ implies $G$ has v.c of size $\geq k$)

Let $G$ be such that has vertices $V = \{v_1, \ldots, v_n\}$.

Assume that $G$ has a i.s. of size $k_0$ for some $k_0 \geq k$.
We further assume, without loss of generality, the i.s include vertices $\{v_1, \ldots, v_{k_0}\}$.

Then vertices $\{v_{k_0+1}, \ldots, v_n\}$ form a v.c. for $G$.

Suppose otherwise, $\exists$ edge $(u,v)$ that is not covered, i.e.,
neither $u \in \{v_{k_0+1}, \ldots, v_n\}$
or $v \in \{v_{k_0+1}, \ldots, v_n\}$.

Thus, $u,v \in \{v_1, \ldots, v_{k_0}\}$, the independent set.
But $(u,v)$ is an edge, contradicts that $\{v_1, \ldots, v_{k_0}\}$ is an i.s.

Can you prove “⇐”??
Chapter 34. NP-Completeness

An important motivation for reduction:
Chapter 34. NP-Completeness

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;
Chapter 34. NP-Completeness

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;
- algorithms solving the second problem can be used to solve the first;

where algorithm $F$ computes the reduction $f$,
Chapter 34. NP-Completeness

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;
- algorithms solving the second problem can be used to solve the first;

where algorithm $F$ computes the reduction $f$, and algorithm $A_2$ solves for $L_2$.
Chapter 34. NP-Completeness

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;
- algorithms solving the second problem can be used to solve the first;

where algorithm $F$ computes the reduction $f$, and algorithm $A_2$ solves for $L_2$

So the combined algorithm (gray-color box) solves for $L_1$. 
Formally,
Formally,

A polynomial-time reduction from $L_1$ to $L_2$, denoted as $L_1 \leq_p L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$,
Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from $L_1$ to $L_2$, denoted as $L_1 \leq_p L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that
Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from $L_1$ to $L_2$, denoted as $L_1 \leq_p L_2$,
is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,
Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from $L_1$ to $L_2$, denoted as $L_1 \leq_p L_2$,
is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$
Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from \( L_1 \) to \( L_2 \), denoted as \( L_1 \leq_p L_2 \), is some mapping function \( f : \{0, 1\}^* \to \{0, 1\}^* \), such that for any \( x \in \{0, 1\}^* \),

\[
x \in L_1 \iff f(x) \in L_2
\]

where \( f \) can be computed in time \( O(|x|^c) \) for some fixed \( c > 0 \).
Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from $L_1$ to $L_2$, denoted as $L_1 \leq_p L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

where $f$ can be computed in time $O(|x|^c)$ for some fixed $c > 0$.

For example, $L_{IS} \leq_p L_V$. 
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$. We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time.

Total time is the sum of time for $F$ and time for $A_2$:

\[O(|x|^c) + O(|f(x)|^d)\]

Now, what is the length of $f(x)$?

Because $F$ runs in time $O(|x|^c)$, the number of bits outputted by $F$ is $O(|x|^c)$.

So

\[O(|x|^c) + O(|f(x)|^d) = O(|x|^c + O((|x|^c)^d)) = O(|x|^{cd})\]
Chapter 34. NP-Completeness

**Theorem**: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof**: Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$

We need to show that the gray box runs in polynomial time
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$.

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.
Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$.

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.

\[
O(|x|^c) + O(|f(x)|^d)
\]
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$.

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.

$$O(|x|^c) + O(|f(x)|^d) \text{ \ now, what is the length of } f(x)?$$
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$.

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.

$$O(|x|^c) + O(|f(x)|^d)$$

now, what is the length of $f(x)$?

Because $F$ runs in time $O(|x|^c)$,
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathbb{P}$, then $L_1 \in \mathbb{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$.

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.

$$O(|x|^c) + O(|f(x)|^d)$$

now, what is the length of $f(x)$?

Because $F$ runs in time $O(|x|^c)$, the number of bits outputted by $F$ is $O(|x|^c)$. 
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$.

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.

$$O(|x|^c) + O(|f(x)|^d)$$

Now, what is the length of $f(x)$?

Because $F$ runs in time $O(|x|^c)$, the number of bits outputted by $F$ is $O(|x|^c)$. So

$$O(|x|^c) + O(|f(x)|^d)$$
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_P L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.

$$O(|x|^c) + O(|f(x)|^d)$$

now, what is the length of $f(x)$?

Because $F$ runs in time $O(|x|^c)$, the number of bits outputted by $F$ is $O(|x|^c)$.

So

$$O(|x|^c) + O(|f(x)|^d) = O(|x|^c + O((|x|^c)^d))$$
Chapter 34. NP-Completeness

**Theorem:** Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

**Proof:** Assume algorithm $F$ computes $f$, and algorithm $A_2$ solves for $L_2$.

We need to show that the gray box runs in polynomial time if both $F$ and $A_2$ runs in polynomial time:

Total time is the sum of time for $F$ and time for $A_2$.

$$O(|x|^c) + O(|f(x)|^d)$$

now, what is the length of $f(x)$?

Because $F$ runs in time $O(|x|^c)$, the number of bits outputted by $F$ is $O(|x|^c)$. So

$$O(|x|^c) + O(|f(x)|^d) = O(|x|^c + O((|x|^c)^d)) = O(|x|^{cd})$$
Chapter 34. NP-Completeness

**Theorem**: Polynomial-time reductions compose (are transitive). That is, if \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

Proof. Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \). For every \( x \in \{0,1\}^* \), \( x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3 \). That is, \( x \in L_1 \iff h(f(x)) \in L_3 \). So composite function \( h \circ f \) realizes reduction \( L_1 \leq_p L_3 \).

