Lecture Note (Part 5)

CSCI 4470/6470 Algorithms, Fall 2023

Liming Cai
Department of Computer Science, UGA

November 30, 2023

Part 5. NP-completeness theory (Chapters 8 and 9)

Topics to be discussed:

- Some (actually a lot of) problems are difficult
- Decision vs search problems
- Polynomial-time verifiable problems
- NP-completeness theory

1. Intractable problems and exhaustive search

1. Intractable problems and exhaustive search

- Tractable problems - solvable in time $O(n^k)$, for constant k;

1. Intractable problems and exhaustive search

- Tractable problems - solvable in time $O(n^k)$, for constant k;

  Reachability: determine if $\exists$ path $s \rightsquigarrow t$ in a given graph

1. Intractable problems and exhaustive search

- Tractable problems - solvable in time $O(n^k)$, for constant k;

  Reachability: determine if $\exists$ path $s \rightsquigarrow t$ in a given graph

- Intractable problems – those (seemingly) without polynomial time algorithms

1. Intractable problems and exhaustive search

- Tractable problems - solvable in time $O(n^k)$, for constant k;

  Reachability: determine if $\exists$ path $s \rightsquigarrow t$ in a given graph

- Intractable problems – those (seemingly) without polynomial time algorithms

  Hamiltonian path:
  Input: a graph $G = (V, E)$,
  Output: "Yes" iff $\exists$ a Hamiltonian path in G

1. Intractable problems and exhaustive search

- Tractable problems - solvable in time $O(n^k)$, for constant k;

  Reachability: determine if $\exists$ path $s \rightsquigarrow t$ in a given graph

- Intractable problems – those (seemingly) without polynomial time algorithms

  Hamiltonian path:
  Input: a graph $G = (V, E)$,
  Output: "Yes" iff $\exists$ a Hamiltonian path in G

  A Hamiltonian path is a path connecting all vertices

1. Intractable problems and exhaustive search

Known algorithms for **Hamiltonian path** are mostly based on *exhaustive search*

1. Intractable problems and exhaustive search

Known algorithms for **Hamiltonian path** are mostly based on *exhaustive search*

- BFS-like: start from a vertex, check all its neighbors, repeat;

1. Intractable problems and exhaustive search

Known algorithms for **Hamiltonian path** are mostly based on *exhaustive search*

- BFS-like: start from a vertex, check all its neighbors, repeat;
- enumeration of all possible permutations of vertices (how?)

1. Intractable problems and exhaustive search

Known algorithms for **Hamiltonian path** are mostly based on *exhaustive search*

- BFS-like: start from a vertex, check all its neighbors, repeat;
- enumeration of all possible permutations of vertices (how?)

Of time complexity $\Omega(n^n)$ or $\Omega(n!)$.

1. Intractable problems and exhaustive search

Boolean formula satisfiability (SAT):

INPUT: a boolean formula $\phi(x_1, \ldots, x_n)$ of n variables;
OUTPUT: "Yes" if and only if ϕ is satisfiable.

1. Intractable problems and exhaustive search

Boolean formula satisfiability (SAT):

INPUT: a boolean formula $\phi(x_1, \ldots, x_n)$ of n variables;
OUTPUT: "Yes" if and only if ϕ is satisfiable.

- boolean formula: e.g.,

$$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

1. Intractable problems and exhaustive search

Boolean formula satisfiability (SAT):

INPUT: a boolean formula $\phi(x_1, \ldots, x_n)$ of n variables;
OUTPUT: "Yes" if and only if ϕ is satisfiable.

- boolean formula: e.g.,

$$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

- A boolean formula is satisfiable if there is an assignment of `T` and `F` values to its variables such that f is evaluated to `T`.

1. Intractable problems and exhaustive search

Boolean formula satisfiability (SAT):

INPUT: a boolean formula $\phi(x_1, \ldots, x_n)$ of n variables;
OUTPUT: "Yes" if and only if ϕ is satisfiable.

- boolean formula: e.g.,

$$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

- A boolean formula is satisfiable if there is an assignment of `T` and `F` values to its variables such that f is evaluated to `T`.

  e.g.,
  $f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ is satisfiable

1. Intractable problems and exhaustive search

Boolean formula satisfiability (SAT):

INPUT: a boolean formula $\phi(x_1, \ldots, x_n)$ of n variables;
OUTPUT: "Yes" if and only if ϕ is satisfiable.

- boolean formula: e.g.,

$$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

- A boolean formula is satisfiable if there is an assignment of `T` and `F` values to its variables such that f is evaluated to `T`.

  e.g.,
  $f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ is satisfiable

  $g(x_1, x_2) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
  is not satisfiable.

1. Intractable problems and exhaustive search

Some facts about SAT

1. Intractable problems and exhaustive search

Some facts about SAT

- it is a **decision problem**;

1. Intractable problems and exhaustive search

Some facts about SAT

- it is a **decision problem**;
- from electronic circuit tests and other applications;

1. Intractable problems and exhaustive search

Some facts about SAT

- it is a **decision problem**;
- from electronic circuit tests and other applications;
- no **polynomial time** algorithms have been found;
  even of time complexity: $O(n^{1000})$ seemingly impossible

1. Intractable problems and exhaustive search

Some facts about SAT

- it is a **decision problem**;
- from electronic circuit tests and other applications;
- no **polynomial time** algorithms have been found;
  even of time complexity: $O(n^{1000})$ seemingly impossible
- the first problem was proved *NP-complete* [Cook' 1971]

1. Intractable problems and exhaustive search

Some facts about SAT

- it is a **decision problem**;
- from electronic circuit tests and other applications;
- no **polynomial time** algorithms have been found;
  even of time complexity: $O(n^{1000})$ seemingly impossible
- the first problem was proved *NP-complete* [Cook' 1971]
- can be solved by simple exhaustive search (in-class exercise)

1. Intractable problems and exhaustive search

Solve SAT problem with a recursive algorithm:

1. Intractable problems and exhaustive search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?

1. Intractable problems and exhaustive search

Solve SAT problem with a recursive algorithm:

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

1. Intractable problems and 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?

1. Intractable problems and 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

1. Intractable problems and 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?

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

1. Intractable problems and 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?

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

1. Intractable problems and 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?

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

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

1. Intractable problems and 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?

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

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

1. Intractable problems and exhaustive search

Solve SAT problem with a recursive algorithm:

- what data will the recursion be applied to?
  boolean formula $f(x_1, \dots, x_n)$
- what is the terminating (base) case?
  $n = 0$, formula without variables
- what is the recursive case?

$$f(x_1, \dots, x_{n-1}, x_n) = f(x_1, \dots, x_{n-1}, \mathtt{T}) \vee f(x_1, \dots, x_{n-1}, \mathtt{F})$$

$$f(x_1, \dots, x_{n-1}, \mathtt{T}) \Longrightarrow g(x_1, \dots, x_{n-1})$$
$$f(x_1, \dots, x_{n-1}, \mathtt{F})$$

1. Intractable problems and 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?

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

$$f(x_1, \ldots, x_{n-1}, \mathtt{T}) \Longrightarrow g(x_1, \ldots, x_{n-1})$$
$$f(x_1, \ldots, x_{n-1}, \mathtt{F}) \Longrightarrow h(x_1, \ldots, x_{n-1})$$

1. Intractable problems and exhaustive search

```
Algorithm SAT-Solver(f(x, n))

if n=0,
  return f;
else
  g(x, n-1) = f(x, n, xn=T);
  h(x, n-1) = f(x, n, xn=F);
  return SAT-Solver(g(x, n-1)) or SAT-Solver(h(x, n-1))
```

1. Intractable problems and exhaustive search

```
Algorithm SAT-Solver(f(x, n))

if n=0,
  return f;
else
  g(x, n-1) = f(x, n, xn=T);
  h(x, n-1) = f(x, n, xn=F);
  return SAT-Solver(g(x, n-1)) or SAT-Solver(h(x, n-1))
```

- Are all assignments searched?

1. Intractable problems and exhaustive search

```
Algorithm SAT-Solver(f(x, n))

if n=0,
  return f;
else
  g(x, n-1) = f(x, n, xn=T);
  h(x, n-1) = f(x, n, xn=F);
  return SAT-Solver(g(x, n-1)) or SAT-Solver(h(x, n-1))
```

- Are all assignments searched?
- draw a *search tree* based on the algorithm
  - what does the tree look like?
  - what does each path mean?

1. Intractable problems and exhaustive search

```
Algorithm SAT-Solver(f(x, n))

if n=0,
  return f;
else
  g(x, n-1) = f(x, n, xn=T);
  h(x, n-1) = f(x, n, xn=F);
  return SAT-Solver(g(x, n-1)) or SAT-Solver(h(x, n-1))
```

- Are all assignments searched?
- draw a *search tree* based on the algorithm
  - what does the tree look like?
  - what does each path mean? how many paths?

