CSE 431/531: Analysis of Algorithms

NP-Completeness

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

Q: Why do we study negative results?

NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

Q: Why do we study negative results?

- A given problem X cannot be solved in polynomial time.

NP-Completeness Theory

- The topics we discussed so far are positive results: how to design efficient algorithms for solving a given problem.
- NP-Completeness provides negative results: some problems can not be solved efficiently.

Q: Why do we study negative results?

- A given problem X cannot be solved in polynomial time.
- Without knowing it, you will have to keep trying to find polynomial time algorithm for solving X. All our efforts are doomed!

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Reason for Efficient = Polynomial Time

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5} \log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then k is small, say 4

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then k is small, say 4
- A good cut separating problems: for most natural problems, either we have a polynomial time algorithm, or the best algorithm runs in time $\Omega(2^{n^c})$ for some c

Efficient = Polynomial Time

- Polynomial time: $O(n^k)$ for any constant $k > 0$
- Example: $O(n), O(n^2), O(n^{2.5}\log n), O(n^{100})$
- Not polynomial time: $O(2^n), O(n^{\log n})$
- Almost all algorithms we learnt so far run in polynomial time

Reason for Efficient = Polynomial Time

- For natural problems, if there is an $O(n^k)$-time algorithm, then k is small, say 4
- A good cut separating problems: for most natural problems, either we have a polynomial time algorithm, or the best algorithm runs in time $\Omega(2^{n^c})$ for some c
- Do not need to worry about the computational model

Pseudo-Polynomial Is not Polynomial!

Polynomial:

- Kruskal's algorithm for minimum spanning tree: $O(n \lg n + m)$
- Floyd-Warshall for all-pair shortest paths: $O(n^3)$

Reason: we need to specify $m \geq n - 1$ edges in the input

Pseudo-Polynomial Is not Polynomial!

Polynomial:

- Kruskal's algorithm for minimum spanning tree: $O(n \lg n + m)$
- Floyd-Warshall for all-pair shortest paths: $O(n^3)$

Reason: we need to specify $m \geq n - 1$ edges in the input

Pseudo-Polynomial:

- Knapsack Problem: $O(nW)$, where W is the maximum weight the Knapsack can hold

Reason: to specify integer in $[0, W]$, we only need $O(\lg W)$ bits.

Outline

Recall: Knapsack Problem

Input: n items, each item i with a weight w_i, and a value v_i; a bound W on the total weight the knapsack can hold

Output: the maximum value of items the knapsack can hold, i.e, a set $S \subseteq \{1, 2, \cdots, n\}$:

$$\max \sum_{i \in S} v_i \qquad \text{s.t.} \sum_{i \in S} w_i \leq W$$

- DP is $O(nW)$-time algorithm, not a real polynomial time
- Knapsack is NP-hard: it is unlikely that the problem can be solved in polynomial time

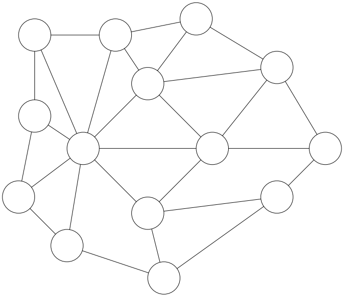
Example: Hamiltonian Cycle Problem

Def. Let G be an undirected graph. A Hamiltonian Cycle (HC) of G is a cycle C in G that passes each vertex of G exactly once.

Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle



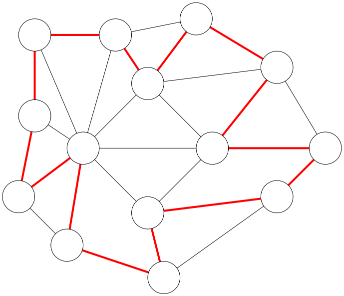
Example: Hamiltonian Cycle Problem

Def. Let G be an undirected graph. A Hamiltonian Cycle (HC) of G is a cycle C in G that passes each vertex of G exactly once.

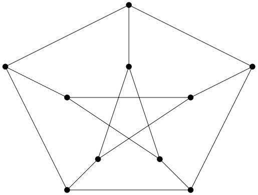
Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle



Example: Hamiltonian Cycle Problem



- The graph is called the Petersen Graph. It has no HC.

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time

Example: Hamiltonian Cycle Problem

Hamiltonian Cycle (HC) Problem

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

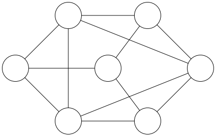
Output: whether G contains a Hamiltonian cycle

Algorithm for Hamiltonian Cycle Problem:

- Enumerate all possible permutations, and check if it corresponds to a Hamiltonian Cycle
- Running time: $O(n!m) = 2^{O(n \lg n)}$
- Better algorithm: $2^{O(n)}$
- Far away from polynomial time
- HC is NP-hard: it is unlikely that it can be solved in polynomial time.

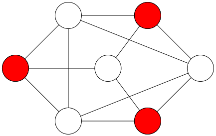
Maximum Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



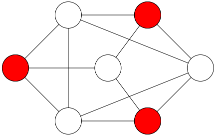
Maximum Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Maximum Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



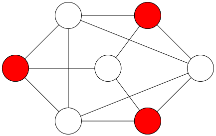
Maximum Independent Set Problem

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

Output: the size of the maximum independent set of G

Maximum Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Maximum Independent Set Problem

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

Output: the size of the maximum independent set of G

- Maximum Independent Set is NP-hard

Formula Satisfiability

Formula Satisfiability

Input: boolean formula with n variables, with $\vee, \wedge, \neg$ operators.

Output: whether the boolean formula is satisfiable

- Example: $\neg((\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3))$ is not satisfiable
- Trivial algorithm: enumerate all possible assignments, and check if each assignment satisfies the formula

Formula Satisfiability

Formula Satisfiability

Input: boolean formula with n variables, with $\vee, \wedge, \neg$ operators.

Output: whether the boolean formula is satisfiable

- Example: $\neg((\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3))$ is not satisfiable
- Trivial algorithm: enumerate all possible assignments, and check if each assignment satisfies the formula
- Formula Satisfiablity is NP-hard

Outline

Decision Problem Vs Optimization Problem

Def. A problem X is called a decision problem if the output is either 0 or 1 (yes/no).

Decision Problem Vs Optimization Problem

Def. A problem X is called a decision problem if the output is either 0 or 1 (yes/no).

- When we define the P and NP, we only consider decision problems.

Decision Problem Vs Optimization Problem

Def. A problem X is called a decision problem if the output is either 0 or 1 (yes/no).

- When we define the P and NP, we only consider decision problems.

Fact For each optimization problem X, there is a decision version X' of the problem. If we have a polynomial time algorithm for the decision version X', we can solve the original problem X in polynomial time.

Optimization to Decision

Shortest Path

Input: graph $G = (V, E)$, weight w, s, t and a bound L

Output: whether there is a path from s to t of length at most L

Optimization to Decision

Shortest Path

Input: graph $G = (V, E)$, weight w, s, t and a bound L

Output: whether there is a path from s to t of length at most L

Maximum Independent Set

Input: a graph G and a bound k

Output: whether there is an independent set of size at least k

Encoding

The input of a problem will be encoded as a binary string.

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String:

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 111101

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 11110111110001

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 1111011111100011111000011000001

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 111101111100011111000011000001 1100001101

Encoding

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 111101111100011111000011000001 1100001101111111111000001

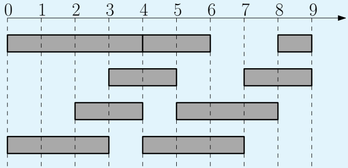
Encoding

The input of an problem will be encoded as a binary string.

Encoding

The input of an problem will be encoded as a binary string.

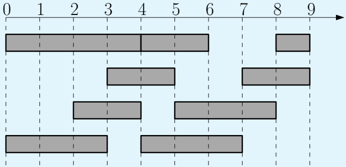
Example: Interval Scheduling Problem



Encoding

The input of an problem will be encoded as a binary string.

Example: Interval Scheduling Problem

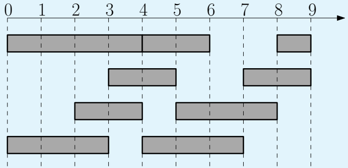


- $(0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)$

Encoding

The input of an problem will be encoded as a binary string.

Example: Interval Scheduling Problem



- $(0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)$
- Encode the sequence into a binary string as before

Encoding

Def. The size of an input is the length of the encoded string s for the input, denoted as $|s|$.

Q: Does it matter how we encode the input instances?

Encoding

Def. The size of an input is the length of the encoded string s for the input, denoted as $|s|$.

Q: Does it matter how we encode the input instances?

A: No! As long as we are using a "natural" encoding. We only care whether the running time is polynomial or not

Define Problem as a Set

Def. A decision problem X is the set of strings on which the output is yes. i.e, $s \in X$ if and only if the correct output for the input s is 1 (yes).

Define Problem as a Set

Def. A decision problem X is the set of strings on which the output is yes. i.e, $s \in X$ if and only if the correct output for the input s is 1 (yes).

Def. An algorithm A solves a problem X if, $A(s) = 1$ if and only if $s \in X$.

Define Problem as a Set

Def. A decision problem X is the set of strings on which the output is yes. i.e, $s \in X$ if and only if the correct output for the input s is 1 (yes).

Def. An algorithm A solves a problem X if, $A(s) = 1$ if and only if $s \in X$.

Def. A has a polynomial running time if there is a polynomial function $p(\cdot)$ so that for every string s, the algorithm A terminates on s in at most $p(|s|)$ steps.

Complexity Class P

Def. The complexity class P is the set of decision problems X that can be solved in polynomial time.

Complexity Class P

Def. The complexity class P is the set of decision problems X that can be solved in polynomial time.

- The decision versions of interval scheduling, shortest path and minimum spanning tree all in P.

Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC

Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that G contains a Hamiltonian cycle?

Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that G contains a Hamiltonian cycle?

A: Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of G

Certifier for Hamiltonian Cycle (HC)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that G contains a Hamiltonian cycle?

A: Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of G

Def. The message Alice sends to Bob is called a certificate, and the algorithm Bob runs is called a certifier.

Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given graph $G = (V, E)$ and integer k, such that there is an independent set of size k in G, how can Alice convince Bob that there is such a set?

Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given graph $G = (V, E)$ and integer k, such that there is an independent set of size k in G, how can Alice convince Bob that there is such a set?

A: Alice gives a set of size k to Bob and Bob checks if it is really a independent set in G.

Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given graph $G = (V, E)$ and integer k, such that there is an independent set of size k in G, how can Alice convince Bob that there is such a set?

A: Alice gives a set of size k to Bob and Bob checks if it is really a independent set in G.

- Certificate: a set of size k

Certifier for Independent Set (Ind-Set)

- Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for Ind-Set
- Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given graph $G = (V, E)$ and integer k, such that there is an independent set of size k in G, how can Alice convince Bob that there is such a set?

A: Alice gives a set of size k to Bob and Bob checks if it is really a independent set in G.

- Certificate: a set of size k
- Certifier: check if the given set is really an independent set

Graph Isomorphism

Graph Isomorphism

Input: two graphs G_1 and G_2,

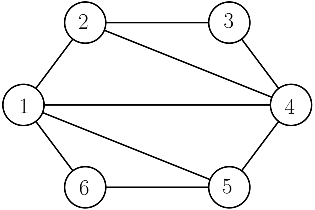
Output: whether two graphs are isomorphic to each other

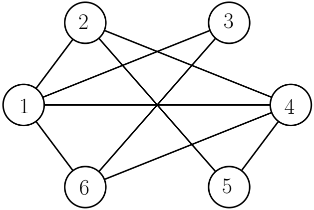
Graph Isomorphism

Graph Isomorphism

Input: two graphs G_1 and G_2,

Output: whether two graphs are isomorphic to each other



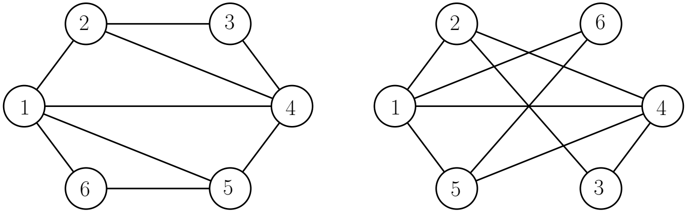


Graph Isomorphism

Graph Isomorphism

Input: two graphs G_1 and G_2,

Output: whether two graphs are isomorphic to each other

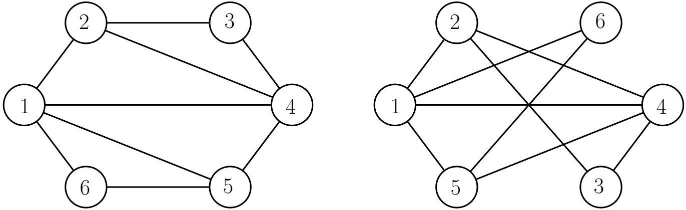


Graph Isomorphism

Graph Isomorphism

Input: two graphs G_1 and G_2,

Output: whether two graphs are isomorphic to each other



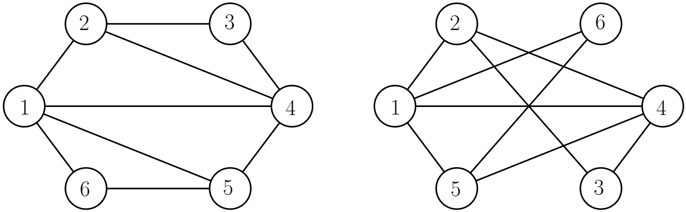
- What is the certificate?

Graph Isomorphism

Graph Isomorphism

Input: two graphs G_1 and G_2,

Output: whether two graphs are isomorphic to each other



- What is the certificate?
- What is the certifier?

The Complexity Class NP

Def. B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t
- there is a polynomial function p such that, $s \in X$ if and only if there is string t such that $|t| \leq p(|s|)$ and $B(s,t) = 1$.

The string t such that $B(s,t) = 1$ is called a certificate.

The Complexity Class NP

Def. B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t
- there is a polynomial function p such that, $s \in X$ if and only if there is string t such that $|t| \leq p(|s|)$ and $B(s,t) = 1$.

The string t such that $B(s,t) = 1$ is called a certificate.

Def. The complexity class NP is the set of all problems for which there exists an efficient certifier.

Hamiltonian Cycle $\in$ NP

- Input: Graph G

Hamiltonian Cycle $\in$ NP

- Input: Graph G
- Certificate: a sequence S of edges in G

Hamiltonian Cycle $\in$ NP

- Input: Graph G
- Certificate: a sequence S of edges in G
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function p

Hamiltonian Cycle $\in$ NP

- Input: Graph G
- Certificate: a sequence S of edges in G
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function p
- Certifier B: $B(G, S) = 1$ if and only if S is an HC in G

Hamiltonian Cycle $\in$ NP

- Input: Graph G
- Certificate: a sequence S of edges in G
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function p
- Certifier B: $B(G, S) = 1$ if and only if S is an HC in G
- Clearly, B runs in polynomial time

Hamiltonian Cycle $\in$ NP

- Input: Graph G
- Certificate: a sequence S of edges in G
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function p
- Certifier B: $B(G, S) = 1$ if and only if S is an HC in G
- Clearly, B runs in polynomial time
- $G \in \text{HC} \qquad \Longleftrightarrow \qquad \exists S,\ B(G, S) = 1$

Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on V

Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on V
- Certificate: a 1-1 function $f : V \to V$

Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on V
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function p

Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on V
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function p
- Certifier B: $B((G_1, G_2), f) = 1$ if and only if for every $u, v \in V$, we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.

Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on V
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function p
- Certifier B: $B((G_1, G_2), f) = 1$ if and only if for every $u, v \in V$, we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.
- Clearly, B runs in polynomial time

Graph Isomorphism $\in$ NP

- Input: two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on V
- Certificate: a 1-1 function $f : V \to V$
- $|\text{encoding}(f)| \leq p(|\text{encoding}(G_1, G_2)|)$ for some polynomial function p
- Certifier B: $B((G_1, G_2), f) = 1$ if and only if for every $u, v \in V$, we have $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$.
- Clearly, B runs in polynomial time
- $(G_1, G_2) \in \text{GI} \quad \Longleftrightarrow \quad \exists f,\ B((G_1, G_2), f) = 1$

Maximum Independent Set $\in$ NP

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

Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer k
- Certificate: a set $S \subseteq V$ of size k

Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer k
- Certificate: a set $S \subseteq V$ of size k
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function p

Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer k
- Certificate: a set $S \subseteq V$ of size k
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function p
- Certifier B: $B((G, k), S) = 1$ if and only if S is an independent set in G

Maximum Independent Set $\in$ NP

- Input: graph $G = (V, E)$ and integer k
- Certificate: a set $S \subseteq V$ of size k
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function p
- Certifier B: $B((G, k), S) = 1$ if and only if S is an independent set in G
- Clearly, B runs in polynomial time

Maximum Independent Set $\in$ NP

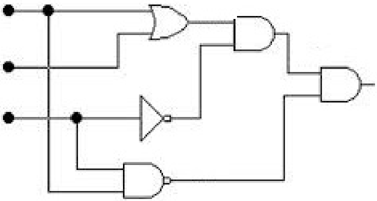
- Input: graph $G = (V, E)$ and integer k
- Certificate: a set $S \subseteq V$ of size k
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G, k)|)$ for some polynomial function p
- Certifier B: $B((G, k), S) = 1$ if and only if S is an independent set in G
- Clearly, B runs in polynomial time

- $(G, k) \in \text{MIS} \quad \Longleftrightarrow \quad \exists S,\ B((G, k), S) = 1$

Circuit Satisfiablity (Circuit-Sat) Problem

Input: a circuit with and/or/not gates

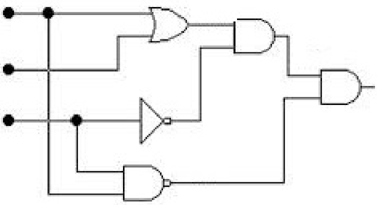
Output: whether there is an assignment such that the output is 1?



Circuit Satisfiablity (Circuit-Sat) Problem

Input: a circuit with and/or/not gates

Output: whether there is an assignment such that the output is 1?



- Is Circuit-Sat $\in$ NP?

$\overline{\text{HC}}$

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

Output: whether G does not contain a Hamiltonian cycle

$\overline{\mathrm{HC}}$

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

Output: whether G does not contain a Hamiltonian cycle

- Is $\overline{\mathrm{HC}} \in \mathrm{NP}$?

$\overline{\text{HC}}$

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

Output: whether G does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?
- Can Alice convince Bob that G is a yes-instance (i.e, G does not contain a HC), if this is true.

$\overline{\text{HC}}$

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

Output: whether G does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?
- Can Alice convince Bob that G is a yes-instance (i.e, G does not contain a HC), if this is true.
- Unlikely

$\overline{\text{HC}}$

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

Output: whether G does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?
- Can Alice convince Bob that G is a yes-instance (i.e, G does not contain a HC), if this is true.
- Unlikely
- Alice can only convince Bob that G is a no-instance

$\overline{\text{HC}}$

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

Output: whether G does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in \text{NP}$?
- Can Alice convince Bob that G is a yes-instance (i.e, G does not contain a HC), if this is true.
- Unlikely
- Alice can only convince Bob that G is a no-instance
- $\overline{\text{HC}} \in \text{Co-NP}$

The Complexity Class Co-NP

Def. For a problem X, the problem $\overline{X}$ is the problem such that $s \in \overline{X}$ if and only if $s \notin X$.

Def. Co-NP is the set of decision problems X such that $\overline{X} \in$ NP.

Def. A tautology is a boolean formula that always evaluates to 1.

Tautology Problem

Input: a boolean formula

Output: whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology

Def. A tautology is a boolean formula that always evaluates to 1.

Tautology Problem

Input: a boolean formula

Output: whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology

Def. A tautology is a boolean formula that always evaluates to 1.

Tautology Problem

Input: a boolean formula

Output: whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology $\in$ Co-NP

Def. A tautology is a boolean formula that always evaluates to 1.

Tautology Problem

Input: a boolean formula

Output: whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology $\in$ Co-NP
- Indeed, Tautology $= \overline{\text{Formula-Unsat}}$

Prime

Prime

Input: an integer $q \geq 2$

Output: whether q is a prime

Prime

Prime

Input: an integer $q \geq 2$

Output: whether q is a prime

- It is easy to certify that q is not a prime

Prime

Prime

Input: an integer $q \geq 2$

Output: whether q is a prime

- It is easy to certify that q is not a prime
- Prime $\in$ Co-NP

Prime

Prime

Input: an integer $q \geq 2$

Output: whether q is a prime

- It is easy to certify that q is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP

Prime

Prime

Input: an integer $q \geq 2$

Output: whether q is a prime

- It is easy to certify that q is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP
- P $\subseteq$ NP $\cap$ Co-NP (see soon)

Prime

Prime

Input: an integer $q \geq 2$

Output: whether q is a prime

- It is easy to certify that q is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP
- P $\subseteq$ NP $\cap$ Co-NP (see soon)
- If a natural problem X is in NP $\cap$ Co-NP, then it is likely that $X \in P$

Prime

Prime

Input: an integer $q \geq 2$

Output: whether q is a prime

- It is easy to certify that q is not a prime
- Prime $\in$ Co-NP
- [Pratt 1970] Prime $\in$ NP
- P $\subseteq$ NP $\cap$ Co-NP (see soon)
- If a natural problem X is in NP $\cap$ Co-NP, then it is likely that $X \in P$
- [AKS 2002] Prime $\in$ P

$P \subseteq NP$

P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

Q: How can Alice convince Bob that s is a yes instance?

P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

Q: How can Alice convince Bob that s is a yes instance?

A: Since $X \in$ P, Bob can check whether $s \in X$ by himself, without Alice's help.

P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

Q: How can Alice convince Bob that s is a yes instance?

A: Since $X \in$ P, Bob can check whether $s \in X$ by himself, without Alice's help.

- The certificate is an empty string

P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

Q: How can Alice convince Bob that s is a yes instance?

A: Since $X \in$ P, Bob can check whether $s \in X$ by himself, without Alice's help.

- The certificate is an empty string
- Thus, $X \in$ NP and P $\subseteq$ NP

P $\subseteq$ NP

- Let $X \in$ P and $s \in X$

Q: How can Alice convince Bob that s is a yes instance?

A: Since $X \in$ P, Bob can check whether $s \in X$ by himself, without Alice's help.

- The certificate is an empty string
- Thus, $X \in$ NP and P $\subseteq$ NP
- Similarly, P $\subseteq$ Co-NP, thus P $\subseteq$ NP $\cap$ Co-NP

Is P = NP?

Is P = NP?

- A famous, big, and fundamental open problem in computer science

Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made

Is P $=$ NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- General belief is P $\neq$ NP

Is P $=$ NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- General belief is P $\neq$ NP
- It would be too amazing if P $=$ NP: if one can check a solution efficiently, then one can find a solution efficiently

Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- General belief is P $\neq$ NP
- It would be too amazing if P = NP: if one can check a solution efficiently, then one can find a solution efficiently
- Complexity assumption: P $\neq$ NP

Is P $=$ NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- General belief is P $\neq$ NP
- It would be too amazing if P $=$ NP: if one can check a solution efficiently, then one can find a solution efficiently
- Complexity assumption: P $\neq$ NP
- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:

Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- General belief is P $\neq$ NP
- It would be too amazing if P = NP: if one can check a solution efficiently, then one can find a solution efficiently

- Complexity assumption: P $\neq$ NP
- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:
  - if P $\neq$ NP, then HC $\notin$ P

Is P = NP?

- A famous, big, and fundamental open problem in computer science
- Little progress has been made
- General belief is P $\neq$ NP
- It would be too amazing if P $=$ NP: if one can check a solution efficiently, then one can find a solution efficiently
- Complexity assumption: P $\neq$ NP
- We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:
  - if P $\neq$ NP, then HC $\notin$ P
  - HC $\notin$ P, unless P $=$ NP

Is NP = Co-NP?

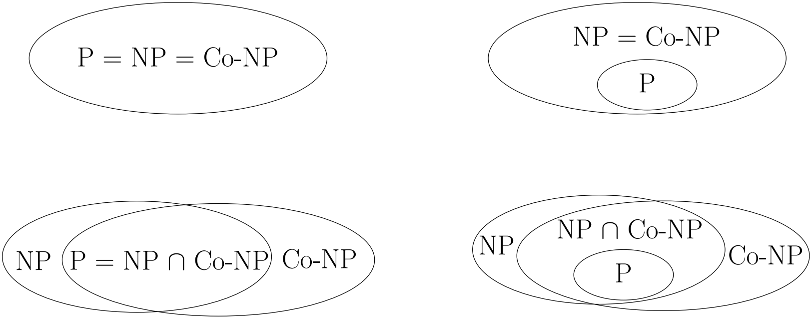
- Again, a big open problem

Is NP = Co-NP?

- Again, a big open problem
- General belief: NP $\neq$ Co-NP.

4 Possibilities of Relationships

Notice that $X \in \text{NP} \iff \overline{X} \in \text{Co-NP}$ and $\text{P} \subseteq \text{NP} \cap \text{Co-NP}$



- General belief: we are in the 4th scenario

Outline

Polynomial-Time Reducations

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

Polynomial-Time Reducations

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

To prove positive results:

Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.

Polynomial-Time Reducations

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

To prove positive results:

Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.

To prove negative results:

Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G

Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G

Lemma HP $\leq_{\mathsf{P}}$ HC.

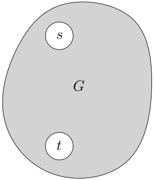
Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G

Lemma HP $\leq_P$ HC.



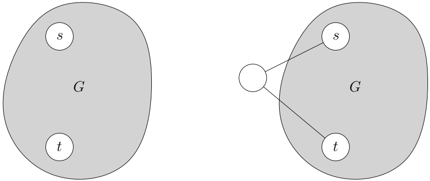
Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G

Lemma HP $\leq_{\mathsf{P}}$ HC.



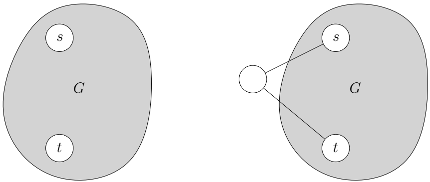
Polynomial-Time Reduction: Example

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G

Lemma HP $\leq_P$ HC.



Obs. G has a HP from s to t if and only if graph on right side has a HC.

NP-Completeness

Def. A problem X is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_\text{P} X$ for every $Y \in \text{NP}$.

NP-Completeness

Def. A problem X is called NP-hard if

2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathrm{NP}$.

NP-Completeness

Def. A problem X is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_{\text{P}} X$ for every $Y \in \text{NP}$.

NP-Completeness

Def. A problem X is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

Theorem If X is NP-complete and $X \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.

NP-Completeness

Def. A problem X is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_{\text{P}} X$ for every $Y \in \text{NP}$.

Theorem If X is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP

NP-Completeness

Def. A problem X is called NP-complete if

1. $X \in \mathrm{NP}$, and
2. $Y \leq_{\mathrm{P}} X$ for every $Y \in \mathrm{NP}$.

Theorem If X is NP-complete and $X \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)

NP-Completeness

Def. A problem X is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_\text{P} X$ for every $Y \in \text{NP}$.

Theorem If X is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)
- To prove $\text{P} = \text{NP}$ (if you believe it), you only need to give an efficient algorithm for any NP-complete problem

NP-Completeness

Def. A problem X is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_{\text{P}} X$ for every $Y \in \text{NP}$.

Theorem If X is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)
- To prove $\text{P} = \text{NP}$ (if you believe it), you only need to give an efficient algorithm for any NP-complete problem
- If you believe $\text{P} \neq \text{NP}$, and proved that a problem X is NP-complete (or NP-hard), stop trying to design efficient algorithms for X

Outline

Def. A problem X is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

Def. A problem X is called NP-complete if

1. $X \in \mathsf{NP}$, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in \mathsf{NP}$.

- How can we find a problem $X \in \mathsf{NP}$ such that every problem $Y \in \mathsf{NP}$ is polynomial time reducible to X? Are we asking for too much?

Def. A problem X is called NP-complete if

1. $X \in \mathrm{NP}$, and
2. $Y \leq_{\mathrm{P}} X$ for every $Y \in \mathrm{NP}$.

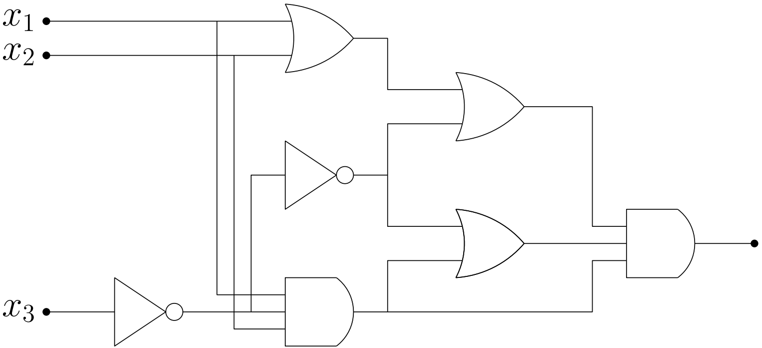
- How can we find a problem $X \in \mathrm{NP}$ such that every problem $Y \in \mathrm{NP}$ is polynomial time reducible to X? Are we asking for too much?
- No! There is indeed a large family of natural NP-complete problems

The First NP-Complete Problem: Circuit-Sat

Circuit Satisfiability (Circuit-Sat)

Input: a circuit

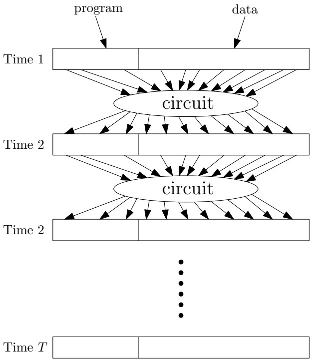
Output: whether the circuit is satisfiable



Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

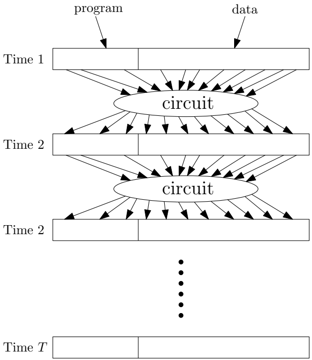
Fact Any algorithm that takes n bits as input and outputs $0/1$ with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.



Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

Fact Any algorithm that takes n bits as input and outputs $0/1$ with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.