But we need to show the reduction is \( \leq_p \), i.e., a polynomial time reduction. Assume that algorithm \( F \) computes \( f \): \( F(x) = f(x) \) in time \( O(|x|^c) \) and algorithm \( H \) computes \( h \): \( H(y) = h(y) \) in time \( O(|y|^d) \). Let \( y = f(x) \), the total time for computing \( (h \circ f) = \text{time of } F \text{ and time of } H = O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O(|x|^cd) = O(|x|^c + |x|^cd) \). So \( L_1 \leq_p L_3 \).
Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$. 

Proof. Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

For every $x \in \{0, 1\}^*$, $x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$.

That is $x \in L_1 \iff h(f(x)) \in L_3$.

So composite function $(h \circ f)$ realizes reduction $L_1 \leq_p L_3$.

But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm $F$ computes $f$: $F(x) = f(x)$ in time $O(|x|^c)$ and algorithm $H$ computes $h$: $H(y) = h(y)$ in time $O(|y|^d)$.

Let $y = f(x)$, the total time for computing $(h \circ f) = \text{time of } F \text{ and time of } H = O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O(|x|^cd) = O(|x|^c + |x|^cd)$. 

So $L_1 \leq_p L_3$. 

Chapter 34. NP-Completeness
Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$. 

**Proof.** Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

For every $x \in \{0, 1\}^*$, $x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$. That is $x \in L_1 \iff h(f(x)) \in L_3$.

So composite function $(h \circ f)$ realizes reduction $L_1 \leq_p L_3$.

But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm $F$ computes $f$: $F(x) = f(x)$ in time $O(|x|^c)$ and algorithm $H$ computes $h$: $H(y) = h(y)$ in time $O(|y|^d)$.

Let $y = f(x)$, the total time for computing $(h \circ f) = O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O(|x|^c d) = O(|x|^c) + O(|x|^c d)$.

So $L_1 \leq_p L_3$. 

Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is, if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof.** Assume functions $f$ for $L_1 \leq_p L_2$;

...
Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is if \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

**Proof.** Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \).
Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is if \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

**Proof.** Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \). for every \( x \in \{0, 1\}^* \),

\[
x \in L_1
\]
Theorem: Polynomial-time reductions compose (are transitive). That is if \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

Proof. Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \).

for every \( x \in \{0, 1\}^* \),

\[
x \in L_1 \iff f(x) \in L_2
\]
Theorem: Polynomial-time reductions compose (are transitive). That is if \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

Proof. Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \). For every \( x \in \{0, 1\}^* \),

\[
x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3
\]
Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof.** Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.
for every $x \in \{0, 1\}^*$,

\[ x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3 \]

That is
Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is, if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof.** Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

for every $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1$.
Chapter 34. NP-Completeness

**Theorem**: Polynomial-time reductions compose (are transitive). That is if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof**. Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

for every $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$
Theorem: Polynomial-time reductions compose (are transitive). That is if \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

Proof. Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \).

for every \( x \in \{0, 1\}^* \),

\[
x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3
\]

That is \( x \in L_1 \iff h(f(x)) \in L_3 \)

So composite function \( (h \circ f) \) realizes reduction \( L_1 \leq L_3 \).
Chapter 34. NP-Completeness

**Theorem**: Polynomial-time reductions compose (are transitive). That is

If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof**. Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

for every $x \in \{0, 1\}^*$,

$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.

But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.
**Theorem:** Polynomial-time reductions compose (are transitive). That is if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof.** Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

For every $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.

But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm $F$ computes $f$: $F(x) = f(x)$ in time $O(|x|^c)$
Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is, if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

for every $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.

But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm $F$ computes $f$: $F(x) = f(x)$ in time $O(|x|^c)$
and algorithm $H$ computes $h$: $H(y) = h(y)$ in time $O(|y|^d)$
Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is

If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof.** Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$.

for every $x \in \{0, 1\}^*$,

$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.

But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm $F$ computes $f$: $F(x) = f(x)$ in time $O(|x|^c)$

and algorithm $H$ computes $h$: $H(y) = h(y)$ in time $O(|y|^d)$

Let $y = f(x)$, the total time for computing $(h \circ f) = \text{time of } F \text{ and time of } H$

$= O(|x|^c) + O(|f(x)|^d)$
Chapter 34. NP-Completeness

**Theorem:** Polynomial-time reductions compose (are transitive). That is, if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

**Proof.** Assume functions $f$ for $L_1 \leq_p L_2$; function $h$ for $L_2 \leq_p L_3$. For every $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.

But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm $F$ computes $f$: $F(x) = f(x)$ in time $O(|x|^c)$

and algorithm $H$ computes $h$: $H(y) = h(y)$ in time $O(|y|^d)$

Let $y = f(x)$, the total time for computing $(h \circ f) = $ time of $F$ and time of $H$

$$= O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O((|x|^c)^d)$$
**Theorem:** Polynomial-time reductions compose (are transitive). That is

If \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

**Proof.** Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \).

for every \( x \in \{0, 1\}^* \),

\[
x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3
\]

That is \( x \in L_1 \iff h(f(x)) \in L_3 \)

So composite function \((h \circ f)\) realizes reduction \( L_1 \leq L_3 \).

But we need to show the reduction is \( \leq_p \), i.e., a polynomial time reduction.

Assume that algorithm \( F \) computes \( f \): \( F(x) = f(x) \) in time \( O(|x|^c) \)
and algorithm \( H \) computes \( h \): \( H(y) = h(y) \) in time \( O(|y|^d) \)

Let \( y = f(x) \), the total time for computing \((h \circ f) = \) time of \( F \) and time of \( H \)

\[
= O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O((|x|^c)^d) = O(|x|^c) + O(|x|^{cd})
\]
Chapter 34. NP-Completeness

**Theorem**: Polynomial-time reductions compose (are transitive). That is if \( L_1 \leq_p L_2 \) and \( L_2 \leq_p L_3 \), then \( L_1 \leq_p L_3 \).