1. Intractable problems and exhaustive search

```
Algorithm SAT-Solver(f(x, n))

1. if n=0,
2.   return f;
3. else
4.   g(x, n-1) = f(x, n, xn=T);
5.   h(x, n-1) = f(x, n, xn=F);
6.   return SAT-Solver(g(x, n-1)) or SAT-Solver(h(x, n-1))
```

- Are all assignments searched?
- draw a *search tree* based on the algorithm
  - what does the tree look like?
  - what does each path mean? how many paths?
- time?

1. Intractable problems and exhaustive search

```
Algorithm SAT-Solver(f(x, n))

1. if n=0,
2.   return f;
3. else
4.   g(x, n-1) = f(x, n, xn=T);
5.   h(x, n-1) = f(x, n, xn=F);
6.   return SAT-Solver(g(x, n-1)) or SAT-Solver(h(x, n-1))
```

- Are all assignments searched?
- draw a *search tree* based on the algorithm
  - what does the tree look like?
  - what does each path mean? how many paths?
- time? $T(n) = 2T(n-1) + cn$, $T(0) = c$,

1. Intractable problems and exhaustive search

```
Algorithm SAT-Solver(f(x, n))

1. if n=0,
2.   return f;
3. else
4.   g(x, n-1) = f(x, n, xn=T);
5.   h(x, n-1) = f(x, n, xn=F);
6.   return SAT-Solver(g(x, n-1)) or SAT-Solver(h(x, n-1))
```

- Are all assignments searched?
- draw a *search tree* based on the algorithm
  - what does the tree look like?
  - what does each path mean? how many paths?
- time? $T(n) = 2T(n-1) + cn$, $T(0) = c$, $\Longrightarrow T(n) = \Theta(2^n)$

1. Intractable problems and exhaustive search

Non-naive exhaustive search

1. Intractable problems and exhaustive search

Non-naive exhaustive search

Max Independent Set:
Input: graph $G = (V, E)$;
Output: $I \subseteq V$, where $\forall\, u, v \in V$, $(u, v) \notin E$, s.t.,
$|I|$ is the maximum.

1. Intractable problems and exhaustive search

Non-naive exhaustive search

Max Independent Set:
Input: graph $G = (V, E)$;
Output: $I \subseteq V$, where $\forall\, u, v \in V$, $(u, v) \notin E$, s.t.,
$|I|$ is the maximum.

- enumerating all possible subsets of V, $\Omega(2^n)$;

1. Intractable problems and exhaustive search

Non-naive exhaustive search

Max Independent Set:
Input: graph $G = (V, E)$;
Output: $I \subseteq V$, where $\forall\, u, v \in V$, $(u, v) \notin E$, s.t.,
$|I|$ is the maximum.

- enumerating all possible subsets of V, $\Omega(2^n)$;
- non-naive exhaustive search

1. Intractable problems and exhaustive search

```
Function MaxIS-Solver (G, I);   // I initialized to empty
1. if G is not empty
2.    choose an arbitrary vertex v from G;
3.    let G1 = G - {v} - all neighbors of v;
4.    let G2 = G - {v};
5.    MaxIS-Solver (G1, I1);
6.    MaxIS-Solver (G2, I2);
9.    if |I1|+1 >= |I2|
7.       I = I U I1 U {v};
8.    else
9.       I = I U I2;
```

1. Intractable problems and exhaustive search

1. Intractable problems and exhaustive search

- running time $T(n) = T(n-1) + T(n-2) + n$, $n = |V|$

  $T(n) = O(1.619^n)$

1. Intractable problems and exhaustive search

- running time $T(n) = T(n-1) + T(n-2) + n$, $n = |V|$

  $T(n) = O(1.619^n)$

- can we do better?

1. Intractable problems and exhaustive search

- running time $T(n) = T(n-1) + T(n-2) + n$, $n = |V|$

  $T(n) = O(1.619^n)$

- can we do better? Yes! (by revising the algorithm slightly)

$$T(n) = T(n-1) + T(n-3) + n$$

1. Intractable problems and exhaustive search

- running time $T(n) = T(n-1) + T(n-2) + n$, $n = |V|$

  $T(n) = O(1.619^n)$

- can we do better? Yes! (by revising the algorithm slightly)

$$T(n) = T(n-1) + T(n-3) + n$$

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

2. Decision vs search problems

2. Decision vs search problems

- Both **Hamiltonian Path** and **SAT** are decision problem

2. Decision vs search problems

- Both **Hamiltonian Path** and **SAT** are decision problem
- **Max Independent Set** is an optimization problem.

2. Decision vs search problems

- Both **Hamiltonian Path** and **SAT** are decision problem
- **Max Independent Set** is an optimization problem.

Claim: To investigate tractability, it suffices to study decision problems

2. Decision vs search problems

2. Decision vs search problems

Max Independent Set:
Input: graph $G = (V, E)$;
Output: $I \subseteq V$, where $\forall\, u, v \in V$, $(u, v) \notin E$, s.t.,
$|I|$ is the maximum.

Independent Set (decision version)
Input: graph $G = (V, E)$ and k;
Output: "yes" iff G has an independent set of size k;

---

2. Decision vs search problems

Max Independent Set:

Input: graph $G = (V, E)$;

Output: $I \subseteq V$, where $\forall\, u, v \in V$, $(u, v) \notin E$, s.t.,
$|I|$ is the maximum.

Independent Set (decision version)

Input: graph $G = (V, E)$ and k;

Output: "yes" iff G has an independent set of size k;

What are the relationship between the two problems?

2. Decision vs search problems

Max Independent Set:
Input: graph $G = (V, E)$;
Output: $I \subseteq V$, where $\forall\, u, v \in V$, $(u, v) \notin E$, s.t.,
$|I|$ is the maximum.

Independent Set (decision version)
Input: graph $G = (V, E)$ and k;
Output: "yes" iff G has an independent set of size k;

What are the relationship between the two problems?

- **Max Independent Set** is solvable in $O(n^d)$
  $\Longrightarrow$ **Independent Set** is solvable in $O(n^d)$

2. Decision vs search problems

Max Independent Set:

Input: graph $G = (V, E)$;

Output: $I \subseteq V$, where $\forall\, u, v \in V$, $(u, v) \notin E$, s.t.,
$|I|$ is the maximum.

Independent Set (decision version)

Input: graph $G = (V, E)$ and k;

Output: "yes" iff G has an independent set of size k;

What are the relationship between the two problems?

- **Max Independent Set** is solvable in $O(n^d)$
  $\Longrightarrow$ **Independent Set** is solvable in $O(n^d)$
- **Independent Set** is solvable in $O(n^c)$
  $\Longrightarrow$ **Max Independent Set** is solvable in $O(n^{c+1})$

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

Construct Algorithm B_{MIS}
(using A_{IS} as a block box to solve **Max Independent Set**)

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

Construct Algorithm B_{MIS}
(using A_{IS} as a block box to solve **Max Independent Set**)

- Input: $G = (V, E)$;

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

Construct Algorithm B_{MIS}
(using A_{IS} as a block box to solve **Max Independent Set**)

- Input: $G = (V, E)$;
- for all $k = 1, 2, \ldots,$ query $A_{\mathrm{IS}}(G, k) =?$

2. Decision vs search problems

Assume algorithm $A_{\rm IS}$ for decision problem **Independent Set**

Construct Algorithm $B_{\rm MIS}$
(using $A_{\rm IS}$ as a block box to solve **Max Independent Set**)

- Input: $G = (V, E)$;
- for all $k = 1, 2, \ldots,$ query $A_{\rm IS}(G, k) =?$
  let k_0 be the largest allowing $A_{\rm IS}(G, k_0) =$ "yes".

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

Construct Algorithm B_{MIS}
(using A_{IS} as a block box to solve **Max Independent Set**)

- Input: $G = (V, E)$;
- for all $k = 1, 2, \ldots,$ query $A_{\mathrm{IS}}(G, k) =?$
  let k_0 be the largest allowing $A_{\mathrm{IS}}(G, k_0) =$ "yes".
- for all vertices v in G, query $A_{\mathrm{IS}}(G \backslash \{v\}, k_0) =?$

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

Construct Algorithm B_{MIS}
(using A_{IS} as a block box to solve **Max Independent Set**)

- Input: $G = (V, E)$;
- for all $k = 1, 2, \dots,$ query $A_{\text{IS}}(G, k) =?$
  let k_0 be the largest allowing $A_{\text{IS}}(G, k_0) =$ "yes".
- for all vertices v in G, query $A_{\text{IS}}(G \backslash \{v\}, k_0) =?$
  if answer = "yes" remove v from V
  else keep and mark v in G.