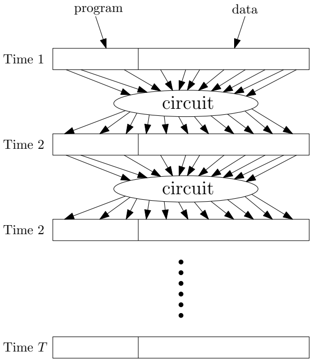


- Then, we can show that any problem $Y \in$ NP can be reduced to Circuit-Sat.

Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

Fact Any algorithm that takes n bits as input and outputs $0/1$ with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.



- Then, we can show that any problem $Y \in$ NP can be reduced to Circuit-Sat.
- We prove HC $\leq_P$ Circuit-Sat as an example.

HC $\leq_P$ Circuit-Sat

check-HC(G, S)

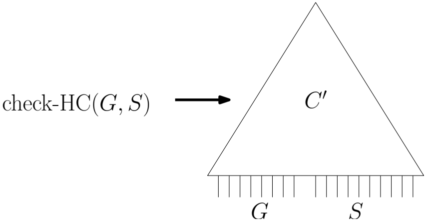
- Let check-HC(G, S) be the certifier for the Hamiltonian cycle problem: check-HC(G, S) returns 1 if S is a Hamiltonian cycle is G and 0 otherwise.

HC $\leq_P$ Circuit-Sat

check-HC(G, S)

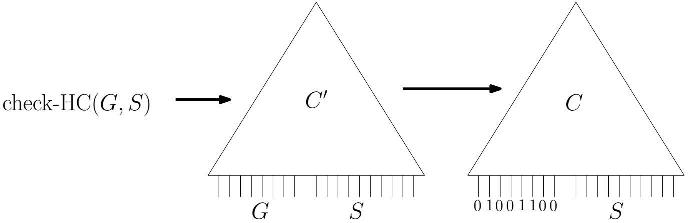
- Let check-HC(G, S) be the certifier for the Hamiltonian cycle problem: check-HC(G, S) returns 1 if S is a Hamiltonian cycle is G and 0 otherwise.
- G is a yes-instance if and only if there is an S such that check-HC(G, S) returns 1

HC $\leq_P$ Circuit-Sat



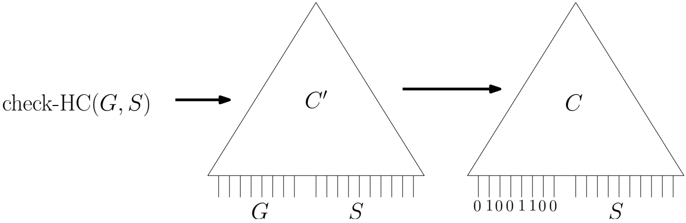
- Let check-HC(G, S) be the certifier for the Hamiltonian cycle problem: check-HC(G, S) returns 1 if S is a Hamiltonian cycle is G and 0 otherwise.
- G is a yes-instance if and only if there is an S such that check-HC(G, S) returns 1
- Construct a circuit C' for the algorithm check-HC

HC $\leq_P$ Circuit-Sat



- Let check-HC(G, S) be the certifier for the Hamiltonian cycle problem: check-HC(G, S) returns 1 if S is a Hamiltonian cycle is G and 0 otherwise.
- G is a yes-instance if and only if there is an S such that check-HC(G, S) returns 1
- Construct a circuit C' for the algorithm check-HC
- hard-wire the instance G to the circuit C' to obtain the circuit C

HC $\leq_P$ Circuit-Sat



- Let check-HC(G, S) be the certifier for the Hamiltonian cycle problem: check-HC(G, S) returns 1 if S is a Hamiltonian cycle is G and 0 otherwise.
- G is a yes-instance if and only if there is an S such that check-HC(G, S) returns 1
- Construct a circuit C' for the algorithm check-HC
- hard-wire the instance G to the circuit C' to obtain the circuit C
- G is a yes-instance if and only if C is satisfiable

$Y \leq_P$ Circuit-Sat, For Every $Y \in$NP

- Let check-Y(s, t) be the certifier for problem Y: check-Y(s, t) returns 1 if t is a valid certificate for s.
- s is a yes-instance if and only if there is a t such that check-Y(s, t) returns 1

- Construct a circuit C' for the algorithm check-Y
- hard-wire the instance s to the circuit C' to obtain the circuit C
- s is a yes-instance if and only if C is satisfiable $\square$

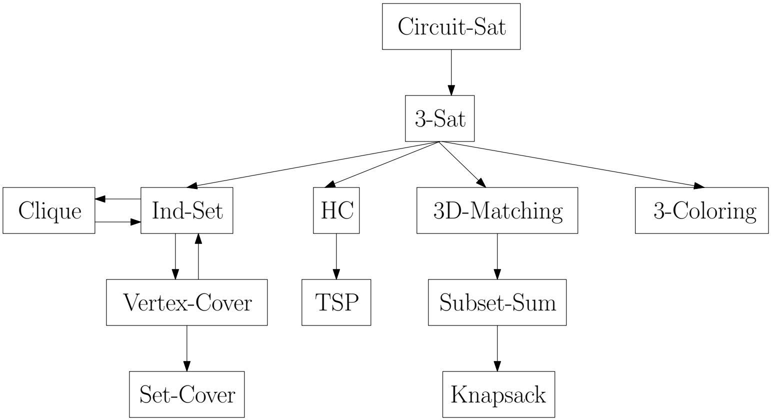
$Y \leq_P$ Circuit-Sat, For Every $Y \in$NP

- Let check-Y(s,t) be the certifier for problem Y: check-Y(s,t) returns 1 if t is a valid certificate for s.
- s is a yes-instance if and only if there is a t such that check-Y(s,t) returns 1

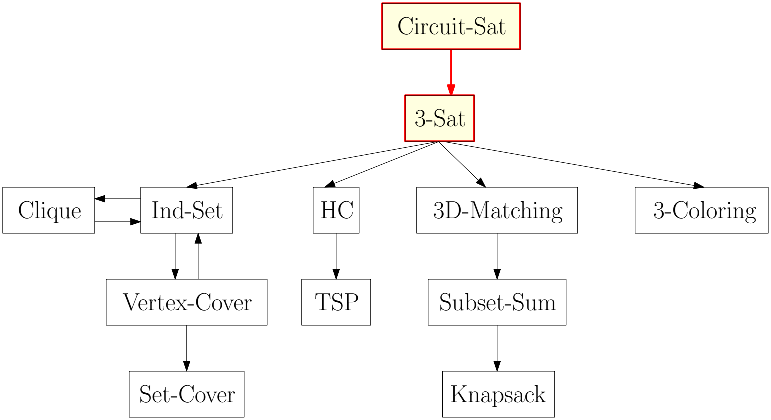
- Construct a circuit C' for the algorithm check-Y
- hard-wire the instance s to the circuit C' to obtain the circuit C
- s is a yes-instance if and only if C is satisfiable $\square$

Theorem Circuit-Sat is NP-complete.

Reductions of NP-Complete Problems



Reductions of NP-Complete Problems



3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- Boolean variables: $x_1, x_2, \cdots, x_n$

3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- Boolean variables: $x_1, x_2, \cdots, x_n$
- Literals: x_i or $\neg x_i$

3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- Boolean variables: $x_1, x_2, \cdots, x_n$
- Literals: x_i or $\neg x_i$
- Clause: disjunction ("or") of at most 3 literals: $x_3 \vee \neg x_4$, $x_1 \vee x_8 \vee \neg x_9$, $\neg x_2 \vee \neg x_5 \vee x_7$

3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- Boolean variables: $x_1, x_2, \cdots, x_n$
- Literals: x_i or $\neg x_i$
- Clause: disjunction ("or") of at most 3 literals: $x_3 \vee \neg x_4$, $x_1 \vee x_8 \vee \neg x_9$, $\neg x_2 \vee \neg x_5 \vee x_7$
- 3-CNF formula: conjunction ("and") of clauses: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

3-Sat

3-Sat

Input:	a 3-CNF formula
Output:	whether the 3-CNF is satisfiable

3-Sat

3-Sat

Input: a 3-CNF formula

Output: whether the 3-CNF is satisfiable

- To satisfy a 3-CNF, we need to satisfy all clauses

3-Sat

3-Sat

Input: a 3-CNF formula

Output: whether the 3-CNF is satisfiable

- To satisfy a 3-CNF, we need to satisfy all clauses
- To satisfy a clause, we need to satisfy at least 1 literal

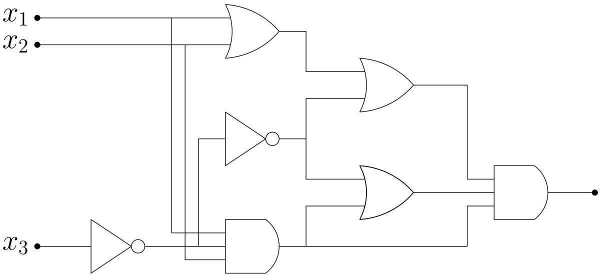
3-Sat

3-Sat

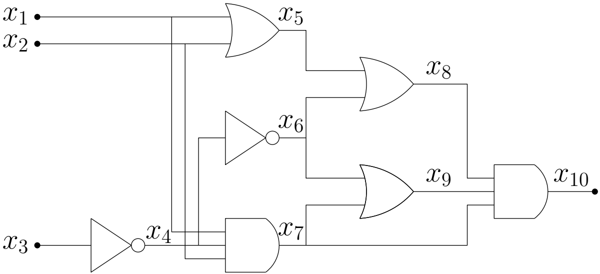
Input: a 3-CNF formula

Output: whether the 3-CNF is satisfiable

- To satisfy a 3-CNF, we need to satisfy all clauses
- To satisfy a clause, we need to satisfy at least 1 literal
- Assignment $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$ satisfies $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

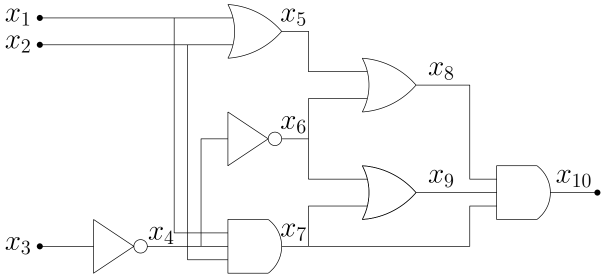

x_1
x_2
x_3

Circuit-Sat $\leq_P$ 3-Sat



- Associate every wire with a new variable

Circuit-Sat $\leq_P$ 3-Sat



- Associate every wire with a new variable
- The circuit is equivalent to the following formula:

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Circuit-Sat $\leq_P$ 3-Sat

$$
\begin{aligned}
&(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4) \\
&\wedge (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6) \\
&\wedge (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}
\end{aligned}
$$

Convert each clause to a 3-CNF

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge\ (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge\ (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge\, (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge\, (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \land (x_5 = x_1 \lor x_2) \land (x_6 = \neg x_4)$$
$$\land\, (x_7 = x_1 \land x_2 \land x_4) \land (x_8 = x_5 \lor x_6)$$
$$\land\, (x_9 = x_6 \lor x_9) \land (x_{10} = x_8 \land x_9 \land x_7) \land x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \lor x_2 \quad \Leftrightarrow$$

$$(x_1 \lor x_2 \lor \neg x_5) \quad \land$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \lor x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge\ (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge\ (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$$

$$(x_1 \vee x_2 \vee \neg x_5) \quad \wedge$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge\, (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge\, (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Convert each clause to a 3-CNF

$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$

$(x_1 \vee x_2 \vee \neg x_5) \quad \wedge$

$(x_1 \vee \neg x_2 \vee x_5) \quad \wedge$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge\, (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge\, (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$$

$$(x_1 \vee x_2 \vee \neg x_5) \quad \wedge$$

$$(x_1 \vee \neg x_2 \vee x_5) \quad \wedge$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge\ (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge\ (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$$
$$(x_1 \vee x_2 \vee \neg x_5) \quad \wedge$$
$$(x_1 \vee \neg x_2 \vee x_5) \quad \wedge$$
$$(\neg x_1 \vee x_2 \vee x_5) \quad \wedge$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$
\begin{aligned}
&(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4) \\
&\wedge (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6) \\
&\wedge (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}
\end{aligned}
$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$$

$$(x_1 \vee x_2 \vee \neg x_5) \quad \wedge$$

$$(x_1 \vee \neg x_2 \vee x_5) \quad \wedge$$

$$(\neg x_1 \vee x_2 \vee x_5) \quad \wedge$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \wedge (x_5 = x_1 \vee x_2) \wedge (x_6 = \neg x_4)$$
$$\wedge\, (x_7 = x_1 \wedge x_2 \wedge x_4) \wedge (x_8 = x_5 \vee x_6)$$
$$\wedge\, (x_9 = x_6 \vee x_9) \wedge (x_{10} = x_8 \wedge x_9 \wedge x_7) \wedge x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \vee x_2 \quad \Leftrightarrow$$
$$(x_1 \vee x_2 \vee \neg x_5) \quad \wedge$$
$$(x_1 \vee \neg x_2 \vee x_5) \quad \wedge$$
$$(\neg x_1 \vee x_2 \vee x_5) \quad \wedge$$
$$(\neg x_1 \vee \neg x_2 \vee x_5)$$

x_1	x_2	x_5	$x_5 \leftrightarrow x_1 \vee x_2$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Circuit-Sat $\leq_P$ 3-Sat

- Circuit $\iff$ Formula $\iff$ 3-CNF

Circuit-Sat $\leq_P$ 3-Sat

- Circuit $\Longleftrightarrow$ Formula $\Longleftrightarrow$ 3-CNF
- The circuit is satisfiable if and only if the 3-CNF is satisfiable

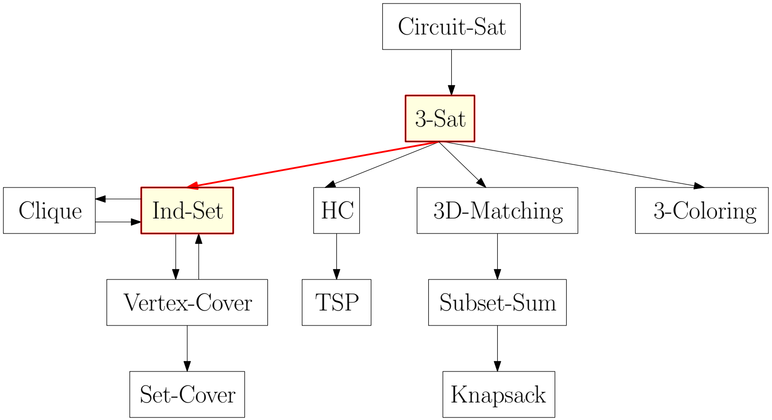
Circuit-Sat $\leq_P$ 3-Sat

- Circuit $\iff$ Formula $\iff$ 3-CNF
- The circuit is satisfiable if and only if the 3-CNF is satisfiable
- The size of the 3-CNF formula is polynomial (indeed, linear) in the size of the circuit

Circuit-Sat $\leq_P$ 3-Sat

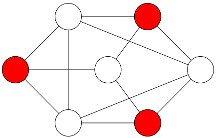
- Circuit $\iff$ Formula $\iff$ 3-CNF
- The circuit is satisfiable if and only if the 3-CNF is satisfiable
- The size of the 3-CNF formula is polynomial (indeed, linear) in the size of the circuit
- Thus, Circuit-Sat $\leq_P$ 3-Sat