**Proof.** Assume functions \( f \) for \( L_1 \leq_p L_2 \); function \( h \) for \( L_2 \leq_p L_3 \).

for every \( x \in \{0, 1\}^* \),

\[
\begin{align*}
x \in L_1 & \iff f(x) \in L_2 & \iff h(f(x)) \in L_3
\end{align*}
\]

That is \( x \in L_1 \iff h(f(x)) \in L_3 \)

So composite function \((h \circ f)\) realizes reduction \( L_1 \leq L_3 \).
But we need to show the reduction is \( \leq_p \), i.e., a polynomial time reduction.

Assume that algorithm \( F \) computes \( f \): \( F(x) = f(x) \) in time \( O(|x|^c) \)
and algorithm \( H \) computes \( h \): \( H(y) = h(y) \) in time \( O(|y|^d) \)

Let \( y = f(x) \), the total time for computing \((h \circ f) = \) time of \( F \) and time of \( H \)

\[
= O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O((|x|^c)^d) = O(|x|^c) + O(|x|^{cd})
\]

So \( L_1 \leq_p L_3 \).
Some conclusions:
Some conclusions:

• Using $\leq_p$, languages in $\mathcal{NP}$ can be ordered partially;
Chapter 34. NP-Completeness

Some conclusions:

- Using $\leq_p$, languages in $NP$ can be ordered partially;
- If those languages at the end of a $\leq_p$ chain have polynomial-time algorithms, so does every language on the chain.
Some conclusions:

- Using $\leq_p$, languages in $\mathcal{NP}$ can be ordered partially;
- If those languages at the end of a $\leq_p$ chain have polynomial-time algorithms, so does every language on the chain.
- Informally, those at the end of a $\leq_p$ chain are called NP-hard.
Chapter 34. NP-Completeness

**Definition 1:** $L$ is **NP-hard** if for every language $L' \in \text{NP}$, $L' \leq_p L$.

**Definition 2:** $L$ is **NP-complete** if (1) $L$ is NP-hard and (2) $L \in \text{NP}$.

**Properties of NP-hard problems**

- If $L$ is NP-hard and $L \in \text{P}$, then $\text{P} = \text{NP}$.

**Proof?**

- If $L$ is NP-hard and $L \leq_p L'$, then $L'$ is NP-hard.

**Proof?**

**How to prove a language is NP-hard?**
Definition 1: \( L \) is **NP-hard** if for every language \( L' \in \mathcal{NP} \), \( L' \leq_p L \).

Definition 2: \( L \) is **NP-complete** if (1) \( L \) is NP-hard and (2) \( L \in \mathcal{NP} \).
Chapter 34. NP-Completeness

Definition 1: \( L \) is **NP-hard** if for every language \( L' \in \mathcal{NP} \), \( L' \leq_p L \).

Definition 2: \( L \) is **NP-complete**
Chapter 34. NP-Completeness

Definition 1: $L$ is **NP-hard** if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 2: $L$ is **NP-complete** if (1) $L$ is NP-hard
Definition 1: \( L \) is **NP-hard** if for every language \( L' \in \mathcal{NP} \), \( L' \leq_p L \).

Definition 2: \( L \) is **NP-complete** if (1) \( L \) is NP-hard and (2) \( L \in \mathcal{NP} \).
Definition 1: $L$ is **NP-hard** if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 2: $L$ is **NP-complete** if (1) $L$ is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems
Chapter 34. NP-Completeness

**Definition 1:** $L$ is **NP-hard** if for every language $L' \in \mathcal{NP}$, $L' \leq^p L$.

**Definition 2:** $L$ is **NP-complete** if (1) $L$ is NP-hard and (2) $L \in \mathcal{NP}$.

**Properties of NP-hard problems**

- If $L$ is NP-hard

Chapter 34. NP-Completeness

**Definition 1**: $L$ is **NP-hard** if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

**Definition 2**: $L$ is **NP-complete** if (1) $L$ is NP-hard and (2) $L \in \mathcal{NP}$.

**Properties of NP-hard problems**

- If $L$ is NP-hard and $L \in \mathcal{P}$,
Chapter 34. NP-Completeness

**Definition 1:** $L$ is **NP-hard** if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

**Definition 2:** $L$ is **NP-complete** if (1) $L$ is NP-hard and (2) $L \in \mathcal{NP}$.

**Properties of NP-hard problems**

- If $L$ is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
Definition 1: $L$ is **NP-hard** if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 2: $L$ is **NP-complete** if (1) $L$ is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If $L$ is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  
  Proof?
Definition 1: $L$ is **NP-hard** if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 2: $L$ is **NP-complete** if (1) $L$ is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If $L$ is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  
  Proof?

- If $L$ is NP-hard
**Definition 1:** \( L \) is **NP-hard** if for every language \( L' \in \mathcal{NP} \), \( L' \leq_p L \).

**Definition 2:** \( L \) is **NP-complete** if (1) \( L \) is NP-hard and (2) \( L \in \mathcal{NP} \).

**Properties of NP-hard problems**

- If \( L \) is NP-hard and \( L \in \mathcal{P} \), then \( \mathcal{P} = \mathcal{NP} \).
  - Proof?

- If \( L \) is NP-hard and \( L \leq_p L' \),
Definition 1: \( L \) is **NP-hard** if for every language \( L' \in \mathcal{NP} \), \( L' \leq_p L \).