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

Construct Algorithm B_{MIS}
(using A_{IS} as a block box to solve **Max Independent Set**)

- Input: $G = (V, E)$;
- for all $k = 1, 2, \ldots,$ query $A_{\text{IS}}(G, k) =?$
  let k_0 be the largest allowing $A_{\text{IS}}(G, k_0) =$ "yes".
- for all vertices v in G, query $A_{\text{IS}}(G \backslash \{v\}, k_0) =?$
  if answer = "yes" remove v from V
  else keep and mark v in G.
- return V (as a max independent set for G)

2. Decision vs search problems

Assume algorithm A_{IS} for decision problem **Independent Set**

Construct Algorithm B_{MIS}
(using A_{IS} as a block box to solve **Max Independent Set**)

- Input: $G = (V, E)$;
- for all $k = 1, 2, \ldots,$ query $A_{\text{IS}}(G, k) =?$
  let k_0 be the largest allowing $A_{\text{IS}}(G, k_0) =$ "yes".
- for all vertices v in G, query $A_{\text{IS}}(G \backslash \{v\}, k_0) =?$
  if answer = "yes" remove v from V
  else keep and mark v in G.
- return V (as a max independent set for G)

Time: $T_{B_{\text{MIS}}} = T_{A_{\text{IS}}} \times O(n)$

2. Decision vs search problems

2. Decision vs search problems

Theorem: **Max Independent Set** is solvable in polynomial time if and only if **Independent Set** is.

2. Decision vs search problems

2. Decision vs search problems

Consider the problem **MST** vs problem **MST-D**:

2. Decision vs search problems

Consider the problem **MST** vs problem **MST-D**:

MST-D:
Input: G and w;
Output: "yes" iff G has a m.s.t. of weight $\leq w$.

2. Decision vs search problems

Consider the problem **MST** vs problem **MST-D**:

MST-D:
Input: G and w;
Output: "yes" iff G has a m.s.t. of weight $\leq w$.

- Algorithm for **MST-D** can be used to solve **MST** (without using too much extra time).

2. Decision vs search problems

Consider the problem **MST** vs problem **MST-D**:

MST-D:
Input: G and w;
Output: "yes" iff G has a m.s.t. of weight $\leq w$.

- Algorithm for **MST-D** can be used to solve **MST** (without using too much extra time).

  How? (In classroom exercise)

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

  Given G, if also given a path p,
  we can verify if p is indeed a Hamiltonian path for G,

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

  Given G, if also given a path p,
  we can verify if p is indeed a Hamiltonian path for G, easily

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

  Given G, if also given a path p,
  we can verify if p is indeed a Hamiltonian path for G, easily

  here is how:
  (1) check if p contains $n-1$ edges;

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

  Given G, if also given a path p,
  we can verify if p is indeed a Hamiltonian path for G, easily

  here is how:
  (1) check if p contains $n-1$ edges;
  (2) all edges on p are legit;

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

  Given G, if also given a path p,
  we can verify if p is indeed a Hamiltonian path for G, easily

  here is how:
  (1) check if p contains $n-1$ edges;
  (2) all edges on p are legit;
  (3) all vertices on the path are different

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

  Given G, if also given a path p,
  we can verify if p is indeed a Hamiltonian path for G, easily

  here is how:
  (1) check if p contains $n-1$ edges;
  (2) all edges on p are legit;
  (3) all vertices on the path are different

  time?

3. Polynomial solvable vs verifiable

From now on, discussions will be focused on decision problems only.

- **Hamiltonian Path** cannot be answered in a polynomial time;
- but an answer can be verified in polynomial time;

  Given G, if also given a path p,
  we can verify if p is indeed a Hamiltonian path for G, easily

  here is how:
  (1) check if p contains $n-1$ edges;
  (2) all edges on p are legit;
  (3) all vertices on the path are different

  time? – a polynomial in n.

3. Polynomial solvable vs verifiable

Conclusions:

3. Polynomial solvable vs verifiable

Conclusions:

- **Hamiltonian Path** can be verified in polynomial time;

3. Polynomial solvable vs verifiable

Conclusions:

- **Hamiltonian Path** can be verified in polynomial time;
- **SAT** can be verified in polynomial time; why?

3. Polynomial solvable vs verifiable

Conclusions:

- **Hamiltonian Path** can be verified in polynomial time;
- **SAT** can be verified in polynomial time; why?
- **Independent Set** can be verified in polynomial time; why?

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

For every decision problem $D \in \mathcal{NP}$, which decides on input x to answer "yes" or "no",

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

For every decision problem $D \in \mathcal{NP}$, which decides on input x to answer "yes" or "no", there exists a verifier V_D such that

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

For every decision problem $D \in \mathcal{NP}$, which decides on input x to answer "yes" or "no", there exists a verifier V_D such that

$$\forall x, D(x) \begin{cases} = \text{"}yes\text{"} & \exists y, V_D(x,y) = \text{ TRUE} \\ = \text{"}no\text{"} & \forall y, V_D(x,y) = \text{ FALSE} \end{cases}$$

where V_D can be computed in polynomial time, and y is called a *certificate* or *witness* to an "yes" answer.

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

For every decision problem $D \in \mathcal{NP}$, which decides on input x to answer "yes" or "no", there exists a verifier V_D such that

$$\forall x, D(x) \begin{cases} = \text{"}yes\text{"} & \exists y, V_D(x, y) = \text{ TRUE} \\ = \text{"}no\text{"} & \forall y, V_D(x, y) = \text{ FALSE} \end{cases}$$

where V_D can be computed in polynomial time, and y is called a *certificate* or *witness* to an "yes" answer.

Note: D may not be computed in polynomial time

3. Polynomial solvable vs verifiable

Analog: a trial case:

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)
- x a suspect, to be determined if $C(x) = 1/0$ (guilty or not)

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)
- x a suspect, to be determined if $C(x) = 1/0$ (guilty or not)
- p is declared guilty **iff** there should be some evidence y

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)
- x a suspect, to be determined if $C(x) = 1/0$ (guilty or not)
- p is declared guilty **iff** there should be some evidence y
- y can be verified by a jury, i.e., $V(x, y) =$ TRUE

---

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)
- x a suspect, to be determined if $C(x) = 1/0$ (guilty or not)
- p is declared guilty **iff** there should be some evidence y
- y can be verified by a jury, i.e., $V(x, y) =$ TRUE

Collecting evidence (by prosecutor) may take a long time

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)
- x a suspect, to be determined if $C(x) = 1/0$ (guilty or not)
- p is declared guilty **iff** there should be some evidence y
- y can be verified by a jury, i.e., $V(x, y) =$ TRUE

Collecting evidence (by prosecutor) may take a long time
i.e., C is difficult to compute

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)
- x a suspect, to be determined if $C(x) = 1/0$ (guilty or not)
- p is declared guilty **iff** there should be some evidence y
- y can be verified by a jury, i.e., $V(x, y) =$ TRUE

Collecting evidence (by prosecutor) may take a long time
i.e., C is difficult to compute

But verification by the jury can be done relatively easily

3. Polynomial solvable vs verifiable

Analog: a trial case:

- C a decision problem (trial)
- x a suspect, to be determined if $C(x) = 1/0$ (guilty or not)
- p is declared guilty **iff** there should be some evidence y
- y can be verified by a jury, i.e., $V(x, y) =$ TRUE

Collecting evidence (by prosecutor) may take a long time
i.e., C is difficult to compute

But verification by the jury can be done relatively easily
i.e., V is easy to compute

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$. How to prove these problems are in $\mathcal{NP}$?

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

$\mathcal{P} \subseteq \mathcal{NP}$

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

$\mathcal{P} \subseteq \mathcal{NP}$ why?

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

$\mathcal{P} \subseteq \mathcal{NP}$ why?

- **Reachabililty** is in $\mathcal{NP}$.

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$. How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

$\mathcal{P} \subseteq \mathcal{NP}$ why?

- **Reachabililty** is in $\mathcal{NP}$.
- Is **SAT** in $\mathcal{NP}$?

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

$\mathcal{P} \subseteq \mathcal{NP}$ why?

- **Reachabililty** is in $\mathcal{NP}$.
- Is **SAT** in $\mathcal{NP}$? Is **Hamiltonian Path** in $\mathcal{NP}$?

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

$\mathcal{P} \subseteq \mathcal{NP}$ why?

- **Reachabililty** is in $\mathcal{NP}$.
- Is **SAT** in $\mathcal{NP}$? Is **Hamiltonian Path** in $\mathcal{NP}$?
- Are all problems in $\mathcal{NP}$ also in $\mathcal{P}$?

3. Polynomial solvable vs verifiable

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

- **SAT**, **Hamiltonian Path**, **Independent Set** are all in $\mathcal{NP}$.
  How to prove these problems are in $\mathcal{NP}$?

Definition: $\mathcal{P}$ is the class of decision problems whose answers can be decided in polynomial time.

- **Reachabililty** is in $\mathcal{P}$.

$\mathcal{P} \subseteq \mathcal{NP}$ why?

- **Reachabililty** is in $\mathcal{NP}$.
- Is **SAT** in $\mathcal{NP}$? Is **Hamiltonian Path** in $\mathcal{NP}$?
- Are all problems in $\mathcal{NP}$ also in $\mathcal{P}$?
- $\mathcal{NP} = \mathcal{P}$?