Reductions of NP-Complete Problems



Recall: Independent Set Problem

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Independent Set (Ind-Set) Problem

Input: $G = (V, E), k$

Output: whether there is an independent set of size k in G

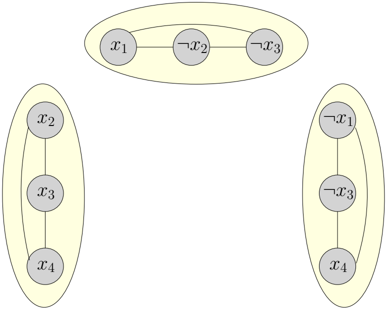
3-Sat $\leq_P$ Ind-Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

3-Sat $\leq_P$ Ind-Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

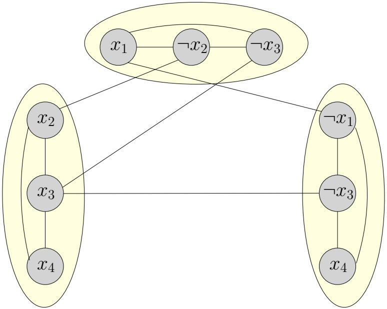
- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group



3-Sat $\leq_P$ Ind-Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

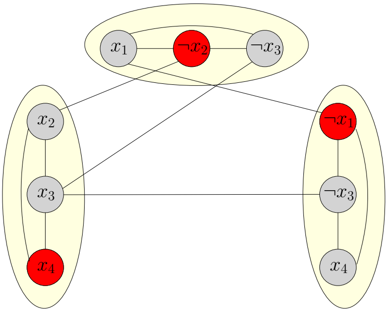
- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals



3-Sat $\leq_P$ Ind-Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

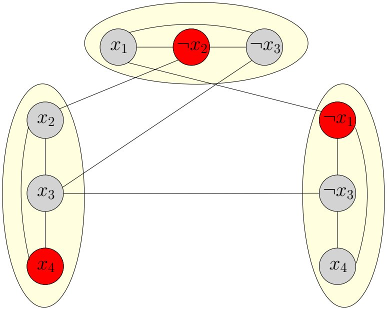
- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = $ #clauses



3-Sat $\leq_P$ Ind-Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#$clauses

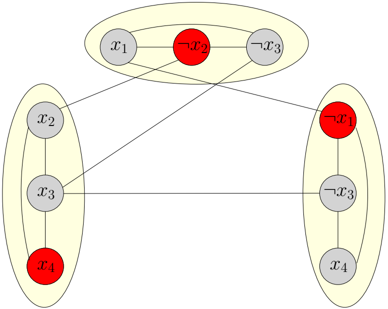


3-Sat instance is yes-instance $\Leftrightarrow$ clique instance is yes-instance:

3-Sat $\leq_P$ Ind-Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#$clauses

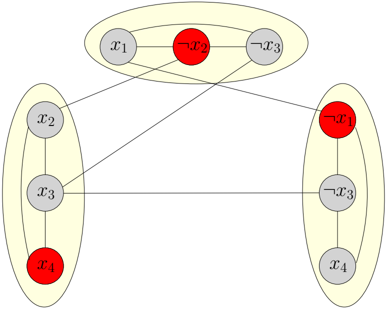


3-Sat instance is yes-instance $\Leftrightarrow$ clique instance is yes-instance:

- satisfying assignment $\Rightarrow$ independent set of size k
- independent set of size k $\Rightarrow$ satisfying assignment

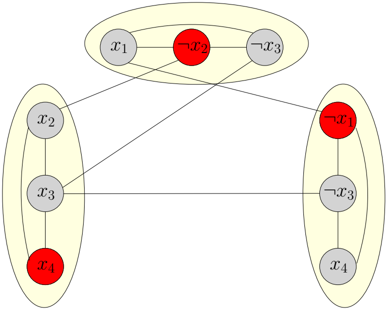
Satisfying Assignment $\Rightarrow$ IS of Size k

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$



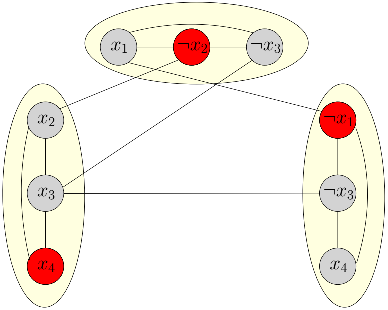
Satisfying Assignment $\Rightarrow$ IS of Size k

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$
- For every clause, at least 1 literal is satisfied



Satisfying Assignment $\Rightarrow$ IS of Size k

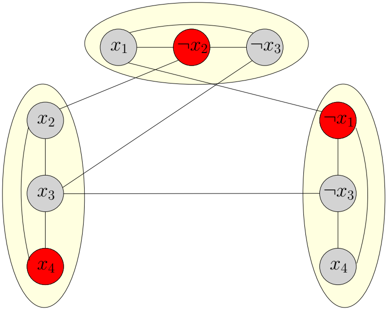
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$
- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent the literal



Satisfying Assignment $\Rightarrow$ IS of Size k

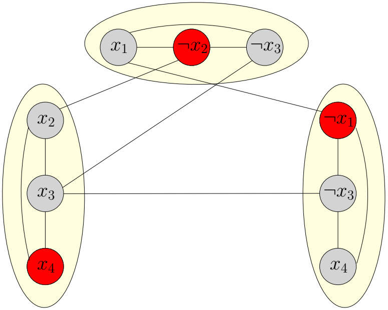
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent the literal
- So, 1 literal from each group



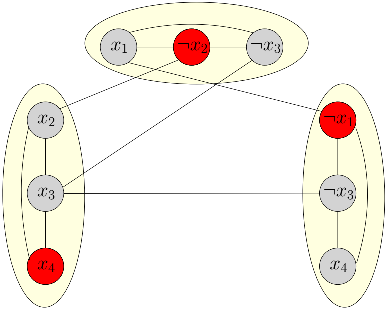
Satisfying Assignment $\Rightarrow$ IS of Size k

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$
- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent the literal
- So, 1 literal from each group
- No contradictions among the selected literals



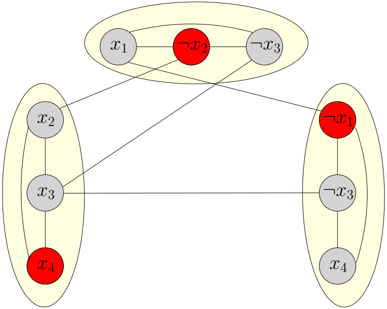
Satisfying Assignment $\Rightarrow$ IS of Size k

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$
- For every clause, at least 1 literal is satisfied
- Pick the vertex correspondent the literal
- So, 1 literal from each group
- No contradictions among the selected literals
- An IS of size k



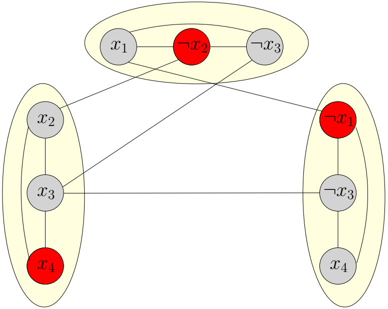
IS of Size k $\Rightarrow$ Satisfying Assignment

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$



IS of Size k $\Rightarrow$ Satisfying Assignment

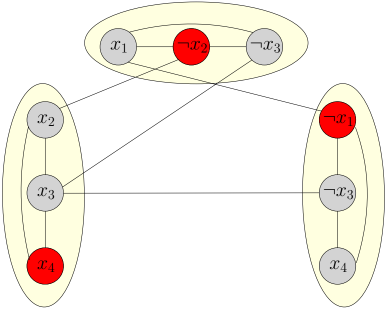
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$
- For every group, exactly one literal is selected in IS



IS of Size $k \Rightarrow$ Satisfying Assignment

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

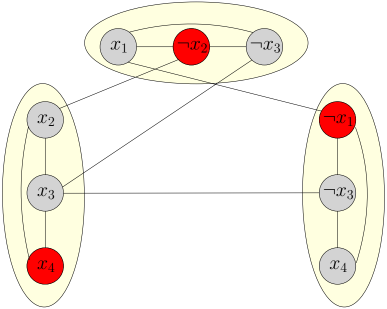
- For every group, exactly one literal is selected in IS
- No contradictions among the selected literals



IS of Size k $\Rightarrow$ Satisfying Assignment

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

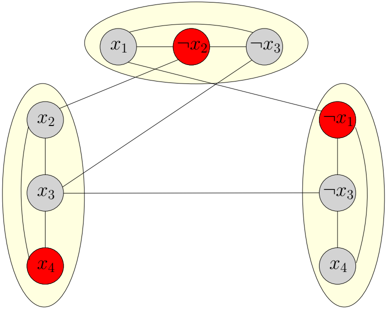
- For every group, exactly one literal is selected in IS
- No contradictions among the selected literals
- If x_i is selected in IS, set $x_i = 1$



IS of Size $k \Rightarrow$ Satisfying Assignment

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

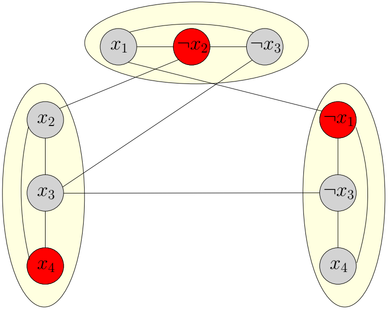
- For every group, exactly one literal is selected in IS
- No contradictions among the selected literals
- If x_i is selected in IS, set $x_i = 1$
- If $\neg x_i$ is selected in IS, set $x_i = 0$



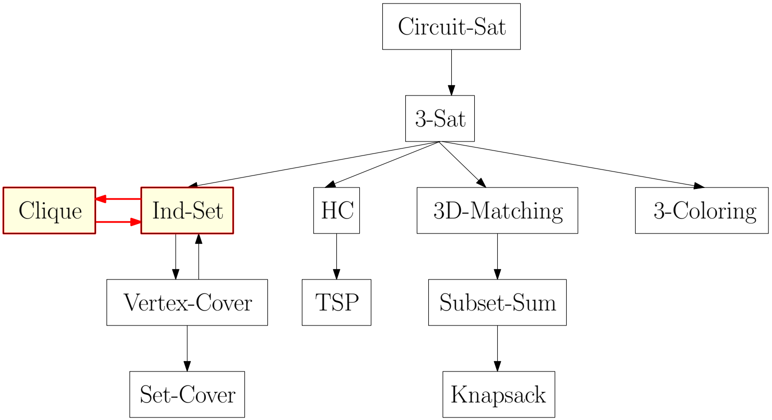
IS of Size $k \Rightarrow$ Satisfying Assignment

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

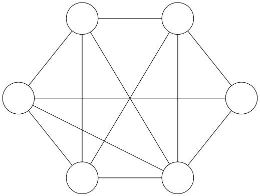
- For every group, exactly one literal is selected in IS
- No contradictions among the selected literals
- If x_i is selected in IS, set $x_i = 1$
- If $\neg x_i$ is selected in IS, set $x_i = 0$
- Otherwise, set x_i arbitrarily



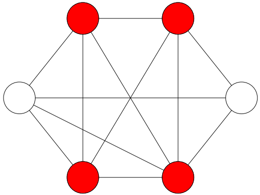
Reductions of NP-Complete Problems



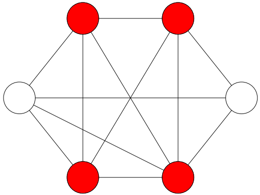
Def. A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$



Def. A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$



Def. A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$

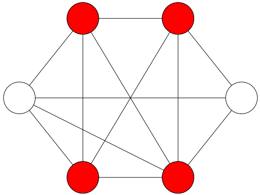


Clique Problem

Input: $G = (V, E)$ and integer $k > 0$,

Output: whether there exists a clique of size k in G

Def. A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$



Clique Problem

Input: $G = (V, E)$ and integer $k > 0$,

Output: whether there exists a clique of size k in G

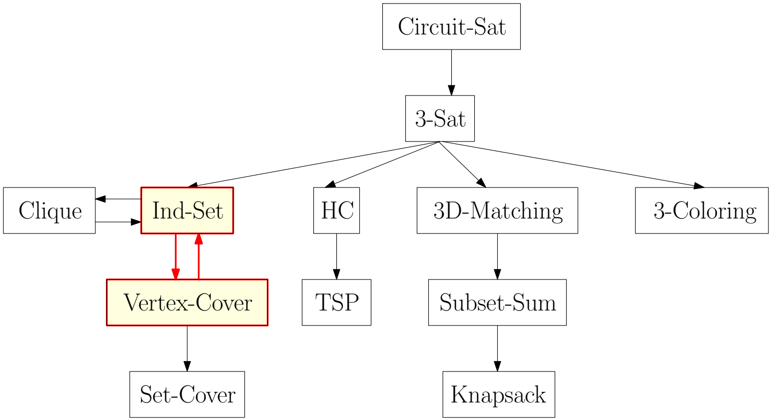
- What is the relationship between Clique and Ind-Set?

Clique $=_P$ Ind-Set

Def. Given a graph $G = (V, E)$, define $\overline{G} = (V, \overline{E})$ be the graph such that $(u, v) \in \overline{E}$ if and only if $(u, v) \notin E$.

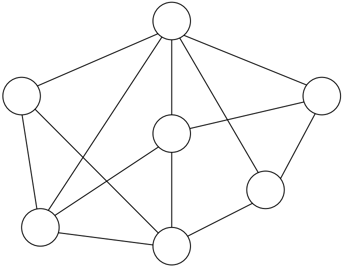
Obs. S is an independent set in G if and only if S is a clique in $\overline{G}$.

Reductions of NP-Complete Problems



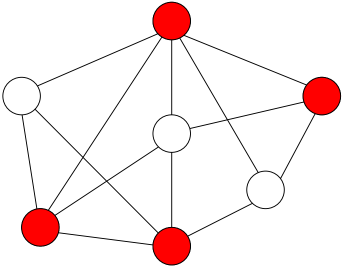
Vertex-Cover

Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



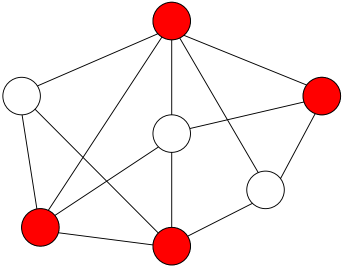
Vertex-Cover

Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



Vertex-Cover

Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



Vertex-Cover Problem

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

Output: whether there is a vertex cover of G of size at most k

Vertex-Cover $=_P$ Ind-Set

Vertex-Cover $=_P$ Ind-Set

Q: What is the relationship between Vertex-Cover and Ind-Set?

Vertex-Cover $=_P$ Ind-Set

Q: What is the relationship between Vertex-Cover and Ind-Set?

A: S is a vertex-cover of $G = (V, E)$ if and only if $V \setminus S$ is an independent set of G.

A Strategy of Polynomial Reduction

Recall the definition of polynomial time reductions:

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

A Strategy of Polynomial Reduction

Recall the definition of polynomial time reductions:

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

- In general, algorithm for Y can call the algorithm for X many times.