Definition 2: \( L \) is **NP-complete** if (1) \( L \) is NP-hard and (2) \( L \in \mathcal{NP} \).

**Properties of NP-hard problems**

- If \( L \) is NP-hard and \( L \in \mathcal{P} \), then \( \mathcal{P} = \mathcal{NP} \).
  
  Proof?

- If \( L \) is NP-hard and \( L \leq_p L' \), then \( L' \) is NP-hard.
Chapter 34. NP-Completeness

**Definition 1:** \( L \) is **NP-hard** if for every language \( L' \in \mathcal{NP} \), \( L' \leq_p L \).

**Definition 2:** \( L \) is **NP-complete** if (1) \( L \) is NP-hard and (2) \( L \in \mathcal{NP} \).

**Properties of NP-hard problems**

- If \( L \) is NP-hard and \( L \in \mathcal{P} \), then \( \mathcal{P} = \mathcal{NP} \).
  Proof?

- If \( L \) is NP-hard and \( L \leq_p L' \), then \( L' \) is NP-hard.
  Proof?
Chapter 34. NP-Completeness

**Definition 1:** \( L \) is **NP-hard** if for every language \( L' \in \mathcal{NP} \), \( L' \leq_p L \).

**Definition 2:** \( L \) is **NP-complete** if (1) \( L \) is NP-hard and (2) \( L \in \mathcal{NP} \).

**Properties of NP-hard problems**

- If \( L \) is NP-hard and \( L \in \mathcal{P} \), then \( \mathcal{P} = \mathcal{NP} \). 
  Proof?

- If \( L \) is NP-hard and \( L \leq_p L' \), then \( L' \) is NP-hard. 
  Proof?

**How to prove a language is NP-hard?**
Chapter 34. NP-Completeness

4. NP-Completeness Proofs

To prove a language \( L \) is NP-complete, we need to show it is NP-hard. That is, we need to show for every language \( L' \in \text{NP} \), \( L' \leq_p L \).

 Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to \( L \).

 Instead, formulate a generic language that represents all languages in NP and prove that every language in NP can be reduced to the generic language in polynomial time.

 To obtain such a generic language, we need to consider the definition of languages in NP.
Chapter 34. NP-Completeness

4. NP-Completeness Proofs

To prove a language $L$ is NP-complete,
4. NP-Completeness Proofs

To prove a language $L$ is NP-complete, we need to show it is NP-hard.
4. NP-Completeness Proofs

To prove a language $L$ is NP-complete, we need to show it is NP-hard. That is, we need to show
4. NP-Completeness Proofs

To prove a language $L$ is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP}, \ L' \leq_p L$$
4. NP-Completeness Proofs

To prove a language $L$ is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP}, \ L' \leq_p L$$

• Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to $L$. 
Chapter 34. NP-Completeness

4. NP-Completeness Proofs

To prove a language $L$ is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP}, L' \leq_p L$$

- Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to $L$.
- Instead, formulate a generic language that represents all languages in NP
4. NP-Completeness Proofs

To prove a language \( L \) is NP-complete, we need to show it is NP-hard. That is, we need to show

\[
\text{for every language } L' \in \mathcal{NP}, L' \leq_p L
\]

- Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to \( L \).
- Instead, formulate a generic language that represents all languages in NP and prove that every language in \( \mathcal{NP} \) can be reduced to the generic language in polynomial time.
4. NP-Completeness Proofs

To prove a language $L$ is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\quad \text{for every language } L' \in \mathcal{NP}, \ L' \leq_p L$$

- Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to $L$.

- Instead, formulate a generic language that represents all languages in NP and prove that every language in $\mathcal{NP}$ can be reduced to the generic language in polynomial time.

- To obtain such a generic language, we need to consider the definition of languages in NP.
Recall the definition of languages in $\mathcal{NP}$:

$L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$, $x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$ and $A_L$ runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction $x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \uparrow x \in L \iff f(x) \in L_{tbd}$ where $L_{tbd}$ is a language to be defined.

Can we identify $L_{tbd}$ and $f$?
Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm $A_L$, such that for every $x \in \{0, 1\}^*$, $x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$ and $A_L$ runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction $x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \leftrightarrow f(x) \in L_{tbd}$ where $L_{tbd}$ is a language to be defined.

Can we identify $L_{tbd}$ and $f$?
Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L$$
Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.
Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c,$$
Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.
Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

The “iff” relationship looks a little like the relationship in a reduction
Recall the definition of languages in \( \mathcal{NP} \):

Let \( L \subseteq \{0, 1\}^* \) be any language in the class \( \mathcal{NP} \). Then there is a deterministic algorithm \( A_L \), and a constant \( c > 0 \), such that, for every \( x \in \{0, 1\}^* \),

\[
x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1
\]

and \( A_L \) runs in polynomial time.

The “iff” relationship looks a little like the relationship in a reduction

\[
x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1
\]
Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

The “iff” relationship looks a little like the relationship in a reduction

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

$\uparrow$
Recall the definition of languages in \( \mathcal{NP} \) :

Let \( L \subseteq \{0,1\}^* \) be any language in the class \( \mathcal{NP} \). Then there is a deterministic algorithm \( A_L \), and a constant \( c > 0 \), such that, for every \( x \in \{0,1\}^* \),

\[
x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1
\]

and \( A_L \) runs in polynomial time.

