Analysis of Algorithms

NP Completeness

NP-Completeness

- **Polynomial-time algorithms**

  on inputs of size n, worst-case running time is $O(n^k)$, for a constant k

- Not all problems can be solved in polynomial time
  - Some problems cannot be solved by any computer no matter how much time is provided (Turing's Halting problem) – such problems are called **undecidable**
  - Some problems can be solved but not in $O(n^k)$

Turing’s Halting Problem

- The halting problem
  - Given a program and inputs for it, decide whether it will run forever or will eventually stop.
  - This is not the same thing as actually running a given program and seeing what happens. The halting problem asks whether there is any general prescription for deciding how long to run an arbitrary program so that its halting or non-halting will be revealed.
  - In this abstract framework, there are no resource limitations of memory or time on the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting.
  - Example
    - **while(1) {x = x+1;}**
  - The reason the halting problem is famous is because it is undecidable, which means there is no computable function that correctly determines which programs halt and which ones do not.

34.1 Polynomial Time

- Polynomial time solvable problem are regarded as tractable.
  - Even if the current best algorithm for a problem has a running time of $\Theta(n^{100})$, it is likely that an algorithm with a much better running time will soon be discovered.
  - Problems for many reasonable models of computation, that can be solved in one model can be solved in polynomial in another.
  - Polynomial-time solvable problems has a nice closure property.

$$f, g \quad are \quad polynomial$$
$$\Rightarrow f(g) \quad is \quad also\ polynomial$$

Class of "P" Problems

- **Class P** consists of (decision) problems that are solvable in polynomial time:

    there exists an algorithm that can solve the problem in $O(n^k)$, k constant

- Problems in P are also called **tractable**
- Problems not in P are also called **intractable**
    - Can be solved in reasonable time only for small inputs

Decision problems

A class of problems where for each input algorithm have to produce one of two possible answers - “yes” or “no”, i.e. computable functions of the type

$f\colon \mathbf{N} \to \{0,1\}$

Optimization & Decision Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is yes or no
- **Optimization problems**
  - Find a solution with the "best" value
- Optimization problems can be cast as decision problems that are easier to study
  - *E.g.:* Shortest path: G = unweighted directed graph
    - Find a path between u and v that uses the fewest edges
    - *Does a path exist from u to v consisting of at most k edges?*

Nondeterministic Algorithms

Nondeterministic algorithm = two stage procedure:

1) Nondeterministic ("guessing") stage:

   generate an arbitrary string that can be thought of as a candidate solution ("certificate")

2) Deterministic ("verification") stage:

   take the certificate and the instance to the problem and returns YES if the certificate represents a solution

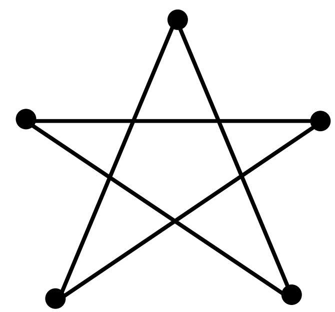
- **Nondeterministic polynomial** (NP) = verification stage is polynomial

Class of “NP” Problems

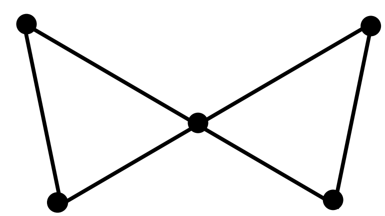
- **Class NP** consists of problems that are verifiable in polynomial time (i.e., could be solved by nondeterministic polynomial algorithms)
  - If we were given a “certificate” of a solution, we could verify that the certificate is correct in time polynomial to the size of the input

E.g.: Hamiltonian Cycle

- **Given:** a directed graph G = (V, E), determine a simple cycle that contains each vertex in V
  - Each vertex can only be visited once
- **Certificate**:
  - Sequence: $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$
- **Verification**:
  1) $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, |V|$
  2) $(v_{|V|}, v_1) \in E$

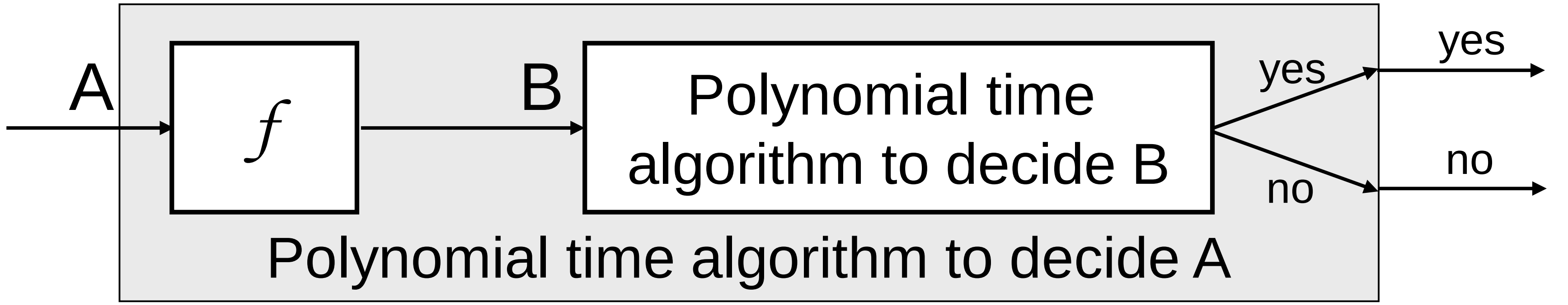


hamiltonian



not hamiltonian

Polynomial Reduction Algorithm



- To solve a decision problem A in polynomial time

1. Use a polynomial time reduction algorithm to transform A into B
2. Run a known polynomial time algorithm for B
3. Use the answer for B as the answer for A