3. Polynomial solvable vs verifiable

3. Polynomial solvable vs verifiable

- Many decision problems are in $\mathcal{NP}$, but not known to be in $\mathcal{P}$

3. Polynomial solvable vs verifiable

- Many decision problems are in $\mathcal{NP}$, but not known to be in $\mathcal{P}$
- How about to prove their lower bounds?

3. Polynomial solvable vs verifiable

- Many decision problems are in $\mathcal{NP}$, but not known to be in $\mathcal{P}$
- How about to prove their lower bounds?
- Some of these problems appear to be "equivalent" – if one of them is in $\mathcal{P}$, so are some others in $\mathcal{P}$.

3. Polynomial solvable vs verifiable

- Many decision problems are in $\mathcal{NP}$, but not known to be in $\mathcal{P}$
- How about to prove their lower bounds?
- Some of these problems appear to be "equivalent" – if one of them is in $\mathcal{P}$, so are some others in $\mathcal{P}$.

  i.e., algorithm for one decision problem can be used to solve another decision problem, without increasing more than a polynomial factor in computation time.

4. $\mathcal{NP}$-completeness theory

Reduction for decision problems

4. $\mathcal{NP}$-completeness theory

Reduction for decision problems

Definition: Let D_1 and D_2 be two decision problems. A mapping $f : \Sigma^* \to \Sigma^*$ is a reduction from D_1 to D_2, if for any $x \in \Sigma^*$,

$$D_1(x) = \text{"yes" if and only if } D_2(f(x)) = \text{"yes"}.$$

4. $\mathcal{NP}$-completeness theory

Reduction for decision problems

Definition: Let D_1 and D_2 be two decision problems. A mapping $f : \Sigma^* \to \Sigma^*$ is a reduction from D_1 to D_2, if for any $x \in \Sigma^*$,

$$D_1(x) = \text{"yes" if and only if } D_2(f(x)) = \text{"yes"}.$$

where $\Sigma = \{0, 1\}$, Σ^* is called the *universe*.

4. $\mathcal{NP}$-completeness theory

Reduction for decision problems

Definition: Let D_1 and D_2 be two decision problems. A mapping $f : \Sigma^* \rightarrow \Sigma^*$ is a reduction from D_1 to D_2, if for any $x \in \Sigma^*$,

$$D_1(x) = \text{"yes" if and only if } D_2(f(x)) = \text{"yes"}.$$

where $\Sigma = \{0, 1\}$, Σ^* is called the *universe*.

denoted with $D_1 \leq D_2$

That is:

4. $\mathcal{NP}$-completeness theory

Reduction for decision problems

Definition: Let D_1 and D_2 be two decision problems. A mapping $f : \Sigma^* \rightarrow \Sigma^*$ is a reduction from D_1 to D_2, if for any $x \in \Sigma^*$,

$$D_1(x) = \text{"yes" if and only if } D_2(f(x)) = \text{"yes"}.$$

where $\Sigma = \{0, 1\}$, Σ^* is called the *universe*.

denoted with $D_1 \leq D_2$

That is:

- to answer question D_1 on x;

4. $\mathcal{NP}$-completeness theory

Reduction for decision problems

Definition: Let D_1 and D_2 be two decision problems. A mapping $f : \Sigma^* \to \Sigma^*$ is a reduction from D_1 to D_2, if for any $x \in \Sigma^*$,

$$D_1(x) = \text{"yes"} \text{ if and only if } D_2(f(x)) = \text{"yes"}.$$

where $\Sigma = \{0, 1\}$, Σ^* is called the *universe*.

denoted with $D_1 \leq D_2$

That is:

- to answer question D_1 on x;
- the transformation converts x to $y = f(x)$ via mapping f;

4. $\mathcal{NP}$-completeness theory

Reduction for decision problems

Definition: Let D_1 and D_2 be two decision problems. A mapping $f : \Sigma^* \rightarrow \Sigma^*$ is a reduction from D_1 to D_2, if for any $x \in \Sigma^*$,

$D_1(x) =$ "yes" if and only if $D_2(f(x)) =$ "yes".

where $\Sigma = \{0, 1\}$, Σ^* is called the *universe*.

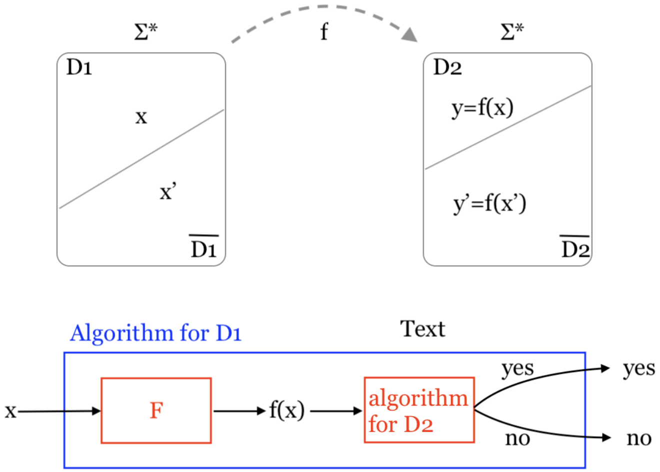
denoted with $D_1 \leq D_2$

That is:

- to answer question D_1 on x;
- the transformation converts x to $y = f(x)$ via mapping f;
- an answer to D_2 on y is then used as answer to x for D_1;

4. $\mathcal{NP}$-completeness theory

Reduction from D_1 to D_2



4. $\mathcal{NP}$-completeness theory

Example: transformation from **SAT** and **Independent Set** (**IS**):

4. $\mathcal{NP}$-completeness theory

Example: transformation from **SAT** and **Independent Set** (**IS**):

- need a mapping function $f : \Sigma^* \to \Sigma^*$;

4. $\mathcal{NP}$-completeness theory

Example: transformation from **SAT** and **Independent Set** (**IS**):

- need a mapping function $f : \Sigma^* \to \Sigma^*$;
- for any input ϕ, $f(\phi) = \langle G_\phi, k_\phi \rangle$;

4. $\mathcal{NP}$-completeness theory

Example: transformation from **SAT** and **Independent Set** (**IS**):

- need a mapping function $f : \Sigma^* \to \Sigma^*$;
- for any input ϕ, $f(\phi) = \langle G_\phi, k_\phi \rangle$;
- satisfying:

$$\mathbf{SAT}(\phi) = \text{"}yes\text{"} \Longleftrightarrow \mathbf{IS}(\langle G_\phi, k_\phi \rangle) = \text{"}yes\text{"}$$

4. $\mathcal{NP}$-completeness theory

Example: transformation from **SAT** and **Independent Set** (**IS**):

- need a mapping function $f : \Sigma^* \to \Sigma^*$;
- for any input ϕ, $f(\phi) = \langle G_\phi, k_\phi \rangle$;
- satisfying:

$$\mathbf{SAT}(\phi) = \text{"}yes\text{"} \Longleftrightarrow \mathbf{IS}(\langle G_\phi, k_\phi \rangle) = \text{"}yes\text{"}$$

- does such a reduction exist?

4. $\mathcal{NP}$-completeness theory

4. $\mathcal{NP}$-completeness theory

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

4. $\mathcal{NP}$-completeness theory

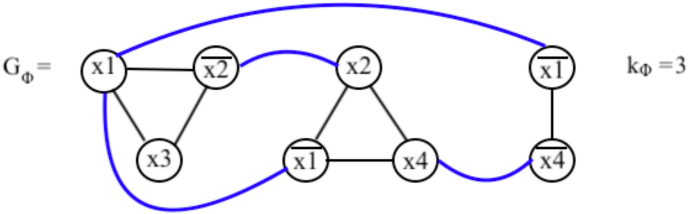
$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

mapped by f to $\downarrow$

4. $\mathcal{NP}$-completeness theory

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

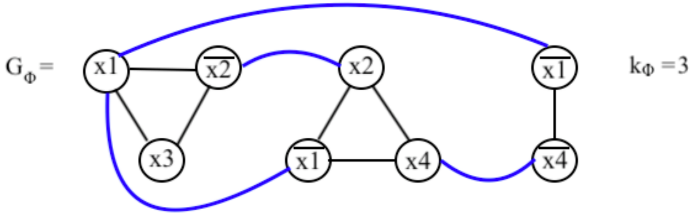
mapped by f to $\downarrow$



4. $\mathcal{NP}$-completeness theory

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$$

mapped by f to $\downarrow$



- ϕ is satisfiable $\Longleftrightarrow$ G_ϕ has ind. set of size ≥ 3.

4. $\mathcal{NP}$-completeness theory

4. $\mathcal{NP}$-completeness theory

- f follows the rule:

4. $\mathcal{NP}$-completeness theory

- f follows the rule:

  (1) map every clause to a complete graph;
  (2) connect vertices formed by a variable and its negation;

4. $\mathcal{NP}$-completeness theory

- f follows the rule:

  (1) map every clause to a complete graph;
  (2) connect vertices formed by a variable and its negation;

- more examples:

4. $\mathcal{NP}$-completeness theory

- f follows the rule:

  (1) map every clause to a complete graph;
  (2) connect vertices formed by a variable and its negation;

- more examples:

  - what will the following formula be reduced to?