The “iff” relationship looks a little like the relationship in a reduction

\[
x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1
\]

\[
\uparrow
\]

\[
x \in L \iff f(x) \in L_{tbd}
\]
Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

$$\uparrow$$

$$x \in L \iff f(x) \in L_{tbd}$$

where $L_{tbd}$ is a language to be defined.
Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0, 1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a deterministic algorithm $A_L$, and a constant $c > 0$, such that, for every $x \in \{0, 1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and $A_L$ runs in polynomial time.

The “iff” relationship looks a little like the relationship in a reduction

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

\[\uparrow\]

$$x \in L \iff f(x) \in L_{tbd}$$

where $L_{tbd}$ is a language to be defined.

Can we identify $L_{tbd}$ and $f$?
Again we examine

\[ x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \]  

(1)
Again we examine
\[ x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \]  
(1)

- \( A_L \) is a deterministic algorithm can be implemented with a boolean circuit \( B_L \) with two sets of input gates \( x = x_1x_2 \ldots x_n \) and \( y = y_1y_2 \ldots y_m \) such that
\[ A_L(x, y) = 1 \text{ if and only if } B_L(x, y) = 1 \]  
(2)
Again we examine

\[ x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \]  

(1)

- \( A_L \) is a deterministic algorithm can be implemented with a boolean circuit \( B_L \) with two sets of input gates \( x = x_1 x_2 \ldots x_n \) and \( y = y_1 y_2 \ldots y_m \) such that

\[ A_L(x, y) = 1 \text{ if and only if } B_L(x, y) = 1 \]  

(2)
Chapter 34. NP-Completeness

Again we examine

\[ x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \quad (1) \]

- \( A_L \) is a deterministic algorithm can be implemented with a boolean circuit \( B_L \) with two sets of input gates \( x = x_1x_2 \ldots x_n \) and \( y = y_1y_2 \ldots y_m \) such that

\[ A_L(x, y) = 1 \text{ if and only if } B_L(x, y) = 1 \quad (2) \]

Because \( x \) is given, circuit \( B_L \) can be made into circuit \( C^x_L \) such that

\[ B_L(x, y) = 1 \text{ if and only if } C^x_L(y) = 1 \quad (3) \]
Chapter 34. NP-Completeness

Again we examine

\[ x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \]  

(1)

- \( A_L \) is a deterministic algorithm can be implemented with a boolean circuit \( B_L \) with two sets of input gates \( x = x_1 x_2 \ldots x_n \) and \( y = y_1 y_2 \ldots y_m \) such that

\[ A_L(x, y) = 1 \text{ if and only if } B_L(x, y) = 1 \]  

(2)

- Because \( x \) is given, circuit \( B_L \) can be made into circuit \( C^x_L \) such that

\[ B_L(x, y) = 1 \text{ if and only if } C^x_L(y) = 1 \]  

(3)
Again we examine
\[ x \in L \iff \exists y, \|y\| \leq |x|^c, A_L(x, y) = 1 \]  \hspace{1cm} (1)

- \( A_L \) is a deterministic algorithm can be implemented with a boolean circuit \( B_L \) with two sets of input gates \( x = x_1x_2 \ldots x_n \) and \( y = y_1y_2 \ldots y_m \) such that
\[
A_L(x, y) = 1 \text{ if and only if } B_L(x, y) = 1
\]  \hspace{1cm} (2)

Because \( x \) is given, circuit \( B_L \) can be made into circuit \( C^x_L \) such that
\[
B_L(x, y) = 1 \text{ if and only if } C^x_L(y) = 1
\]  \hspace{1cm} (3)

- From (1), (2), and (3), we have
\[
x \in L \iff \exists y C^x_L(y) = 1
\]  \hspace{1cm} (4)
Now we have

\[ x \in L \iff \exists y \ C^x_L(y) = 1 \quad (5) \]

Define: a boolean circuit \( C \) is satisfiable if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

e.g., \( C(x_1, x_2) = (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2) \) is satisfiable;
but \( D(x_1, x_2) = (x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) is not!

Define the following language:

\[ \text{CSAT} = \{ C : \text{circuit } C \text{ is satisfiable} \} \]

From (4), we have

\[ x \in L \iff C^x_L \in \text{CSAT} \quad (6) \]

It remains to be shown that reducing algorithm \( A_L \) to circuit \( B_L \) is valid; and that the reduction can be done in polynomial time.
Chapter 34. NP-Completeness

Now we have

\[ x \in L \iff \exists y C^x_L(y) = 1 \]  \hspace{1cm} (5)

- Define: a boolean circuit \( C \) is **satisfiable** if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

It remains to be shown that reducing algorithm \( A_L \) to circuit \( B_L \) is valid; and that the reduction can be done in polynomial time.
Chapter 34. NP-Completeness

Now we have

\[ x \in L \iff \exists y C^x_L(y) = 1 \]  \hspace{1cm} (5)

- Define: a boolean circuit \( C \) is **satisfiable** if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

  e.g., \( C(x_1, x_2) = (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2) \) is satisfiable;
Now we have

\[ x \in L \iff \exists y C^x_L (y) = 1 \tag{5} \]

* Define: a boolean circuit \( C \) is **satisfiable** if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

  e.g., \( C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \) is satisfiable;
  but \( D(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \) is not!
Now we have

\[ x \in L \iff \exists y C^x_L(y) = 1 \]  \hspace{1cm} (5)

- Define: a boolean circuit \( C \) is **satisfiable** if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

  e.g., \( C(x_1, x_2) = (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2) \) is satisfiable;
  but \( D(x_1, x_2) = (x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) is not!

- Define the following language:

  \[ CSAT = \{ C : \text{circuit } C \text{ is satisfiable } \} \]
Chapter 34. NP-Completeness

Now we have

\[ x \in L \iff \exists y C^x_L(y) = 1 \] (5)

- Define: a boolean circuit \( C \) is **satisfiable** if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

  e.g., \( C(x_1, x_2) = (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2) \) is satisfiable;
  but \( D(x_1, x_2) = (x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) is not!