A Strategy of Polynomial Reduction

Recall the definition of polynomial time reductions:

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

- In general, algorithm for Y can call the algorithm for X many times.
- However, for most reductions, we call algorithm for X only once

A Strategy of Polynomial Reduction

Recall the definition of polynomial time reductions:

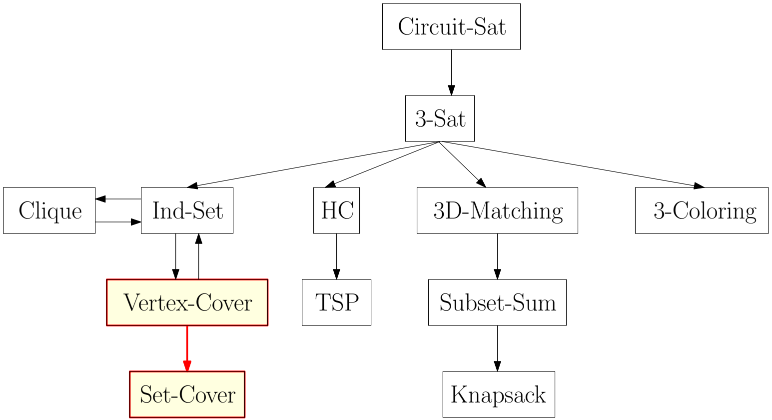
Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

- In general, algorithm for Y can call the algorithm for X many times.
- However, for most reductions, we call algorithm for X only once
- That is, for a given instance s_Y for Y, we only construct one instance s_X for X

A Strategy of Polynomial Reduction

- Given an instance s_Y of problem Y, show how to construct in polynomial time an instance s_X of problem such that:
  - s_Y is a yes-instance of $Y \Rightarrow s_X$ is a yes-instance of X
  - s_X is a yes-instance of $X \Rightarrow s_Y$ is a yes-instance of Y

Reductions of NP-Complete Problems



Set-Cover Problem

Input: ground set U and m subsets $S_1, S_2, \cdots, S_m$ of U and an integer k

Output: whether there is a set $I \subseteq \{1, 2, 3, \cdots, m\}$ of size $\leq k$ such that $\bigcup_{i \in I} S_i = U$

Set-Cover Problem

Input: ground set U and m subsets $S_1, S_2, \cdots, S_m$ of U and an integer k

Output: whether there is a set $I \subseteq \{1, 2, 3, \cdots, m\}$ of size $\leq k$ such that $\bigcup_{i \in I} S_i = U$

Example:

- $U = \{1, 2, 3, 4, 5, 6\}$, $S_1 = \{1, 3, 4\}, S_2 = \{2, 3\}, S_3 = \{3, 6\}, S_4 = \{2, 5\}, S_5 = \{1, 2, 6\}$
- Then $S_1 \cup S_4 \cup S_5 = U$; we need 3 subsets to cover U

Set-Cover Problem

Input: ground set U and m subsets $S_1, S_2, \cdots, S_m$ of U and an integer k

Output: whether there is a set $I \subseteq \{1, 2, 3, \cdots, m\}$ of size $\leq k$ such that $\bigcup_{i \in I} S_i = U$

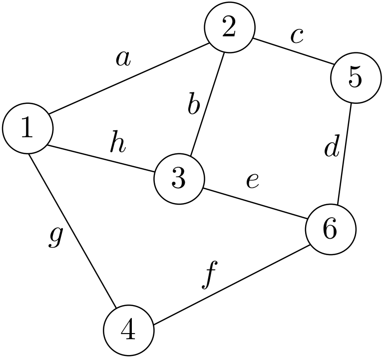
Example:

- $U = \{1, 2, 3, 4, 5, 6\}$, $S_1 = \{1, 3, 4\}, S_2 = \{2, 3\}, S_3 = \{3, 6\}, S_4 = \{2, 5\}, S_5 = \{1, 2, 6\}$
- Then $S_1 \cup S_4 \cup S_5 = U$; we need 3 subsets to cover U

Sample Application

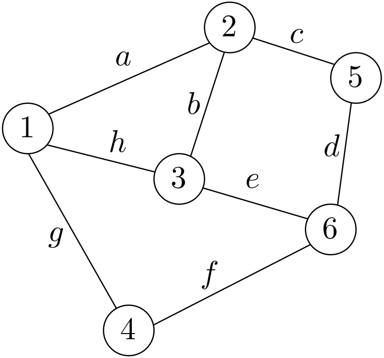
- m available packages for a software
- U is the set of features
- The package i covers the set S_i of features
- want to cover all features using fewest number of packages

Vertex-Cover $\leq_P$ Set-Cover



$$U = \{a, b, c, d, e, f, g\}$$
$$S_1 = \{a, g, h\}$$
$$S_2 = \{a, b, c\}$$
$$S_3 = \{b, e, h\}$$
$$S_4 = \{g, h\}$$
$$S_5 = \{c, d\}$$
$$S_6 = \{d, e, f\}$$

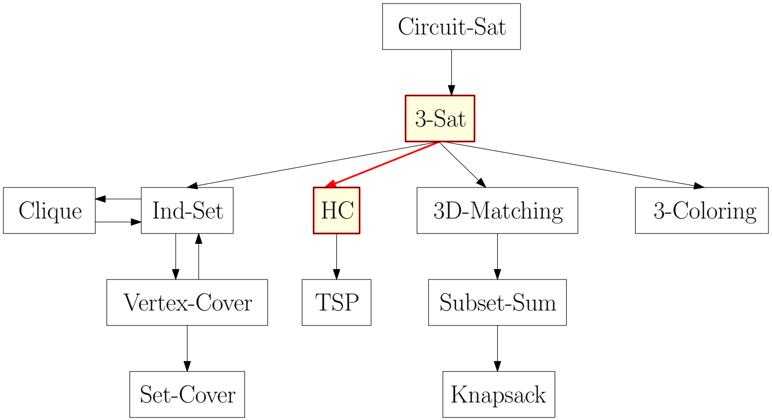
Vertex-Cover $\leq_P$ Set-Cover



$$
\begin{aligned}
U &= \{a,b,c,d,e,f,g\} \\
S_1 &= \{a,g,h\} \\
S_2 &= \{a,b,c\} \\
S_3 &= \{b,e,h\} \\
S_4 &= \{g,h\} \\
S_5 &= \{c,d\} \\
S_6 &= \{d,e,f\}
\end{aligned}
$$

- edges $\implies$ elements in U
- vertices $\implies$ sets
- edge incident on vertex $\implies$ element contained in set
- use vertices to cover edges $\implies$ use sets to cover elements

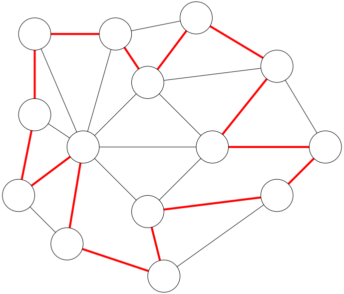
Reductions of NP-Complete Problems



Recall: Hamiltonian Cycle (HC) Problem

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

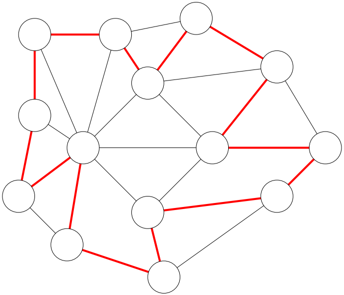
Output: whether G contains a Hamiltonian cycle



Recall: Hamiltonian Cycle (HC) Problem

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

Output: whether G contains a Hamiltonian cycle

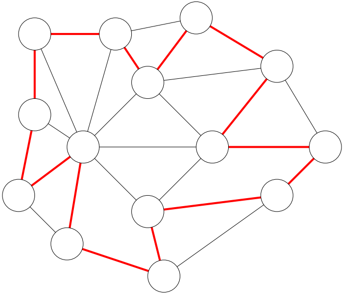


- We consider Hamiltonian Cycle Problem in directed graphs

Recall: Hamiltonian Cycle (HC) Problem

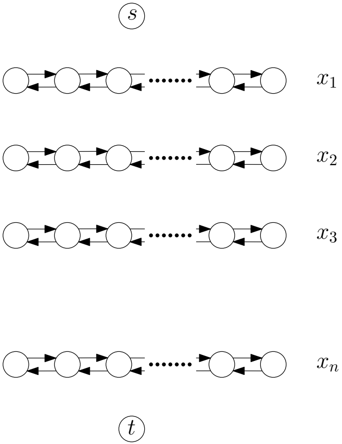
Input: graph $G = (V, E)$

Output: whether G contains a Hamiltonian cycle



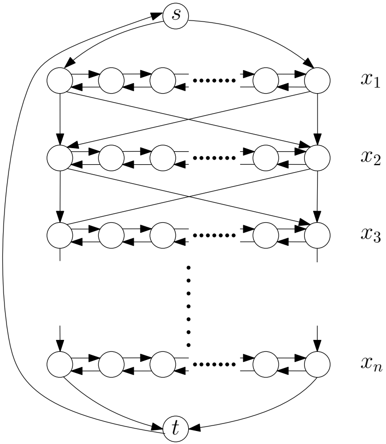
- We consider Hamiltonian Cycle Problem in directed graphs
- Exercise: HC-directed $\leq_P$ HC

3-Sat $\leq_P$ Directed-HC

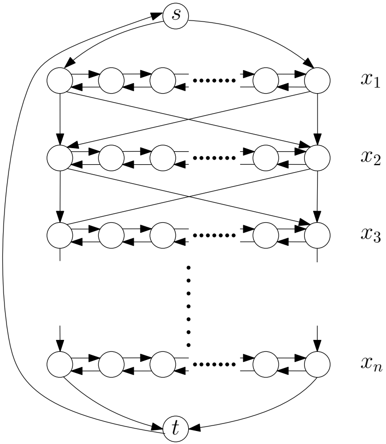


- Vertices s, t
- A long enough double-path P_i for each variable x_i

3-Sat $\leq_P$ Directed-HC

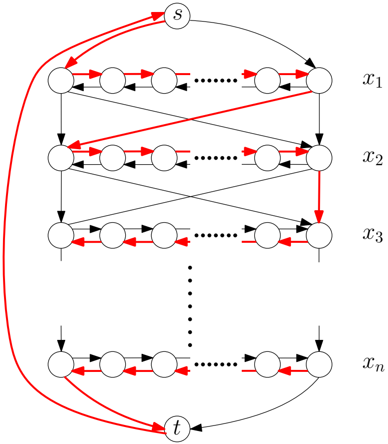


- Vertices s, t
- A long enough double-path P_i for each variable x_i
- Edges from s to P_1
- Edges from P_n to t
- Edges from P_i to P_{i+1}



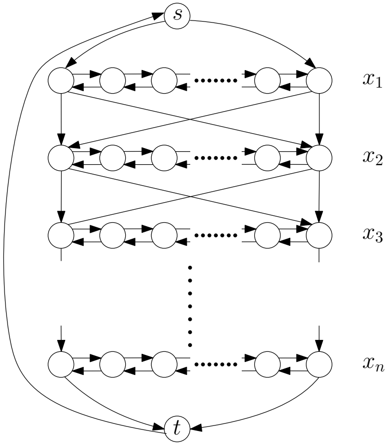
- Vertices s, t
- A long enough double-path P_i for each variable x_i
- Edges from s to P_1
- Edges from P_n to t
- Edges from P_i to P_{i+1}
- $x_i = 1 \iff$ traverse P_i from left to right

3-Sat $\leq_P$ Directed-HC

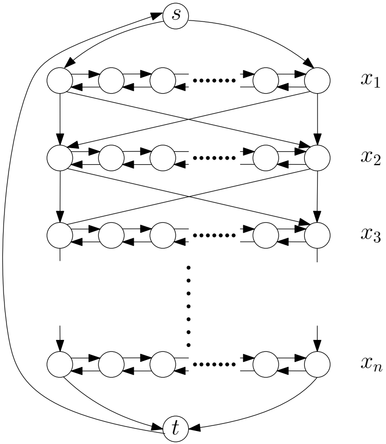


- Vertices s, t
- A long enough double-path P_i for each variable x_i
- Edges from s to P_1
- Edges from P_n to t
- Edges from P_i to P_{i+1}
- $x_i = 1 \iff$ traverse P_i from left to right
- e.g, $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$

3-Sat $\leq_P$ Directed-HC

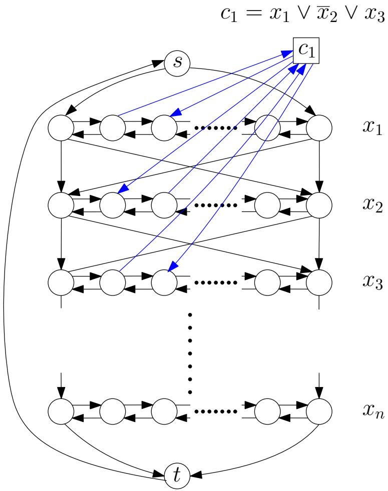


- Vertices s, t
- A long enough double-path P_i for each variable x_i
- Edges from s to P_1
- Edges from P_n to t
- Edges from P_i to P_{i+1}
- $x_i = 1 \iff$ traverse P_i from left to right
- e.g, $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$



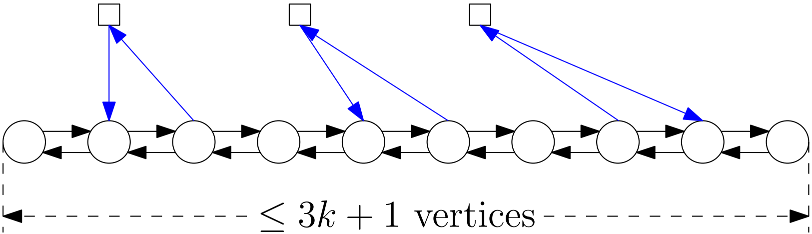
- There are exactly 2^n different Hamiltonian cycles, each correspondent to one assignment of variables

3-Sat $\leq_P$ Directed-HC



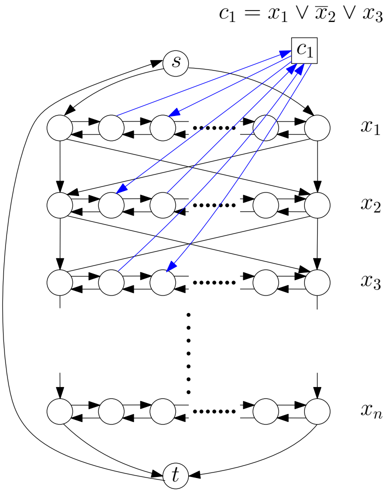
- There are exactly 2^n different Hamiltonian cycles, each correspondent to one assignment of variables
- Add a vertex for each clause, so that the vertex can be visited only if one of the literals is satisfied.