Reductions

- Given two problems A, B, we say that A is **reducible** to B ($A \leq_p B$) if:

1. There exists a function f that converts the input of A to inputs of B in polynomial time
2. A(i) = YES $\Leftrightarrow$ B(f(i)) = YES

NP-Completeness

- A problem B is **NP-complete** if:

  1) $B \in$ **NP**

  2) $A \leq_p B$ for all $A \in$ **NP**

- If B satisfies only property 2) we say that B is **NP-hard**
- No polynomial time algorithm has been discovered for an **NP-Complete** problem
- No one has ever proven that no polynomial time algorithm can exist for any **NP-Complete** problem

NP-Completeness (why NPC?)

- A problem $p \in$NP, and any other problem p' can be translated as p in poly time.
- So if p can be solved in poly time, then all problems in NP can be solved in poly time.

Proving NP-Completeness

Theorem: If A is NP-Complete and A $\leq_p$ B

$\Rightarrow$ B is NP-Hard

In addition, if B $\in$ NP

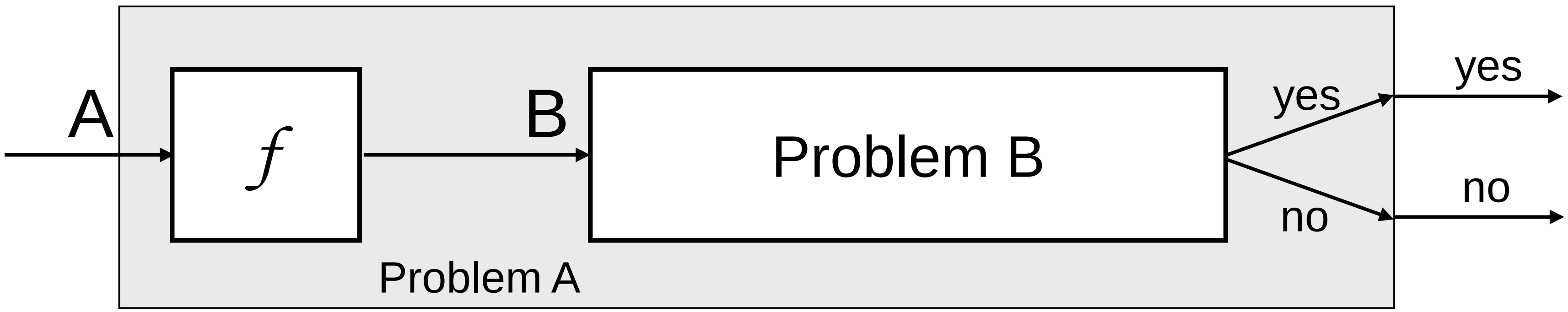
$\Rightarrow$ B is NP-Complete

Proof: Assume that B $\in$ P

Since A $\leq_p$ B $\Rightarrow$ A $\in$ P contradiction!

$\Rightarrow$ B is NP-Hard

Reduction and NP-Completeness



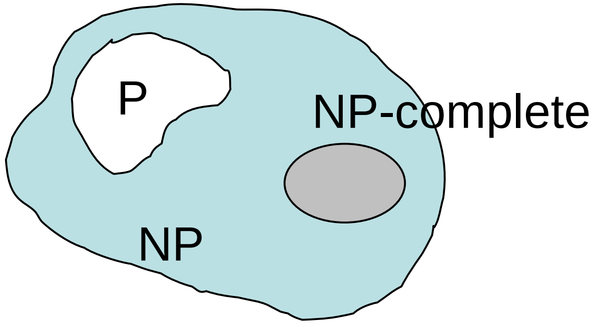
- Suppose we know:
  - No polynomial time algorithm exists for problem B
  - We have a polynomial reduction f from A to B

$\Rightarrow$ No polynomial time algorithm exists for A

Proving NP-Completeness

- Prove that the problem B is in NP
  - A randomly generated string can be checked in polynomial time to determine if it represents a solution
- Show that **one known** NP-Complete problem can be transformed to B in polynomial time
  - No need to check that **all** NP-Complete problems are reducible to B

Is P = NP?



- Any problem in P is also in NP:

$$P \subseteq NP$$

- We can solve problems in P, even without having a certificate
- The big (and open question) is whether P = NP

Theorem: If any NP-Complete problem can be solved in polynomial time $\Rightarrow$ then P = NP.