- Define the following language:

  \[ CSAT = \{ C : \text{circuit } C \text{ is satisfiable } \} \]

- From (4), we have

  \[ x \in L \iff C^x_L \in CSAT \] (6)
Now we have

\[ x \in L \iff \exists y C^x_L(y) = 1 \]  

(5)

• Define: a boolean circuit \( C \) is **satisfiable** if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

  e.g., \( C(x_1, x_2) = (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2) \) is satisfiable;
  but \( D(x_1, x_2) = (x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) is not!

• Define the following language:

  \[ CSAT = \{ C : \text{circuit } C \text{ is satisfiable } \} \]

• From (4), we have

  \[ x \in L \iff C^x_L \in CSAT \]  

(6)

It remains to be shown
Chapter 34. NP-Completeness

Now we have

\[ x \in L \iff \exists y C^x_L(y) = 1 \quad (5) \]

• Define: a boolean circuit \( C \) is \textbf{satisfiable} if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

  e.g., \( C(x_1, x_2) = (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2) \) is satisfiable;
  but \( D(x_1, x_2) = (x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) is not!

• Define the following language:

\[ CSAT = \{ C : \text{circuit } C \text{ is satisfiable} \} \]

• From (4), we have

\[ x \in L \iff C^x_L \in CSAT \quad (6) \]

It remains to be shown

• that reducing \textbf{algorithm} \( A_L \) to \textbf{circuit} \( B_L \) is valid;
Chapter 34. NP-Completeness

Now we have
\[ x \in L \iff \exists y \ C_L^x(y) = 1 \]  
(5)

- Define: a boolean circuit \( C \) is **satisfiable** if there exists at least one set of values \( y \) to its input gates such that \( C(y) = 1 \).

  e.g., \( C(x_1, x_2) = (\neg x_1 \lor \neg x_2) \land (x_1 \lor x_2) \) is satisfiable;
  but \( D(x_1, x_2) = (x_1 \lor x_2) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2) \) is not!

- Define the following language:

  \[ CSAT = \{ C : \text{circuit } C \text{ is satisfiable } \} \]

- From (4), we have

  \[ x \in L \iff C_L^x \in CSAT \]  
(6)

It remains to be shown

- that reducing **algorithm** \( A_L \) to **circuit** \( B_L \) is valid; and

- that the reduction can be done in **polynomial time**.
Chapter 34. NP-Completeness

Unfold deterministic polynomial-time algorithm $A(x, y)$ with input $\langle x, y \rangle$
Chapter 34. NP-Completeness

Unfold deterministic polynomial-time algorithm $A(x, y)$ with input $\langle x, y \rangle$
Chapter 34. NP-Completeness

The algorithm is physically implemented with a boolean circuit.
Chapter 34. NP-Completeness

The algorithm is physically implemented with a boolean circuit.
Chapter 34. NP-Completeness

The algorithm is physically implemented with a boolean circuit.
Chapter 34. NP-Completeness

The algorithm is physically implemented with a boolean circuit

And the circuit can be built from the algorithm in polynomial time.
The above discussion shows that $L_{CSAT}$ is NP-hard.
Chapter 34. NP-Completeness

The above discussion shows that $L_{CSAT}$ is NP-hard.

**Theorem**: Language $CSAT$ is NP-complete.
Chapter 34. NP-Completeness

The above discussion shows that $L_{CSAT}$ is NP-hard.

**Theorem:** Language $CSAT$ is NP-complete.

**Proof:** It suffices to show the $CSAT$ is in NP. (Can you prove this?)
The above discussion shows that $L_{CSAT}$ is NP-hard.

**Theorem**: Language $CSAT$ is NP-complete.

**Proof**: It suffices to show the $CSAT$ is in NP. *(Can you prove this?)*

Actually, the following language SAT was first proved to be NP-complete [Cook’71]:

$$SAT = \{ \phi : \text{CNF boolean formula } \phi \text{ is satisfiable} \}$$
The above discussion shows that $L_{CSAT}$ is NP-hard.

**Theorem**: Language $CSAT$ is NP-complete.

**Proof**: It suffices to show the $CSAT$ is in NP. *(Can you prove this?)*

Actually, the following language SAT was first proved to be NP-complete [Cook’71] https://www.cs.toronto.edu/~sacook/homepage/1971.pdf

$$SAT = \{ \phi : \text{CNF boolean formula } \phi \text{ is satisfiable}\}$$

**Cook’s Theorem**: SAT is NP-complete.
The above discussion shows that $L_{CSAT}$ is NP-hard.

**Theorem:** Language $CSAT$ is NP-complete.

**Proof:** It suffices to show the $CSAT$ is in NP. *(Can you prove this?)*

Actually, the following language SAT was first proved to be NP-complete [Cook'71]

$$SAT = \{ \phi : \text{CNF boolean formula } \phi \text{ is satisfiable} \}$$

**Cook’s Theorem:** SAT is NP-complete.

Cook’s reduction: characterizing a polynomial-time computation on nondeterministic Turing machine with a boolean formula,
Chapter 34. NP-Completeness

The above discussion shows that $L_{CSAT}$ is NP-hard.