A Path Should Be Long Enough



- k: number of clauses

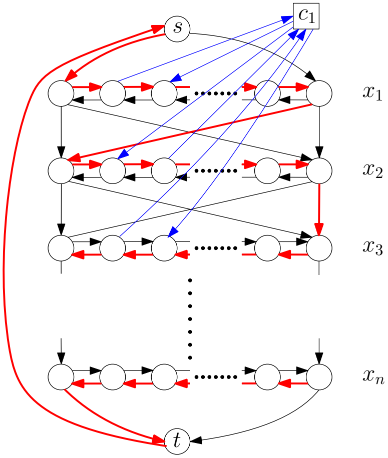
Yes-Instance for 3-Sat $\Rightarrow$ Yes-Instance for Di-HC



- In base graph, construct an HC according to the satisfying assignment

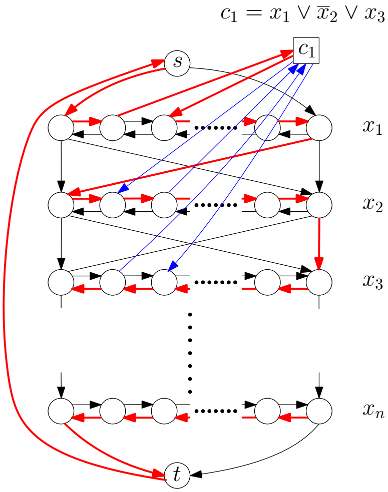
Yes-Instance for 3-Sat $\Rightarrow$ Yes-Instance for Di-HC

$c_1 = x_1 \lor \overline{x}_2 \lor x_3$

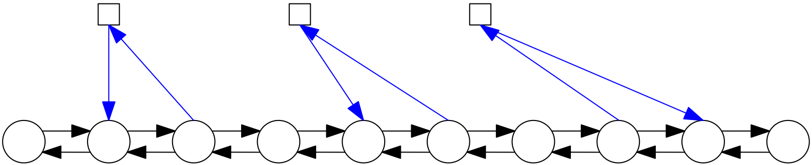


- In base graph, construct an HC according to the satisfying assignment
- For every clause, one literal is satisfied

Yes-Instance for 3-Sat ⇒ Yes-Instance for Di-HC

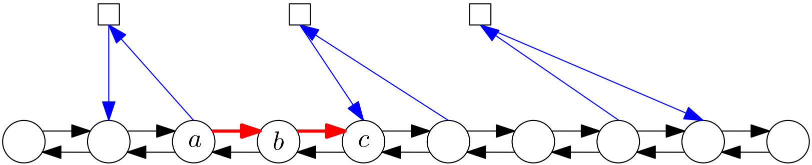


- In base graph, construct an HC according to the satisfying assignment
- For every clause, one literal is satisfied
- Visit the vertex for the clause by taking a "detour" from the path for the literal



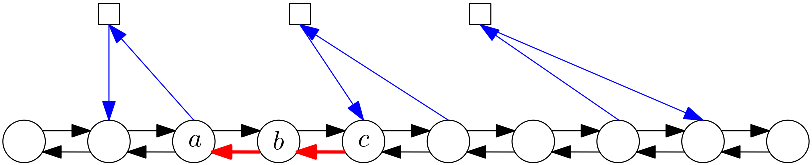
- Idea: for each path P_i, must follow the left-to-right or right-to-right pattern.

Yes-Instance for Di-HC ⇒ Yes-Instance for 3-Sat



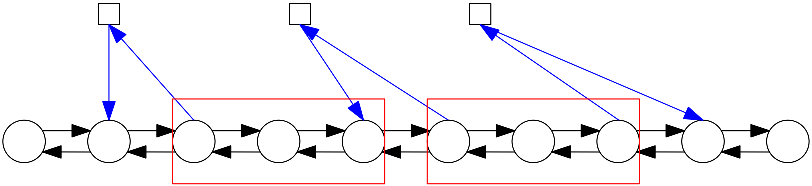
- Idea: for each path P_i, must follow the left-to-right or right-to-right pattern.
- To visit vertex b, can either go a-b-c or b-c-a

Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat



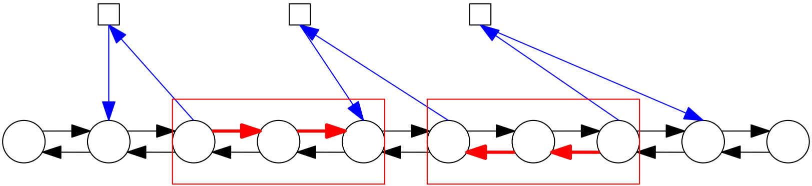
- Idea: for each path P_i, must follow the left-to-right or right-to-right pattern.
- To visit vertex b, can either go a-b-c or b-c-a

Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat



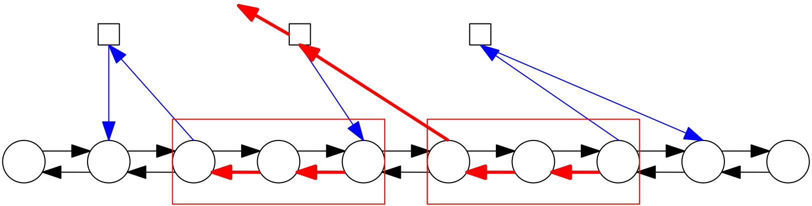
- Idea: for each path P_i, must follow the left-to-right or right-to-right pattern.
- To visit vertex b, can either go a-b-c or b-c-a
- Created "chunks" of 3 vertices.

Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat



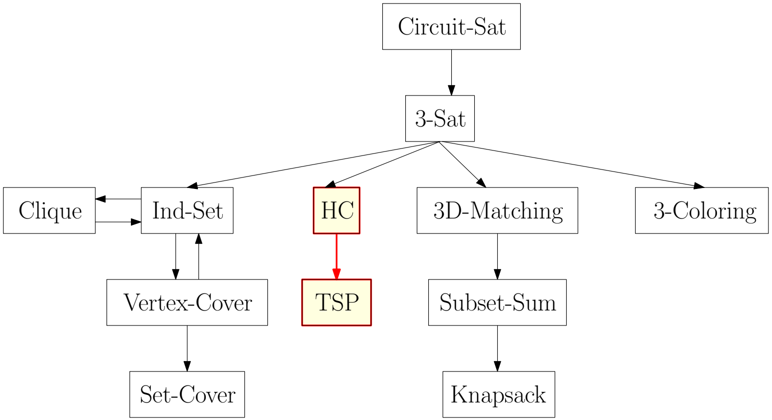
- Idea: for each path P_i, must follow the left-to-right or right-to-right pattern.
- To visit vertex b, can either go a-b-c or b-c-a
- Created "chunks" of 3 vertices.
- Directions of the chunks must be the same

Yes-Instance for Di-HC $\Rightarrow$ Yes-Instance for 3-Sat



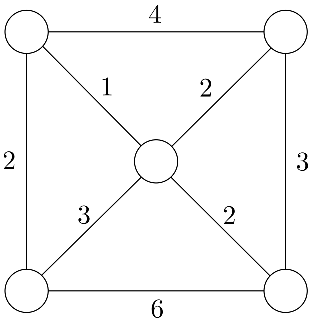
- Idea: for each path P_i, must follow the left-to-right or right-to-right pattern.
- To visit vertex b, can either go a-b-c or b-c-a
- Created "chunks" of 3 vertices.
- Directions of the chunks must be the same
- Can not take a detour to some other path

Reductions of NP-Complete Problems



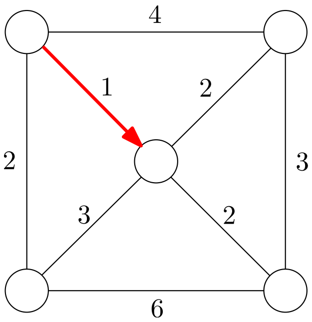
Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



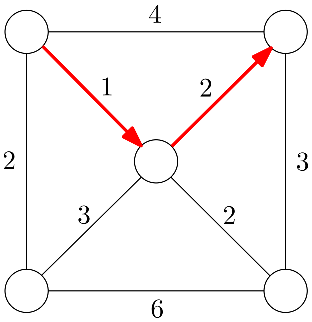
Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



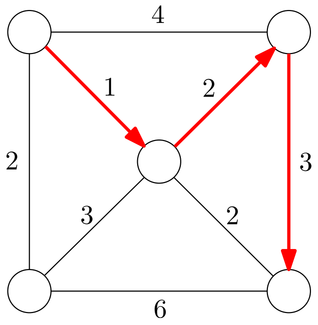
Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



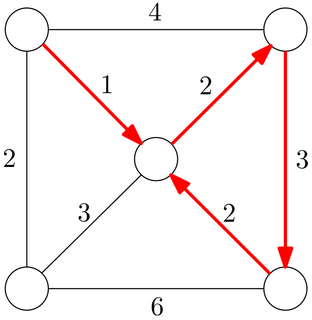
Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



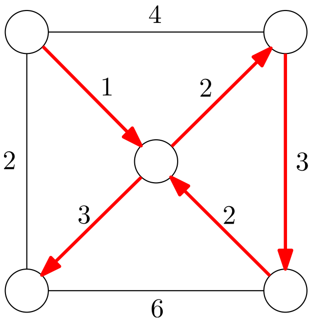
Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



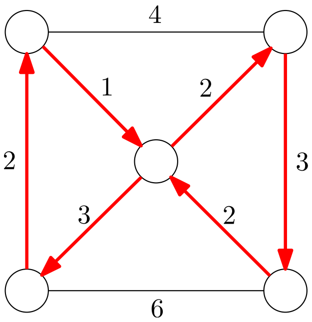
Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



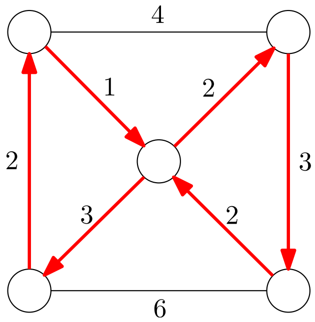
Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost



Traveling Salesman Problem

- A salesman needs to visit n cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost

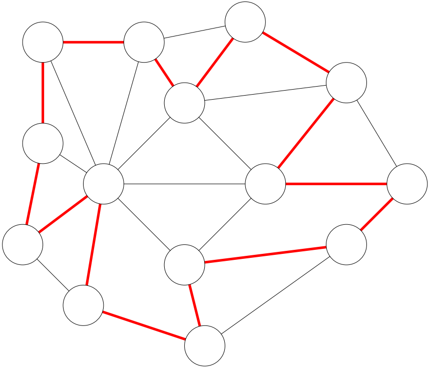


Travelling Salesman Problem (TSP)

Input: a graph $G = (V, E)$, weights $w : E \to \mathbb{R}_{\geq 0}$, and $L > 0$

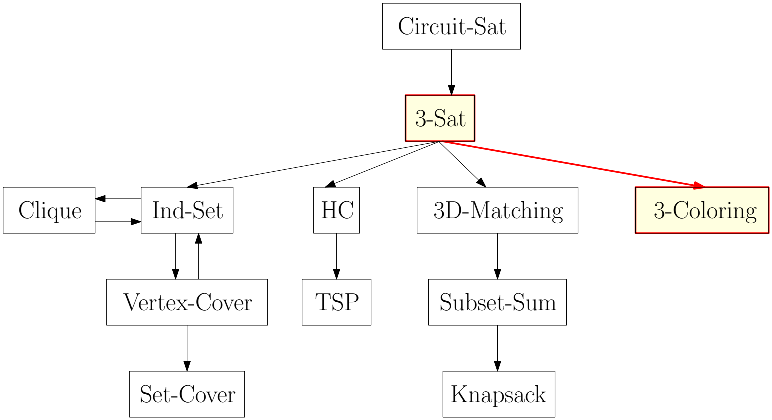
Output: whether there is a tour of length at most D

HC $\leq_P$ TSP



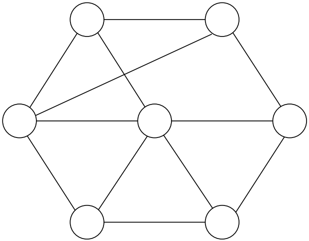
Obs. There is a Hamilton cycle in G if and only if there is a tour for the salesman of length $n = |V|$.

Reductions of NP-Complete Problems



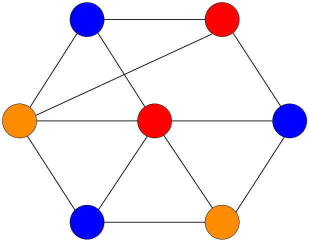
k-coloring problem

Def. A k-coloring of $G = (V, E)$ is a function $f : V \to \{1, 2, 3, \cdots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. G is k-colorable if there is a k-coloring of G.



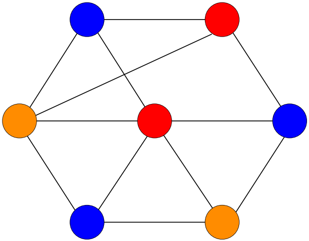
k-coloring problem

Def. A k-coloring of $G = (V, E)$ is a function $f : V \to \{1, 2, 3, \cdots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. G is k-colorable if there is a k-coloring of G.



k-coloring problem

Def. A k-coloring of $G = (V, E)$ is a function $f : V \to \{1, 2, 3, \cdots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. G is k-colorable if there is a k-coloring of G.



k-coloring problem

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

Output: whether G is k-colorable or not

2-Coloring Problem

Obs. A graph G is 2-colorable if and only if it is bipartite.

- There is an $O(m+n)$-time algorithm to decide if a graph G is 2-colorable

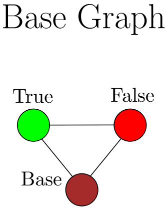
2-Coloring Problem

Obs. A graph G is 2-colorable if and only if it is bipartite.

- There is an $O(m+n)$-time algorithm to decide if a graph G is 2-colorable
- Idea: suppose G is connected. If we fix the color of one vertex in G, then the colors of all other vertices are fixed.

3-SAT $\leq_P$ 3-Coloring

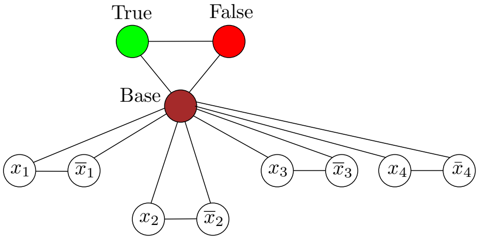
- Construct the base graph



3-SAT $\leq_P$ 3-Coloring

- Construct the base graph

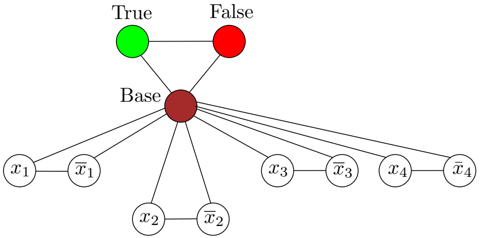
Base Graph



- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

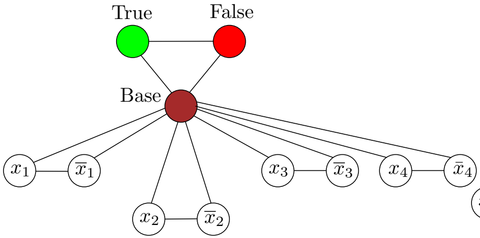
Base Graph

$$x_1 \vee \neg x_2 \vee x_3$$

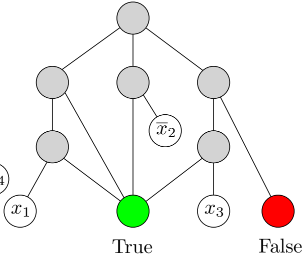


- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

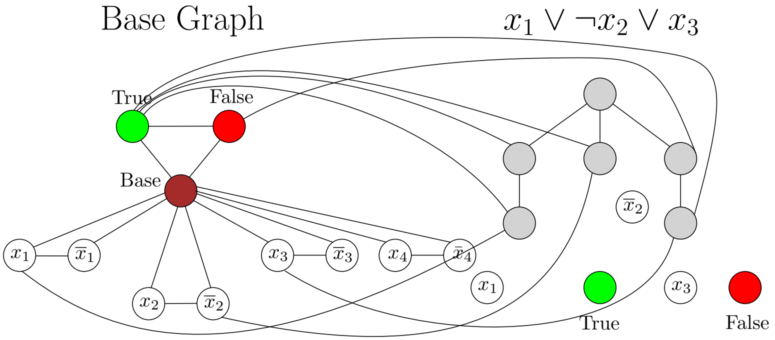


$x_1 \vee \neg x_2 \vee x_3$



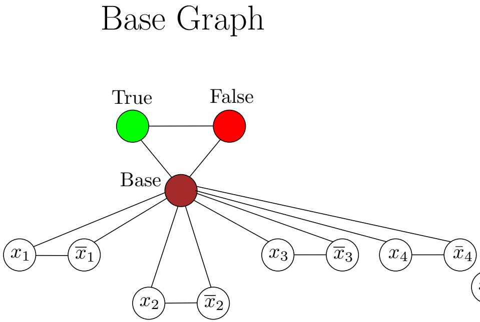
3-SAT $\leq_P$ 3-Coloring

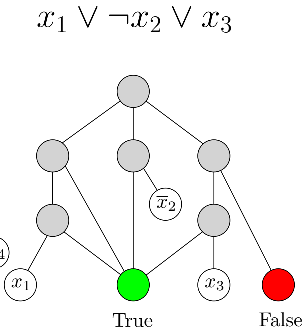
- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.