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

4. $\mathcal{NP}$-completeness theory

- f follows the rule:

  (1) map every clause to a complete graph;
  (2) connect vertices formed by a variable and its negation;

- more examples:

  - what will the following formula be reduced to?

$$\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

  - what about

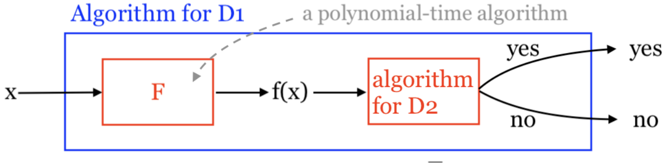
$$\psi(x_1, x_2) = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

4. $\mathcal{NP}$-completeness theory

Definition: Let f be a mapping realizing $D_1 \leq D_2$. The reduction is called a polynomial-time reduction if $f(x)$ can be computed in time $O(|x|^c)$ for some fixed constant c. It is denoted with $D_1 \leq_p D_2$.

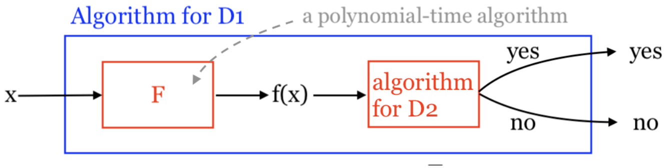
4. $\mathcal{NP}$-completeness theory

Definition: Let f be a mapping realizing $D_1 \leq D_2$. The reduction is called a polynomial-time reduction if $f(x)$ can be computed in time $O(|x|^c)$ for some fixed constant c. It is denoted with $D_1 \leq_p D_2$.



4. $\mathcal{NP}$-completeness theory

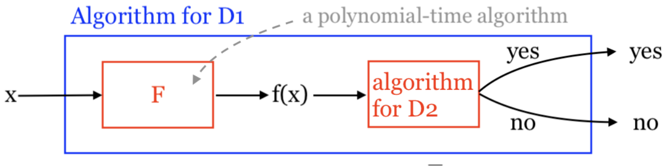
Definition: Let f be a mapping realizing $D_1 \leq D_2$. The reduction is called a polynomial-time reduction if $f(x)$ can be computed in time $O(|x|^c)$ for some fixed constant c. It is denoted with $D_1 \leq_p D_2$.



- If algorithm F runs in time $O(n^3)$;

4. $\mathcal{NP}$-completeness theory

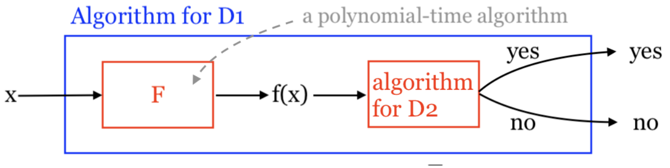
Definition: Let f be a mapping realizing $D_1 \leq D_2$. The reduction is called a polynomial-time reduction if $f(x)$ can be computed in time $O(|x|^c)$ for some fixed constant c. It is denoted with $D_1 \leq_p D_2$.



- If algorithm F runs in time $O(n^3)$;
- if algorithm for D_2 runs in time $O(n^2)$;

4. $\mathcal{NP}$-completeness theory

Definition: Let f be a mapping realizing $D_1 \leq D_2$. The reduction is called a polynomial-time reduction if $f(x)$ can be computed in time $O(|x|^c)$ for some fixed constant c. It is denoted with $D_1 \leq_p D_2$.



- If algorithm F runs in time $O(n^3)$;
- if algorithm for D_2 runs in time $O(n^2)$;
- what is the time of algorithm for D_1?

4. $\mathcal{NP}$-completeness theory

Theorem: Let $D_1 \leq_p D_2$. If $D_2 \in \mathcal{P}$, then $D_1 \in \mathcal{P}$ also.

4. $\mathcal{NP}$-completeness theory

Theorem: Let $D_1 \leq_p D_2$. If $D_2 \in \mathcal{P}$, then $D_1 \in \mathcal{P}$ also.

- Fact1: **SAT** $\leq_p$ **IS**.

4. $\mathcal{NP}$-completeness theory

Theorem: Let $D_1 \leq_p D_2$. If $D_2 \in \mathcal{P}$, then $D_1 \in \mathcal{P}$ also.

- Fact1: **SAT** $\leq_p$ **IS**.
- Conclusion: If **IS** is in $\mathcal{P}$, so is **SAT**.

4. $\mathcal{NP}$-completeness theory

Theorem: Let $D_1 \leq_p D_2$. If $D_2 \in \mathcal{P}$, then $D_1 \in \mathcal{P}$ also.

- Fact1: **SAT** $\leq_p$ **IS**.
- Conclusion: If **IS** is in $\mathcal{P}$, so is **SAT**.

  SAT is "not harder than" **IS** w.r.t. polynomial-time, i.e.,

4. $\mathcal{NP}$-completeness theory

Theorem: Let $D_1 \leq_p D_2$. If $D_2 \in \mathcal{P}$, then $D_1 \in \mathcal{P}$ also.

- Fact1: **SAT** $\leq_p$ **IS**.
- Conclusion: If **IS** is in $\mathcal{P}$, so is **SAT**.

  SAT is "not harder than" **IS** w.r.t. polynomial-time, i.e.,
  IS is "not easier than" **SAT** w.r.t. polynomial-time

4. $\mathcal{NP}$-completeness theory

Transitivity of $\leq_p$:

4. $\mathcal{NP}$-completeness theory

Transitivity of $\leq_p$:

Proposition: If $D_1 \leq_p D_2$ and $D_2 \leq_p D_3$, then $D_1 \leq_p D_3$.

4. $\mathcal{NP}$-completeness theory

Transitivity of $\leq_p$:

Proposition: If $D_1 \leq_p D_2$ and $D_2 \leq_p D_3$, then $D_1 \leq_p D_3$.

Proof: (in-classroom exercise)

4. $\mathcal{NP}$-completeness theory

4. $\mathcal{NP}$-completeness theory

- It is known that **SAT**, **IS**, **Clique**, **Hamiltonian Path** and many others can be polynomially reduced to each other;

4. $\mathcal{NP}$-completeness theory

- It is known that **SAT**, **IS**, **Clique**, **Hamiltonian Path** and many others can be polynomially reduced to each other;
- These problems are all in the class $\mathcal{NP}$;

4. $\mathcal{NP}$-completeness theory

- It is known that **SAT**, **IS**, **Clique**, **Hamiltonian Path** and many others can be polynomially reduced to each other;
- These problems are all in the class $\mathcal{NP}$;
- they seem to form a special group in $\mathcal{NP}$;

4. $\mathcal{NP}$-completeness theory

- It is known that **SAT**, **IS**, **Clique**, **Hamiltonian Path** and many others can be polynomially reduced to each other;
- These problems are all in the class $\mathcal{NP}$;
- they seem to form a special group in $\mathcal{NP}$;
- to understand these problems, a new notion is needed;

4. $\mathcal{NP}$-completeness theory

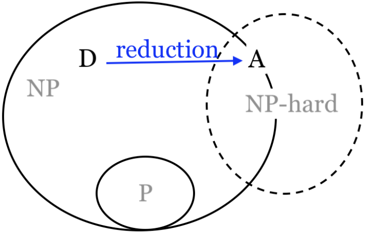
Definition: Decision problem A is called $\mathcal{NP}$-*hard* if for every problem $D \in \mathcal{NP}$, $D \leq_p A$

That is, A is "NOT easier than" any problem in class $\mathcal{NP}$.

4. $\mathcal{NP}$-completeness theory

Definition: Decision problem A is called $\mathcal{NP}$-*hard* if for every problem $D \in \mathcal{NP}$, $D \leq_p A$

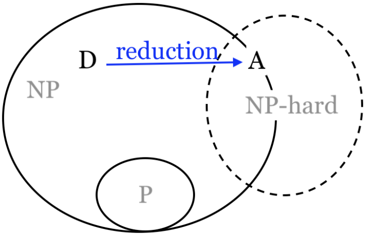
That is, A is "NOT easier than" any problem in class $\mathcal{NP}$.



4. $\mathcal{NP}$-completeness theory

Definition: Decision problem A is called $\mathcal{NP}$-*hard* if for every problem $D \in \mathcal{NP}$, $D \leq_p A$

That is, A is "NOT easier than" any problem in class $\mathcal{NP}$.