**Theorem:** Language $CSAT$ is NP-complete.

**Proof:** It suffices to show the $CSAT$ is in NP. (Can you prove this?)

Actually, the following language SAT was first proved to be NP-complete [Cook’71] https://www.cs.toronto.edu/~sacook/homepage/1971.pdf

$$SAT = \{ \phi : \text{CNF boolean formula } \phi \text{ is satisfiable} \}$$

**Cook’s Theorem:** SAT is NP-complete.

Cook’s reduction: characterizing a polynomial-time computation on nondeterministic Turing machine with a boolean formula, such that a nondeterministic path leading to the accept state corresponds to an assignment to the variables making the the formula TRUE.
Chapter 34. NP-Completeness

It is very easy to convert a boolean formula to a boolean circuit. So
Chapter 34. NP-Completeness

It is very easy to convert a boolean formula to a boolean circuit. So

**Theorem**: $SAT \leq_p CSAT$. 
Chapter 34. NP-Completeness

It is very easy to convert a boolean formula to a boolean circuit. So

**Theorem:** $SAT \leq_p CSAT$.

On the other hand,
Chapter 34. NP-Completeness

It is very easy to convert a boolean formula to a boolean circuit. So

**Theorem**: $SAT \leq_p CSAT$.

On the other hand,

**Theorem**: $CSAT \leq_p SAT$. 

how to convert a circuit to a boolean formula (from network to tree)?
as simple replicating gates may blow-up the size of formula to exponential!
Chapter 34. NP-Completeness

It is very easy to convert a boolean formula to a boolean circuit. So

**Theorem**: $SAT \leq_p CSAT$.

On the other hand,

**Theorem**: $CSAT \leq_p SAT$.

how to convert a circuit to a boolean formula (from network to tree)?
Chapter 34. NP-Completeness

It is very easy to convert a boolean formula to a boolean circuit. So

**Theorem:** $SAT \leq_p CSAT$.

On the other hand,

**Theorem:** $CSAT \leq_p SAT$.

**how to convert a circuit to a boolean formula** (from network to tree)?
simply replicating gates may blow-up the size of formula to exponential!
Chapter 34. NP-Completeness

**Theorem:** $CSAT \leq_p SAT$. 

The diagram shows a circuit with variables $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9,$ and $x_{10}$. The circuit involves logical gates connecting these variables.
Chapter 34. NP-Completeness

**Theorem:** \( CSAT \leq_p SAT \).

is satisfiable if and only if formula \( \phi \) is satisfiable:
Chapter 34. NP-Completeness

**Theorem:** $CSAT \leq_p SAT$.

is satisfiable if and only if formula $\phi$ is satisfiable:

\[
\phi = x_{10} \land (x_4 \leftrightarrow \neg x_3) \\
\land (x_5 \leftrightarrow (x_1 \lor x_2)) \\
\land (x_6 \leftrightarrow \neg x_4) \\
\land (x_7 \leftrightarrow (x_1 \land x_2 \land x_4)) \\
\land (x_8 \leftrightarrow (x_5 \lor x_6)) \\
\land (x_9 \leftrightarrow (x_6 \lor x_7)) \\
\land (x_{10} \leftrightarrow (x_7 \land x_8 \land x_9)).
\]
Chapter 34. NP-Completeness

**Theorem:** $CSAT \leq_p SAT$.

$\phi$ can be transformed to an equivalent CNF formula.
Chapter 34. NP-Completeness

Landscape of NP problems and beyond
Chapter 34. NP-Completeness

Landscape of NP problems and beyond

NP-complete
- Hamilton cycle
- Steiner tree
- Graph 3-coloring
- Satisfiability
- Maximum clique

NP-hard
- Matrix permanent
- Halting problem
- ...

NP
- Factoring
- Graph isomorphism
- ...

P
- Graph connectivity
- Primality testing
- Matrix determinant
- Linear programming
- ...

...
Chapter 34. NP-Completeness

Many problems/languages have been proved NP-complete (Karp70s)
Examples of reduction techniques
Examples of reduction techniques

Example 1: $SAT \leq_p 3SAT$
Examples of reduction techniques

Example 1: SAT $\leq_p$ 3SAT

($z$)
Examples of reduction techniques

Example 1: SAT \(\leq_p\) 3SAT

\[
(z) \mapsto (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)
\]
Examples of reduction techniques

Example 1: \( SAT \leq_p 3SAT \)

\[
(z) \implies (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)
\]

\[
(y, z)
\]
Examples of reduction techniques

Example 1: SAT $\leq_p$ 3SAT

\[
(z) \mapsto (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)
\]

\[
(y, z) \mapsto (y, z, x_1) \land (y, z, \neg x_1)
\]
Examples of reduction techniques

Example 1: SAT $\leq_p$ 3SAT

\[
(z) \implies (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)
\]

\[
(y, z) \implies (y, z, x_1) \land (y, z, \neg x_1)
\]

\[
(x, y, z)
\]
Examples of reduction techniques

Example 1: SAT $\leq_p$ 3SAT

$$(z) \rightarrow (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)$$

$$(y, z) \rightarrow (y, z, x_1) \land (y, z, \neg x_1)$$

$$(x, y, z) \rightarrow (x, y, z)$$
Chapter 34. NP-Completeness

Examples of reduction techniques

Example 1: $\text{SAT} \leq_p \text{3SAT}$

- $(z) \rightarrow (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)$
- $(y, z) \rightarrow (y, z, x_1) \land (y, z, \neg x_1)$
- $(x, y, z) \rightarrow (x, y, z)$
- $(y, z, u, v)$
Chapter 34. NP-Completeness

Examples of reduction techniques

Example 1: SAT $\leq_p$ 3SAT

$(z) \implies (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)$

$(y, z) \implies (y, z, x_1) \land (y, z, \neg x_1)$

$(x, y, z) \implies (x, y, z)$

$(y, z, u, v) \implies (y, z, x_1) \land (\neg x_1, u, v)$
Examples of reduction techniques