3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

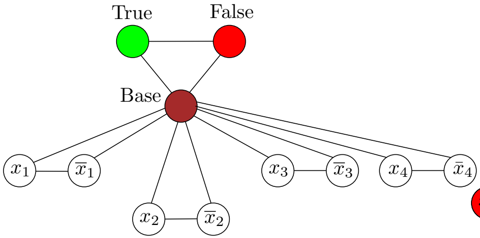




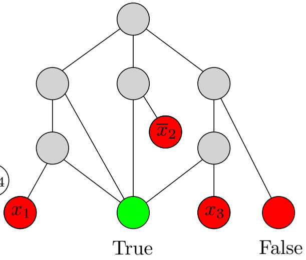
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



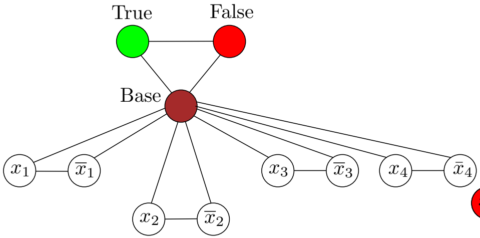
$x_1 \vee \neg x_2 \vee x_3$



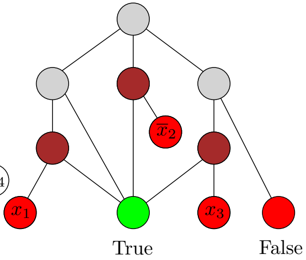
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



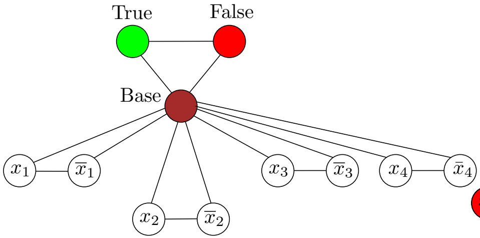
$x_1 \vee \neg x_2 \vee x_3$



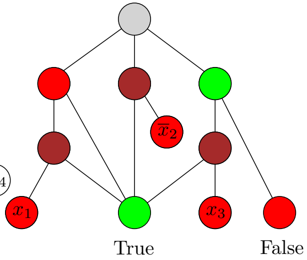
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



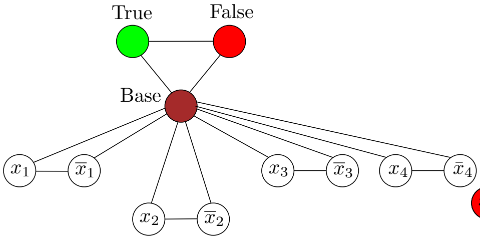
$x_1 \vee \neg x_2 \vee x_3$



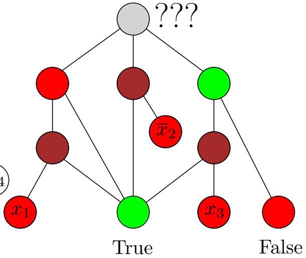
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

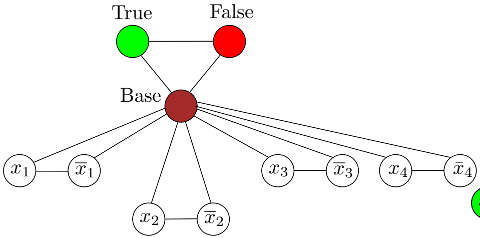


$x_1 \vee \neg x_2 \vee x_3$

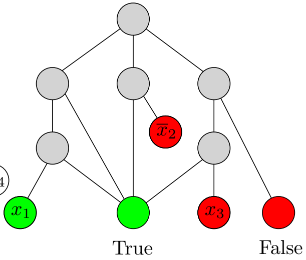


- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



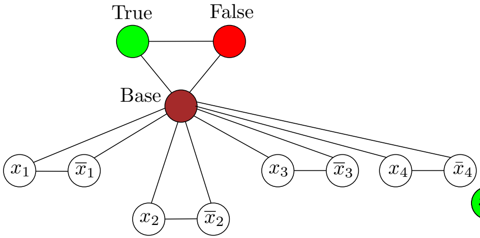
$x_1 \vee \neg x_2 \vee x_3$



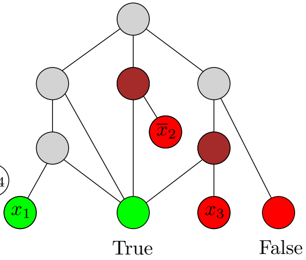
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



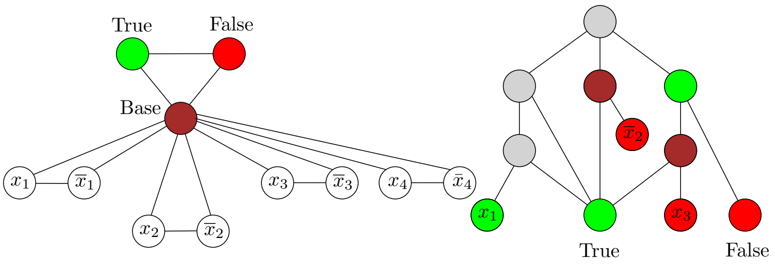
$x_1 \vee \neg x_2 \vee x_3$



- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

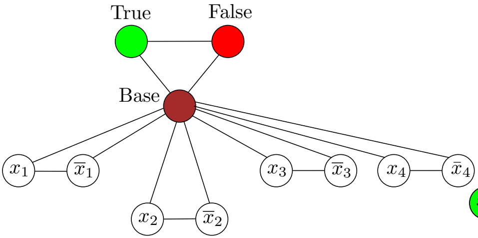
$x_1 \vee \neg x_2 \vee x_3$



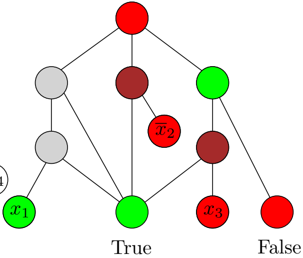
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



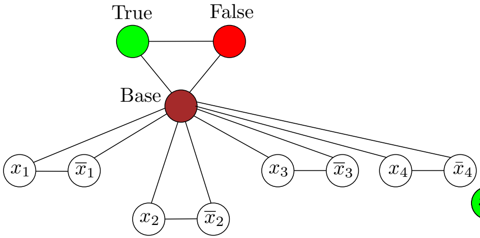
$x_1 \vee \neg x_2 \vee x_3$



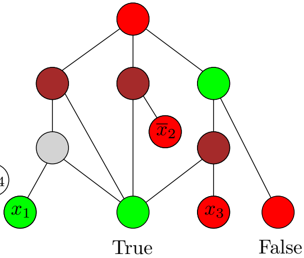
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



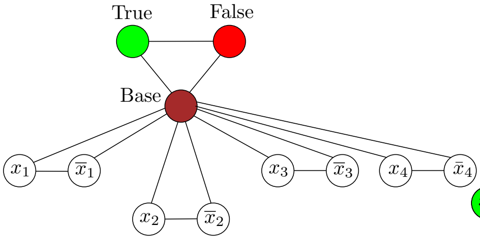
$x_1 \vee \neg x_2 \vee x_3$



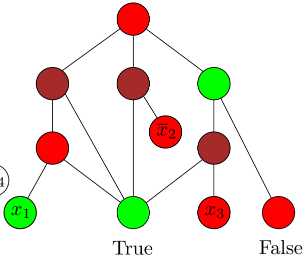
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

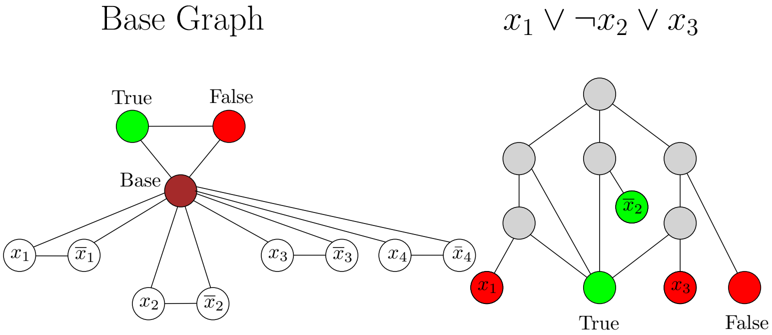


$x_1 \vee \neg x_2 \vee x_3$



3-SAT $\leq_P$ 3-Coloring

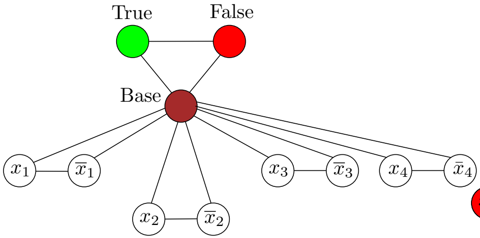
- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.



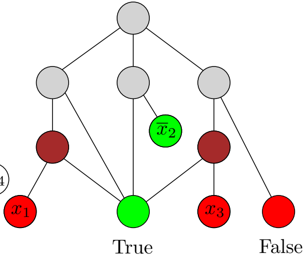
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

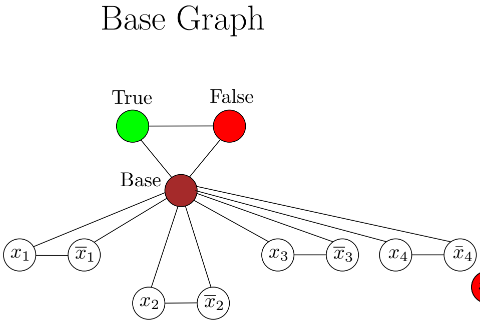


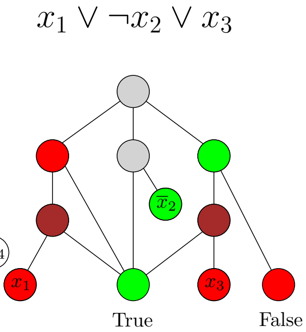
$x_1 \vee \neg x_2 \vee x_3$



3-SAT $\leq_P$ 3-Coloring

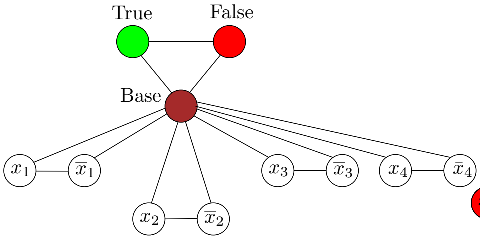
- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.



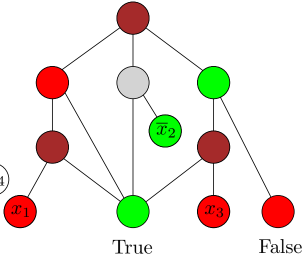


- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph

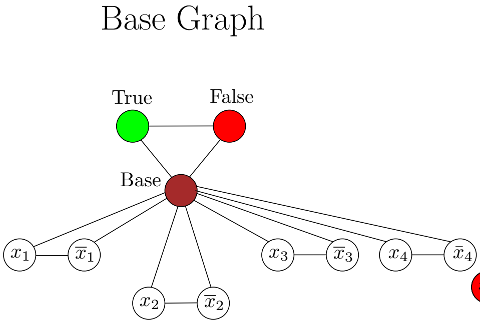


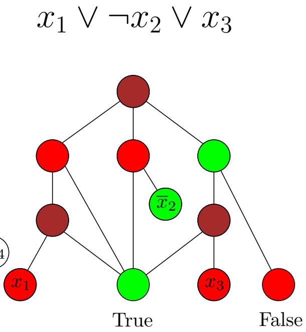
$x_1 \vee \neg x_2 \vee x_3$



3-SAT $\leq_P$ 3-Coloring

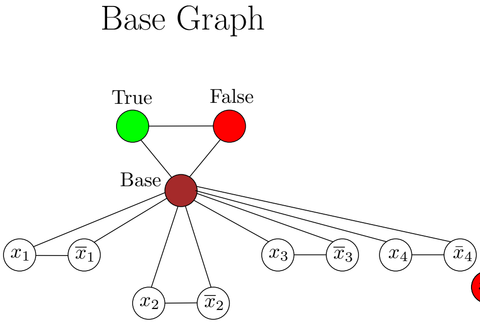
- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

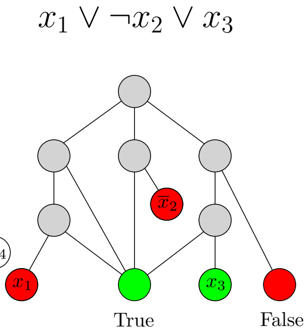




3-SAT $\leq_P$ 3-Coloring

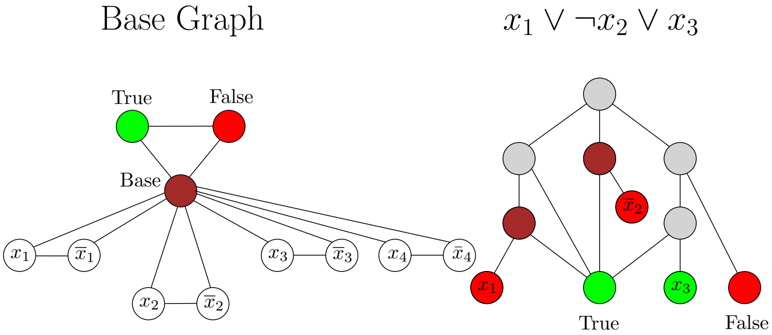
- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.