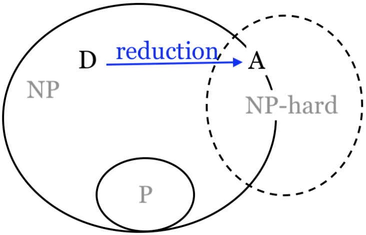
Definition: Decision problem A is called $\mathcal{NP}$-*complete* if

(1) it is $\mathcal{NP}$-hard;
(2) itself is also in $\mathcal{NP}$;

4. $\mathcal{NP}$-completeness theory

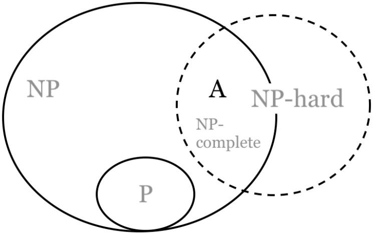
Definition: Decision problem A is called $\mathcal{NP}$*-hard* if for every problem $D \in \mathcal{NP}$, $D \leq_p A$

That is, A is "NOT easier than" any problem in class $\mathcal{NP}$.



Definition: Decision problem A is called $\mathcal{NP}$*-complete* if

(1) it is $\mathcal{NP}$-hard;
(2) itself is also in $\mathcal{NP}$;



4. $\mathcal{NP}$-completeness theory

Theorem: Let A be an $\mathcal{NP}$-hard problem. If $A \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

proof (in-classroom exercise)

Corollary: Any $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problem cannot be solved in polynomial time unless $\mathcal{P} = \mathcal{NP}$.

4. $\mathcal{NP}$-completeness theory

Theorem: Problem **SAT** is $\mathcal{NP}$-complete. That is,

(1) **SAT** $\in\mathcal{NP}$, and
(2) **SAT** is $\mathcal{NP}$-hard.

4. $\mathcal{NP}$-completeness theory

Theorem: Problem **SAT** is $\mathcal{NP}$-complete. That is,

(1) **SAT** $\in\mathcal{NP}$, and
(2) **SAT** is $\mathcal{NP}$-hard.

Proof:

4. $\mathcal{NP}$-completeness theory

Theorem: Problem **SAT** is $\mathcal{NP}$-complete. That is,

(1) **SAT** $\in \mathcal{NP}$, and
(2) **SAT** is $\mathcal{NP}$-hard.

Proof:

- **SAT** $\in \mathcal{NP}$. (in-classroom exercise)

4. $\mathcal{NP}$-completeness theory

Theorem: Problem **SAT** is $\mathcal{NP}$-complete. That is,

(1) **SAT** $\in \mathcal{NP}$, and
(2) **SAT** is $\mathcal{NP}$-hard.

Proof:

- **SAT** $\in \mathcal{NP}$. (in-classroom exercise)
- **SAT** is $\mathcal{NP}$-hard.

4. $\mathcal{NP}$-completeness theory

Theorem: Problem **SAT** is $\mathcal{NP}$-complete. That is,

(1) **SAT** $\in \mathcal{NP}$, and
(2) **SAT** is $\mathcal{NP}$-hard.

Proof:

- **SAT** $\in \mathcal{NP}$. (in-classroom exercise)
- **SAT** is $\mathcal{NP}$-hard.
  this is to show that $\forall D \in \mathcal{NP}$,

$$D \leq_p \textbf{SAT}$$

4. $\mathcal{NP}$-completeness theory

Theorem: Problem **SAT** is $\mathcal{NP}$-complete. That is,

(1) **SAT** $\in\mathcal{NP}$, and
(2) **SAT** is $\mathcal{NP}$-hard.

Proof:

- **SAT** $\in\mathcal{NP}$. (in-classroom exercise)
- **SAT** is $\mathcal{NP}$-hard.
  this is to show that $\forall D \in \mathcal{NP}$,

$$D \leq_p \mathbf{SAT}$$

- very difficult task since D can be any problem in $\mathcal{NP}$.

4. $\mathcal{NP}$-completeness theory

Theorem: Problem **SAT** is $\mathcal{NP}$-complete. That is,

(1) **SAT** $\in \mathcal{NP}$, and
(2) **SAT** is $\mathcal{NP}$-hard.

Proof:

- **SAT** $\in \mathcal{NP}$. (in-classroom exercise)
- **SAT** is $\mathcal{NP}$-hard.
  this is to show that $\forall D \in \mathcal{NP}$,

$$D \leq_p \mathbf{SAT}$$

- very difficult task since D can be any problem in $\mathcal{NP}$.
- [Cook'1971] via nondeterministic Turing machine, very involved.

4. $\mathcal{NP}$-completeness theory

To prove a problem A to be $\mathcal{NP}$-hard, there are two methods:

4. $\mathcal{NP}$-completeness theory

To prove a problem A to be $\mathcal{NP}$-hard, there are two methods:

(1) directly prove that $\forall D \in \mathcal{NP}$, $D \leq_p A$,

4. $\mathcal{NP}$-completeness theory

To prove a problem A to be $\mathcal{NP}$-hard, there are two methods:

(1) directly prove that $\forall D \in \mathcal{NP}$, $D \leq_p A$,

(2) choose a known $\mathcal{NP}$-hard problem B, and prove $B \leq_p A$.

4. $\mathcal{NP}$-completeness theory

To prove a problem A to be $\mathcal{NP}$-hard, there are two methods:

(1) directly prove that $\forall D \in \mathcal{NP}$, $D \leq_p A$,

(2) choose a known $\mathcal{NP}$-hard problem B, and prove $B \leq_p A$.
(why does (2) work?)

4. $\mathcal{NP}$-completeness theory

To prove a problem A to be $\mathcal{NP}$-hard, there are two methods:

(1) directly prove that $\forall D \in \mathcal{NP}$, $D \leq_p A$,

(2) choose a known $\mathcal{NP}$-hard problem B, and prove $B \leq_p A$.
(why does (2) work?)

But eventually there should be some problem first proved to be $\mathcal{NP}$-hard.

4. $\mathcal{NP}$-completeness theory

Instead of proving **SAT** to be $\mathcal{NP}$-hard, we prove the following decision problem **CSAT** to be $\mathcal{NP}$-hard.

4. $\mathcal{NP}$-completeness theory

Instead of proving **SAT** to be $\mathcal{NP}$-hard, we prove the following decision problem **CSAT** to be $\mathcal{NP}$-hard. That is,

$$\forall D \in \mathcal{NP}, D \leq_p \textbf{CSAT}$$

Circuit-Satisfiability (**CSAT**)
Input: a boolean circuit C of
boolean gates $x_1, \ldots, x_n$;
Output: "yes" iff C is satisfiable.

4. $\mathcal{NP}$-completeness theory

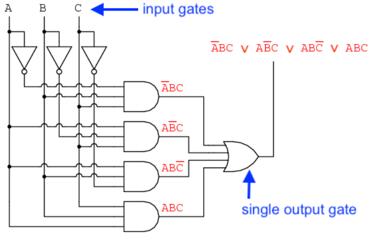
Instead of proving **SAT** to be $\mathcal{NP}$-hard, we prove the following decision problem **CSAT** to be $\mathcal{NP}$-hard. That is,

$$\forall D \in \mathcal{NP}, D \leq_p \textbf{CSAT}$$

Circuit-Satisfiability (CSAT)

Input: a boolean circuit C of boolean gates $x_1, \ldots, x_n$;

Output: "yes" iff C is satisfiable.



4. $\mathcal{NP}$-completeness theory

Recall the definition of arbitrary problem $D \in \mathcal{NP}$:

4. $\mathcal{NP}$-completeness theory

Recall the definition of arbitrary problem $D \in \mathcal{NP}$:

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

For every decision problem $D \in \mathcal{NP}$, which decides on input x to answer "yes" or "no", there exists a verifier V_D such that

$$\forall x, D(x) \begin{cases} = \text{"}yes\text{"} & \exists y, V_D(x, y) = \text{ TRUE} \\ = \text{"}no\text{"} & \forall y, V_D(x, y) = \text{ FALSE} \end{cases}$$

where V_D can be computed in polynomial time, and y is called a *certificate* or *witness* to an "yes" answer.

4. $\mathcal{NP}$-completeness theory

Recall the definition of arbitrary problem $D \in \mathcal{NP}$:

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

For every decision problem $D \in \mathcal{NP}$, which decides on input x to answer "yes" or "no", there exists a verifier V_D such that

$$\forall x, D(x) \begin{cases} = \text{"}yes\text{"} & \exists y, V_D(x,y) = \text{ TRUE} \\ = \text{"}no\text{"} & \forall y, V_D(x,y) = \text{ FALSE} \end{cases}$$

where V_D can be computed in polynomial time, and y is called a *certificate* or *witness* to an "yes" answer.

That is for every input x,

$$D(x) = \text{"}yes\text{"} \text{ iff } \exists y,$$

4. $\mathcal{NP}$-completeness theory

Recall the definition of arbitrary problem $D \in \mathcal{NP}$:

Definition: $\mathcal{NP}$ is the class of decision problems whose answers can be verified in polynomial time.