Example 1: SAT $\leq^p$ 3SAT

$(z) \Rightarrow (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)$

$(y, z) \Rightarrow (y, z, x_1) \land (y, z, \neg x_1)$

$(x, y, z) \Rightarrow (x, y, z)$

$(y, z, u, v) \Rightarrow (y, z, x_1) \land (\neg x_1, u, v)$

$(y, z, u, v, w)$
Chapter 34. NP-Completeness

Examples of reduction techniques

Example 1: \( SAT \leq_p 3SAT \)

\[
(z) \mapsto (z, x_1, x_2) \land (z, x_1, \neg x_2) \land (z, \neg x_1, x_2) \land (z, \neg x_1, \neg x_2)
\]

\[
(y, z) \mapsto (y, z, x_1) \land (y, z, \neg x_1)
\]

\[
(x, y, z) \mapsto (x, y, z)
\]

\[
(y, z, u, v) \mapsto (y, z, x_1) \land (\neg x_1, u, v)
\]

\[
(y, z, u, v, w) \mapsto (y, z, x_1) \land (\neg x_1, u, x_2) \land (\neg x_2, v, w)
\]
Chapter 34. NP-Completeness

Example 2: 3SAT ≤ₚ IS

An assignment TRUE to one literal in each clause corresponds to an independent set in the transformed graph.
Example 2: 3SAT \leq_p IS
Example 2: $3\text{SAT} \leq_p \text{IS}$

\[(x_1 \lor x_2 \lor \overline{x_3}) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_2} \lor x_4)\]
Example 2: $3SAT \leq_p IS$

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

An assignment TRUE to one literal in each clause
Chapter 34. NP-Completeness

Example 2: $3\text{SAT} \leq_p \text{IS}$

An assignment TRUE to one literal in each clause corresponds to an independent set in the transformed graph.
Summary

Scope of the Final Exam
Scope of the Final Exam

- Dynamic programming (4 steps)
- Greedy algorithms (greedy choice property and proof)
- Depth-First-Search
  - algorithm, DFS search tree, time stamps
- Applications
  - topological sort
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree concept/properties of MST, generic, Kruskal’s, and Prim’s
- Shortest path (single source and all pairs) concept/properties of shortest path, greedy algorithms, relaxation technique
  - single source: Bellman-Ford’s, Dijkstra’s
  - all pairs: Floyd-Warshall
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree
  concept/properties of MST
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree
  - concept/properties of MST
  - generic, Kruskal’s and Prim’s
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree
  concept/properties of MST
generic, Kruskal’s and Prim’s

- Shortest path (single source and all pairs)
  concept/properties of shortest path, greedy algorithms, relaxation technique
  single source: Bellman-Ford’s, Dijkstra’s
  all pairs: Floyd-Warshall
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree
  - concept/properties of MST
    - generic, Kruskal’s and Prim’s
- Shortest path (single source and all pairs)
  - concept/properties of shortest path, greedy algorithms, relaxation technique
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree
  - concept/properties of MST
  - generic, Kruskal’s and Prim’s

- Shortest path (single source and all pairs)
  - concept/properties of shortest path, greedy algorithms, relaxation technique
  - single source: Bellman-Ford’s, Dijkstra’s
Summary

Scope of the Final Exam (cont’)

- Minimum spanning tree
  
  concept/properties of MST
  
  generic, Kruskal’s and Prim’s

- Shortest path (single source and all pairs)
  
  concept/properties of shortest path, greedy algorithms, relaxation technique
  
  single source: Bellman-Ford’s, Dijkstra’s
  
  all pairs: Floyd-Warshall
Summary

Scope of the Final Exam (cont')

NP-completeness theory
- definitions of NP class (certificate + verification)
- proof that a language is in NP
- reduction, polynomial-time reduction, properties
- definitions of NP-hard, NP-complete languages, properties
- NP-completeness proofs (simple, limited to previously known reductions)
Summary

Scope of the Final Exam (cont’)}

NP-completeness theory
- Definitions of NP class (certificate + verification)
- Proof that a language is in NP
- Reduction, polynomial-time reduction, properties
- Definitions of NP-hard, NP-complete languages, properties
- NP-completeness proofs (simple, limited to previously known reductions)
Scope of the Final Exam (cont’)

NP-completeness theory
- definitions of NP class (certificate + verification)
- proof that a language is in NP
- reduction, polynomial-time reduction, properties
- definitions of NP-hard, NP-complete languages, properties
- NP-completeness proofs (simple, limited to previously known reductions)
Summary

Scope of the Final Exam (cont’)

▶ NP-completeness theory
Summary

Scope of the Final Exam (cont’)

- NP-completeness theory
  - definitions of NP class (certificate + verification)
Summary

Scope of the Final Exam (cont’)

- NP-completeness theory
  - definitions of NP class (certificate + verification)
  - proof that a language is in NP
Summary

Scope of the Final Exam (cont’)

- NP-completeness theory
  - definitions of NP class (certificate + verification)
  - proof that a language is in NP
  - reduction, polynomial-time reduction, properties
Summary

Scope of the Final Exam (cont’)

- NP-completeness theory
  definitions of NP class (certificate + verification)
  proof that a language is in NP
  reduction, polynomial-time reduction, properties
  definitions of NP-hard, NP-complete languages, properties
Scope of the Final Exam (cont’)

- NP-completeness theory

  definitions of NP class (certificate + verification)

proof that a language is in NP

reduction, polynomial-time reduction, properties

definitions of NP-hard, NP-complete languages, properties

NP-completeness proofs (simple, limited to previously known reductions)