3-SAT $\leq_P$ 3-Coloring

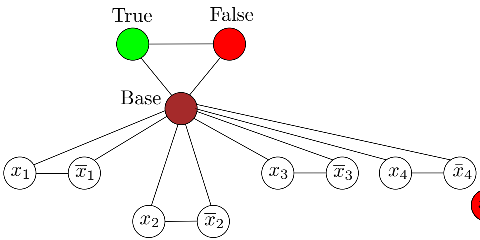
- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.



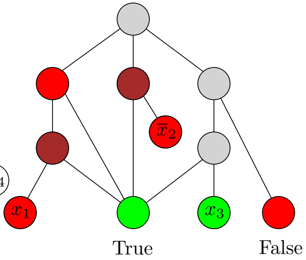
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



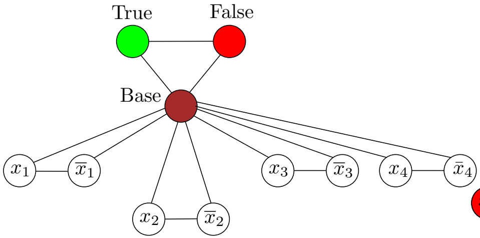
$x_1 \vee \neg x_2 \vee x_3$



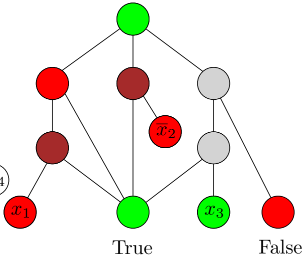
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



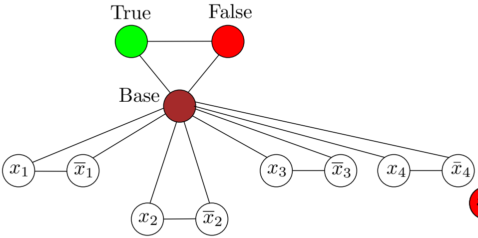
$x_1 \vee \neg x_2 \vee x_3$



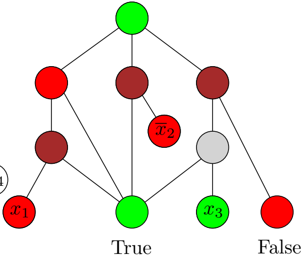
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



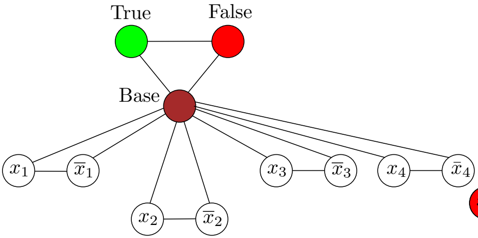
$x_1 \vee \neg x_2 \vee x_3$



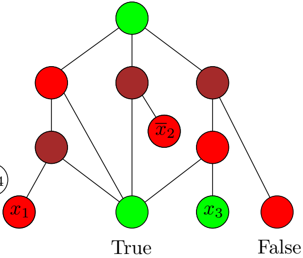
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

Base Graph



$x_1 \vee \neg x_2 \vee x_3$



Outline

Q: How far away are we from proving or disproving $\mathrm{P} = \mathrm{NP}$?

Q: How far away are we from proving or disproving P $=$ NP?

- Try to prove an "unconditional" lower bound on running time of algorithm solving a NP-complete problem.

Q: How far away are we from proving or disproving P = NP?

- Try to prove an "unconditional" lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:

Q: How far away are we from proving or disproving P $=$ NP?

- Try to prove an "unconditional" lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables

Q: How far away are we from proving or disproving P $=$ NP?

- Try to prove an "unconditional" lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$

Q: How far away are we from proving or disproving P $=$ NP?

- Try to prove an "unconditional" lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$
  - Best lower bound is $\Omega(n)$

Q: How far away are we from proving or disproving P $=$ NP?

- Try to prove an "unconditional" lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$
  - Best lower bound is $\Omega(n)$
- Essentially we have no techniques for proving lower bound for running time

Dealing with NP-Hard Problems

- Faster exponential time algorithms
- Solving the problem for special cases
- Fixed parameter tractability
- Approximation algorithms

Faster Exponential Time Algorithms

3-SAT:

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \to 1.844^n \to 1.3334^n$

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \to 1.844^n \to 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \to 1.844^n \to 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \to 1.844^n \to 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:

- Brute-force: $O(n! \cdot \text{poly}(n))$

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \to 1.844^n \to 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:

- Brute-force: $O(n! \cdot \text{poly}(n))$
- Better algorithm: $O(2^n \cdot \text{poly}(n))$

Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \to 1.844^n \to 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:

- Brute-force: $O(n! \cdot \text{poly}(n))$
- Better algorithm: $O(2^n \cdot \text{poly}(n))$
- In practice: TSP Solver can solve Euclidean TSP instances with more than 100,000 vertices

Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees

Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees
- bounded tree-width graphs

Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees
- bounded tree-width graphs
- interval graphs

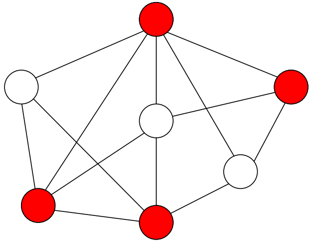
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees
- bounded tree-width graphs
- interval graphs
- $\ldots$

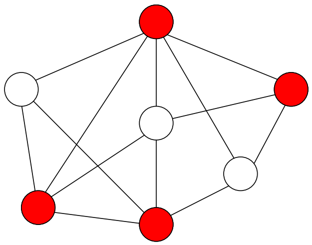
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size k, for a small k (number of nodes is n, number of edges is $\Theta(n)$.)



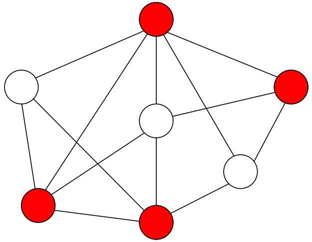
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size k, for a small k (number of nodes is n, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$



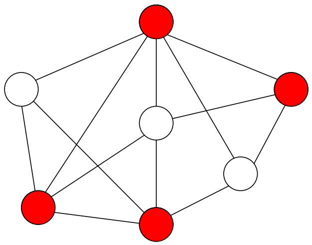
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size k, for a small k (number of nodes is n, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
- Better running time : $O(2^k \cdot kn)$



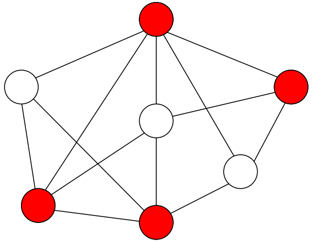
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size k, for a small k (number of nodes is n, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
- Better running time : $O(2^k \cdot kn)$
- Running time is $f(k)n^c$ for some c independent of k



Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size k, for a small k (number of nodes is n, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
- Better running time : $O(2^k \cdot kn)$
- Running time is $f(k)n^c$ for some c independent of k
- Vertex-Cover is fixed-parameter tractable.



Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time

Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time
- Approximation ratio is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution

Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time
- Approximation ratio is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time

Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time
- Approximation ratio is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time

- There is an 1.5-approximation for travelling salesman problem: we can efficiently find a tour whose length is at most 1.5 times the length of the optimal tour

Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time
- Approximation ratio is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time

- There is an 1.5-approximation for travelling salesman problem: we can efficiently find a tour whose length is at most 1.5 times the length of the optimal tour
- 2-approximation for vertex-cover

Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time
- Approximation ratio is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time

- There is an 1.5-approximation for travelling salesman problem: we can efficiently find a tour whose length is at most 1.5 times the length of the optimal tour
- 2-approximation for vertex-cover
- $O(\lg n)$-approximation for set-cover

Outline

Summary

- We consider decision problems
- Inputs are encoded as $\{0,1\}$-strings

Def. The complexity class P is the set of decision problems X that can be solved in polynomial time.

- Alice has a supercomputer, fast enough to run an exponential time algorithm
- Bob has a slow computer, which can only run a polynomial-time algorithm

Def. (Informal) The complexity class NP is the set of problems for which Alice can convince Bob a yes instance is a yes instance

Summary

Def. B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t
- there is a polynomial function p such that, $s \in X$ if and only if there is string t such that $|t| \leq p(|s|)$ and $B(s,t) = 1$.

The string t such that $B(s,t) = 1$ is called a certificate.

Def. The complexity class NP is the set of all problems for which there exists an efficient certifier.

Summary

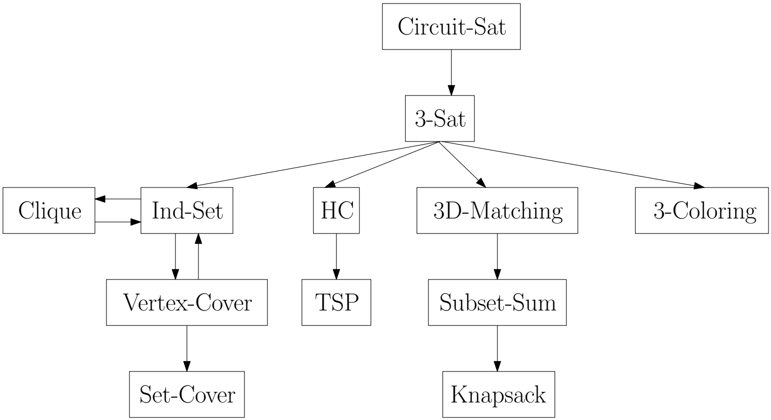
Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

Def. A problem X is called NP-complete if

1. $X \in$ NP, and
2. $Y \leq_{\mathrm{P}} X$ for every $Y \in$ NP.

- If any NP-complete problem can be solved in polynomial time, then $P = NP$
- Unless $P = NP$, a NP-complete problem can not be solved in polynomial time

Summary


Circuit-Sat
3-Sat
Clique
Ind-Set
HC
3D-Matching
3-Coloring
Vertex-Cover
TSP
Subset-Sum
Set-Cover
Knapsack

Summary

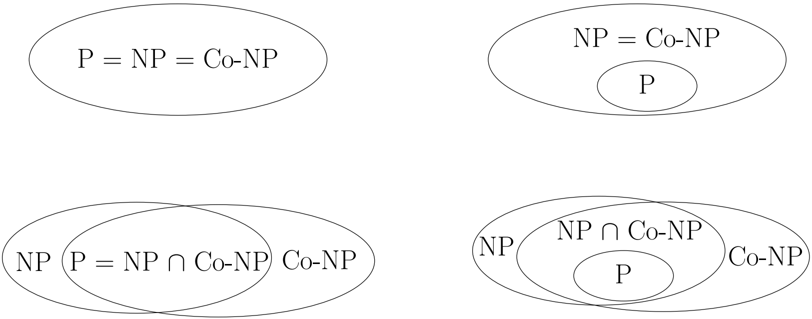
Proof of NP-Completeness for Circuit-Sat

- Fact 1: a polynomial-time algorithm can be converted to a polynomial-size circuit
- Fact 2: for a problem in NP, there is a efficient certifier.
- Given a problem $X \in$ NP, let $B(s, t)$ be the certifier
- Convert $B(s, t)$ to a circuit and hard-wire s to the input gates
- s is a yes-instance if and only if the resulting circuit is satisfiable

- Proof of NP-Completeness for other problems by reductions

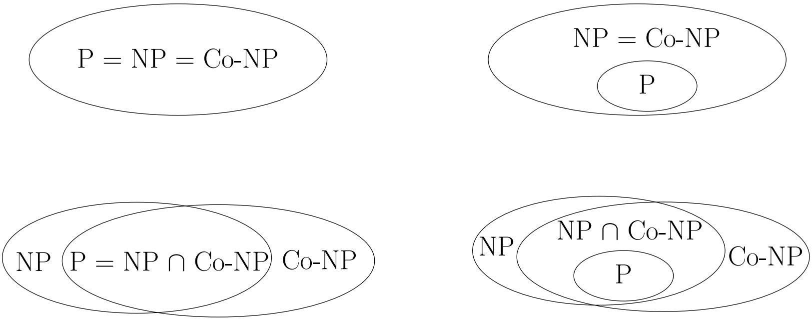
Exercises

Recall the 4 scenarios:



Exercises

Recall the 4 scenarios:



- Prove: P = NP if and only if P = CO-NP

Exercises

For each of the following problem X, answer: whether (1) $X \in$ NP, (2) $X \in$ CO-NP. Each answer is either "yes" or "we do not know".

1. Given a graph $G = (V, E)$, whether G is 4-colorable.

Exercises

For each of the following problem X, answer: whether (1) $X \in$ NP, (2) $X \in$ CO-NP. Each answer is either "yes" or "we do not know".

1. Given a graph $G = (V, E)$, whether G is 4-colorable.
2. Given a graph $G = (V, E)$ and an integer $t > 0$, whether the minimum vertex cover of G has size at least t.

Exercises

For each of the following problem X, answer: whether (1) $X \in$ NP, (2) $X \in$ CO-NP. Each answer is either "yes" or "we do not know".

1. Given a graph $G = (V, E)$, whether G is 4-colorable.
2. Given a graph $G = (V, E)$ and an integer $t > 0$, whether the minimum vertex cover of G has size at least t.
3. Given a directed graph $G = (V, E)$, with weights $w : E \to \mathbb{R}_{>0}$, $s, t \in V$, and a number $L > 0$, whether the length of the shortest path from s to t in G is at most L.

Exercises

For each of the following problem X, answer: whether (1) $X \in$ NP, (2) $X \in$ CO-NP. Each answer is either "yes" or "we do not know".

1. Given a graph $G = (V, E)$, whether G is 4-colorable.
2. Given a graph $G = (V, E)$ and an integer $t > 0$, whether the minimum vertex cover of G has size at least t.
3. Given a directed graph $G = (V, E)$, with weights $w : E \to \mathbb{R}_{>0}$, $s, t \in V$, and a number $L > 0$, whether the length of the shortest path from s to t in G is at most L.
4. Given two boolean formulas, whether the they are equivalent. For example, $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$ and $(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_3)$ are equivalent since they give the same value for every assignment of (x_1, x_2, x_3).

Exercises

Prove the following reductions:

1. 3-Coloring $\leq_P$ 4-Coloring

Exercises

Prove the following reductions:

1. 3-Coloring $\leq_P$ 4-Coloring
2. Hamiltonian-Cycle $\leq_P$ Hamiltonian-Path

Exercises

Prove the following reductions:

1. 3-Coloring $\leq_P$ 4-Coloring
2. Hamiltonian-Cycle $\leq_P$ Hamiltonian-Path
3. Given a directed graph $G = (V, E)$, a weight function $w : E \to \mathbb{Z}$ (weights can be negative) and a vertex $s \in V$, the Simple-Neg-Cycle problem asks whether there is a simple negative cycle in G that contains s. Prove Hamiltonian-Path $\leq_P$ Simple-Neg-Cycle

Exercises

Prove the following reductions:

1. 3-Coloring $\leq_P$ 4-Coloring
2. Hamiltonian-Cycle $\leq_P$ Hamiltonian-Path
3. Given a directed graph $G = (V, E)$, a weight function $w : E \to \mathbb{Z}$ (weights can be negative) and a vertex $s \in V$, the Simple-Neg-Cycle problem asks whether there is a simple negative cycle in G that contains s. Prove Hamiltonian-Path $\leq_P$ Simple-Neg-Cycle
4. Given a graph $G = (V, E)$, the degree-3 spanning tree (D3ST) problem asks whether G contains a spanning tree T of degree at most 3. Prove Hamiltonian-Path $\leq_P$ D3ST.