That is,

For every decision problem $D \in \mathcal{NP}$, which decides on input x to answer "yes" or "no", there exists a verifier V_D such that

$$\forall x, D(x) \begin{cases} = \text{"}yes\text{"} & \exists y, V_D(x,y) = \text{ TRUE} \\ = \text{"}no\text{"} & \forall y, V_D(x,y) = \text{ FALSE} \end{cases}$$

where V_D can be computed in polynomial time, and y is called a *certificate* or *witness* to an "yes" answer.

That is for every input x,

$$D(x) = \text{"}yes\text{"} \text{ iff } \exists y, V_D(x,y) = TRUE$$

4. $\mathcal{NP}$-completeness theory

For every input x,

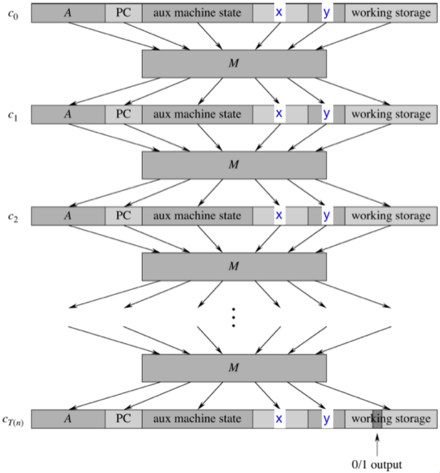
$$D(x) = \text{"}yes\text{"} \quad \text{iff} \quad \exists y, V_D(x, y) = TRUE$$

4. $\mathcal{NP}$-completeness theory

For every input x,

$$D(x) = \text{"}yes\text{"} \quad \text{iff} \quad \exists y, V_D(x, y) = TRUE$$

where verification algorithm $V_D(x, y)$ can be viewed as hardware

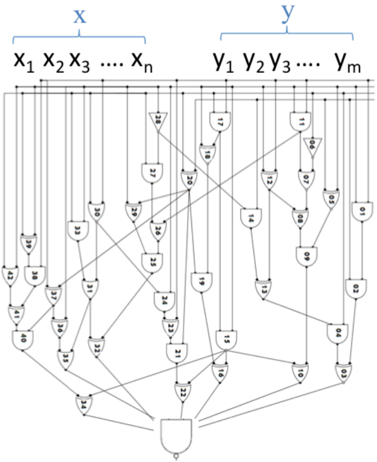


4. $\mathcal{NP}$-completeness theory

For every input x,

$$D(x) = \text{"}yes\text{"} \quad \text{iff} \quad \exists y, V_D(x, y) = TRUE$$

where verification algorithm $V_D(x, y)$ can be viewed as circuit $C(x, y)$



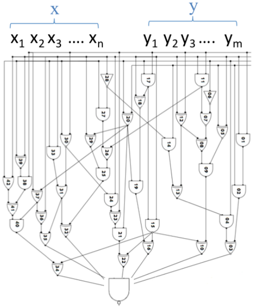
4. $\mathcal{NP}$-completeness theory

For every input x,

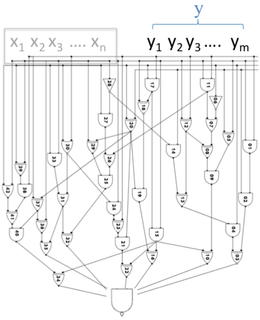
$$D(x) = \text{"}yes\text{"} \quad \text{iff} \quad \exists y, V_D(x,y) = TRUE$$

$$\text{iff} \quad \exists y, C(x,y) = TRUE$$

Turn circuit $C(x,y)$ to $C_x(y)$



$\Longrightarrow$



4. $\mathcal{NP}$-completeness theory

For every input x,

$$\begin{aligned} D(x) = \text{"}yes\text{"} \quad & \text{iff} \quad \exists y, V_D(x,y) = TRUE \\ & \text{iff} \quad \exists y, C(x,y) = TRUE \\ & \text{iff} \quad \exists y, C_x(y) = TRUE \end{aligned}$$

4. $\mathcal{NP}$-completeness theory

For every input x,

$$\begin{aligned} D(x) = \text{"}yes\text{"} \quad & \text{iff} \quad \exists y, V_D(x, y) = TRUE \\ & \text{iff} \quad \exists y, C(x, y) = TRUE \\ & \text{iff} \quad \exists y, C_x(y) = TRUE \end{aligned}$$

- since D is given, then we know V_D of two inputs x and y;

4. $\mathcal{NP}$-completeness theory

For every input x,

$$\begin{aligned} D(x) = \text{"}yes\text{"} \quad & \text{iff} \quad \exists y, V_D(x,y) = TRUE \\ & \text{iff} \quad \exists y, C(x,y) = TRUE \\ & \text{iff} \quad \exists y, C_x(y) = TRUE \end{aligned}$$

- since D is given, then we know V_D of two inputs x and y;
- thus we know C of two inputs x and y;

4. $\mathcal{NP}$-completeness theory

For every input x,

$$\begin{aligned} D(x) = \text{"}yes\text{"} \quad & \text{iff} \quad \exists y, V_D(x,y) = TRUE \\ & \text{iff} \quad \exists y, C(x,y) = TRUE \\ & \text{iff} \quad \exists y, C_x(y) = TRUE \end{aligned}$$

- since D is given, then we know V_D of two inputs x and y;
- thus we know C of two inputs x and y;
- then we know C_x of single input y;

4. $\mathcal{NP}$-completeness theory

For every input x,

$$\begin{aligned} D(x) = \text{"}yes\text{"} \quad & \text{iff} \quad \exists y, V_D(x,y) = TRUE \\ & \text{iff} \quad \exists y, C(x,y) = TRUE \\ & \text{iff} \quad \exists y, C_x(y) = TRUE \end{aligned}$$

- since D is given, then we know V_D of two inputs x and y;
- thus we know C of two inputs x and y;
- then we know C_x of single input y;
- that is, there is a mapping f: $f(x) = C_x$, such that

$$\begin{aligned} D(x) = \text{"}yes\text{"} \quad & \text{iff} \quad \exists \text{ assignment } y \text{ satisfying circuit } C_x \\ & \text{iff} \quad \text{boolean } C_x \text{ is satisfiable} \end{aligned}$$

4. $\mathcal{NP}$-completeness theory

We conclude:

Theorem: $\forall D \in \mathcal{NP}$, $D \leq_p$ **CSAT**.
That is, **CSAT** is $\mathcal{NP}$-hard.

We conclude:

Theorem: $\forall D \in \mathcal{NP}$, $D \leq_p$ **CSAT**.
That is, **CSAT** is $\mathcal{NP}$-hard.

Lemma: **CSAT** $\leq_p$ **SAT**.

We conclude:

Theorem: $\forall D \in \mathcal{NP}$, $D \leq_p$ **CSAT**.
That is, **CSAT** is $\mathcal{NP}$-hard.

Lemma: **CSAT** $\leq_p$ **SAT**.

Theorem: **SAT** is $\mathcal{NP}$-hard (by the lemma and transitivity of $\leq_p$).

4. $\mathcal{NP}$-completeness theory

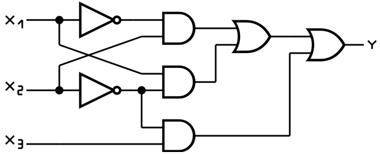
Lemma: **CSAT** $\leq_p$ **SAT**.

Proof idea: to transform a circuit C to a boolean formula ϕ_C.
(Warning: simply unfolding circuits into formulae doesn't work!)

4. $\mathcal{NP}$-completeness theory

Lemma: **CSAT** $\leq_p$ **SAT**.

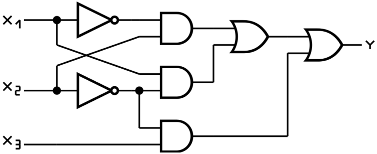
Proof idea: to transform a circuit C to a boolean formula ϕ_C.
(Warning: simply unfolding circuits into formulae doesn't work!)

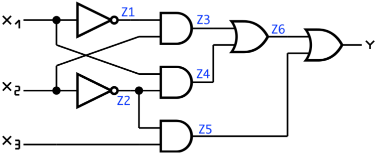
$C =$ 

4. $\mathcal{NP}$-completeness theory

Lemma: **CSAT** $\leq_p$ **SAT**.

Proof idea: to transform a circuit C to a boolean formula ϕ_C.
(Warning: simply unfolding circuits into formulae doesn't work!)

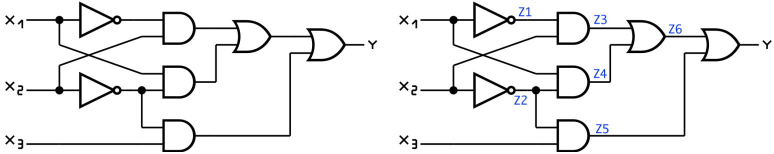
$C =$ 



4. $\mathcal{NP}$-completeness theory

Lemma: **CSAT** $\leq_p$ **SAT**.

Proof idea: to transform a circuit C to a boolean formula ϕ_C.
(Warning: simply unfolding circuits into formulae doesn't work!)

$C =$ 

A boolean formula describes the relationships of boolean "wire variables" in the circuit C:

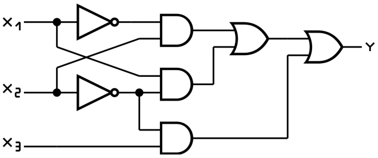
$$\phi_C = (Z_1 \leftrightarrow \neg X_1) \wedge (Z_2 \leftrightarrow \neg X_2) \wedge (Z_3 \leftrightarrow (Z_1 \wedge X_2)) \wedge (Z_4 \leftrightarrow (X_1 \wedge Z_2))$$

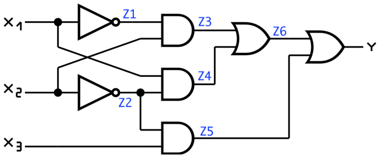
$$\wedge (Z_5 \leftrightarrow (Z_2 \wedge X_3)) \wedge (Z_6 \leftrightarrow (Z_3 \vee Z_4)) \wedge (Y \leftrightarrow (Z_6 \vee Z_5)) \wedge Y$$

4. $\mathcal{NP}$-completeness theory

Lemma: **CSAT** $\leq_p$ **SAT**.

Proof idea: to transform a circuit C to a boolean formula ϕ_C.
(Warning: simply unfolding circuits into formulae doesn't work!)

$C =$ 



A boolean formula describes the relationships of boolean "wire variables" in the circuit C:

$$\phi_C = (Z_1 \leftrightarrow \neg X_1) \wedge (Z_2 \leftrightarrow \neg X_2) \wedge (Z_3 \leftrightarrow (Z_1 \wedge X_2)) \wedge (Z_4 \leftrightarrow (X_1 \wedge Z_2))$$

$$\wedge (Z_5 \leftrightarrow (Z_2 \wedge X_3)) \wedge (Z_6 \leftrightarrow (Z_3 \vee Z_4)) \wedge (Y \leftrightarrow (Z_6 \vee Z_5)) \wedge Y$$

such that

C is satisfiable **iff** ϕ_C is satisfiable

4. $\mathcal{NP}$-completeness theory

The following facts are known:

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard;

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard; (why?)

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard; (why?) it is also $\mathcal{NP}$-complete; (why?)

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard; (why?) it is also $\mathcal{NP}$-complete; (why?)
- How about problem **Clique**?

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard; (why?) it is also $\mathcal{NP}$-complete; (why?)
- How about problem **Clique**? (it is also $\mathcal{NP}$-complete)

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard; (why?) it is also $\mathcal{NP}$-complete; (why?)
- How about problem **Clique**? (it is also $\mathcal{NP}$-complete)
- **Hamiltonian Path** is $\mathcal{NP}$-complete.

4. $\mathcal{NP}$-completeness theory

The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard; (why?) it is also $\mathcal{NP}$-complete; (why?)
- How about problem **Clique**? (it is also $\mathcal{NP}$-complete)
- **Hamiltonian Path** is $\mathcal{NP}$-complete.
  (reductions to this and other problems are very involved!)

4. $\mathcal{NP}$-completeness theory

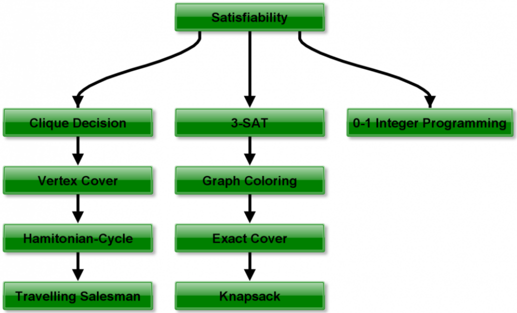
The following facts are known:

- **CSAT** and **SAT** are $\mathcal{NP}$-hard;
- **CSAT** and **SAT** are also $\mathcal{NP}$-complete (why?)
- **IS** is $\mathcal{NP}$-hard; (why?) it is also $\mathcal{NP}$-complete; (why?)
- How about problem **Clique**? (it is also $\mathcal{NP}$-complete)
- **Hamiltonian Path** is $\mathcal{NP}$-complete.
  (reductions to this and other problems are very involved!)
- $\mathcal{NP}$-completeness theory does not answer question

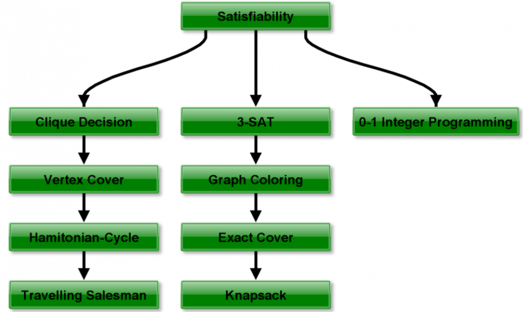
$$\mathcal{P} \ =? \ \mathcal{NP}$$

  but gives strong evidence that $\mathcal{P} \neq \mathcal{NP}$

4. $\mathcal{NP}$-completeness theory

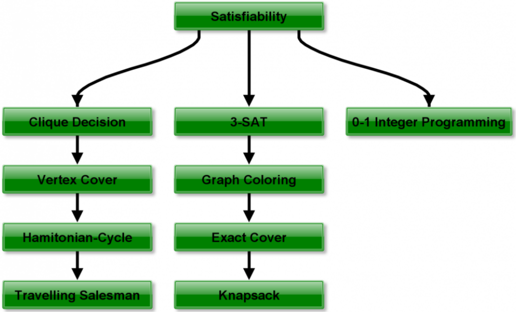


4. $\mathcal{NP}$-completeness theory



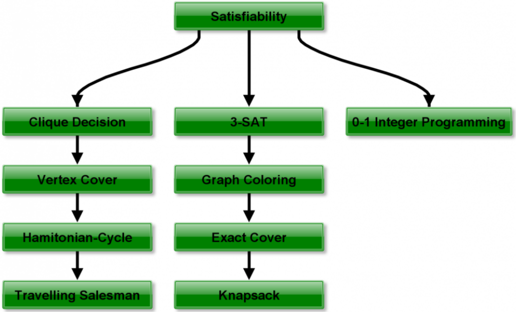
- **SAT** was the first one proved $\mathcal{NP}$-complete (Stephen Cook);

4. $\mathcal{NP}$-completeness theory



- **SAT** was the first one proved $\mathcal{NP}$-complete (Stephen Cook);
- Most of the other reduction proofs were done by Richard Karp;

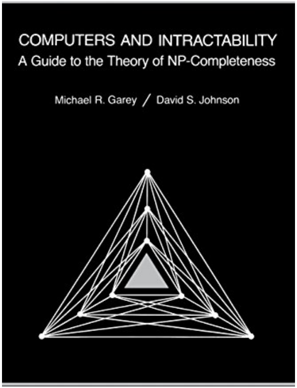
4. $\mathcal{NP}$-completeness theory

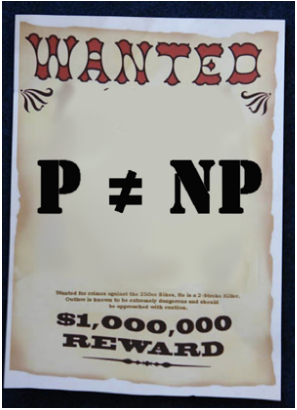


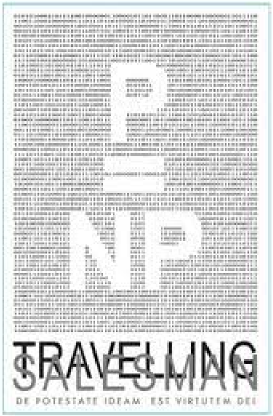
- **SAT** was the first one proved $\mathcal{NP}$-complete (Stephen Cook);
- Most of the other reduction proofs were done by Richard Karp;
- The following book contains hundreds of $\mathcal{NP}$-complete and proofs;

  Garey and Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company (1979).

4. $\mathcal{NP}$-completeness theory







Review

Scope of topics in Quiz#9

Review

Scope of topics in Quiz#9

- decision vs verification;

Review

Scope of topics in Quiz#9

- decision vs verification;
- reduction $\leq$, polynomial-time reduction $\leq_p$; implications of $\leq_p$;

Review

Scope of topics in Quiz#9

- decision vs verification;
- reduction $\leq$, polynomial-time reduction $\leq_p$;
  implications of $\leq_p$;
- definitions of NP-hardness, NP-completeness
  implications of NP-hardness, NP-completeness

Review

Scope of topics in Quiz#9

- decision vs verification;
- reduction $\leq$, polynomial-time reduction $\leq_p$;
  implications of $\leq_p$;
- definitions of NP-hardness, NP-completeness
  implications of NP-hardness, NP-completeness
- known NP-complete, NP-hard problems, and why they are.