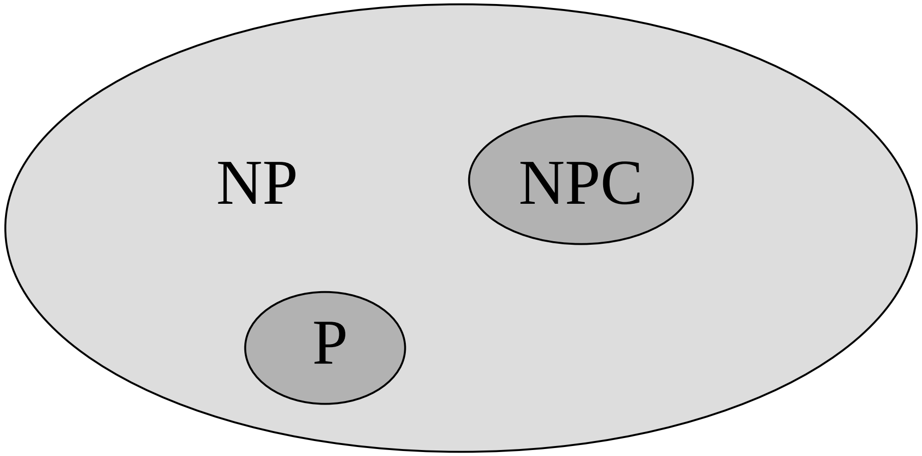
Relation among P, NP, NPC

- $P \subseteq NP$ (Sure)
- $NPC \subseteq NP$ (sure)
- $P = NP$ (or $P \subset NP$, or $P \neq NP$) ???
- $NPC = NP$ (or $NPC \subset NP$, or $NPC \neq NP$) ???
- $P \neq NP$: one of the deepest, most perplexing open research problems in (theoretical) computer science since 1971.

Arguments about P, NP, NPC

- No poly algorithm found for any NPC problem (even so many NPC problems)
- No proof that a poly algorithm cannot exist for any of NPC problems, (even having tried so long so hard).
- Most theoretical computer scientists believe that NPC is intractable (i.e., hard, and P $\neq$ NP).

View of Theoretical Computer Scientists on P, NP, NPC



$$P \subset NP,\ NPC \subset NP,\ P \cap NPC = \emptyset$$

Why discussion on NPC

- If a problem is proved to be NPC, a good evidence for its intractability (hardness).
- Not waste time on trying to find efficient algorithm for it
- Instead, focus on design approximate algorithm or a solution for a special case of the problem
- Some problems looks very easy on the surface, but in fact, is hard (NPC).

P & NP-Complete Problems

- **Shortest simple path**
  - Given a graph G = (V, E) find a **shortest** path from a source to all other vertices
  - Polynomial solution: O(VE)
- **Longest simple path**
  - Given a graph G = (V, E) find a **longest** path from a source to all other vertices
  - NP-complete

P & NP-Complete Problems

- **Euler tour**
  - G = (V, E) a connected, directed graph find a cycle that traverses each edge of G exactly once (may visit a vertex multiple times)
  - Polynomial solution O(E)
- **Hamiltonian cycle**
  - G = (V, E) a connected, directed graph find a cycle that visits each vertex of G exactly once
  - NP-complete

A First NP-complete problem

- Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete,
  - we need a “first” NPC problem.

- Circuit-satisfiability problem

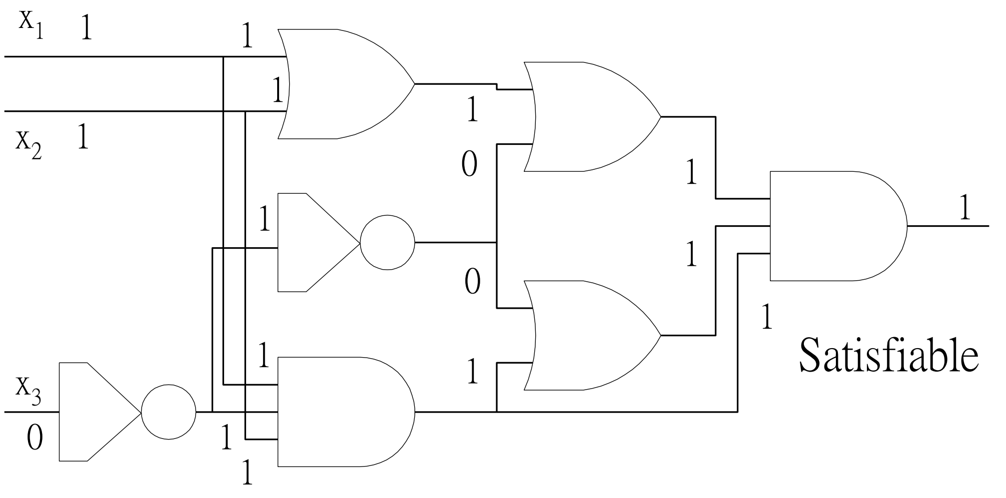
First NP-complete problem—Circuit Satisfiability (problem definition)

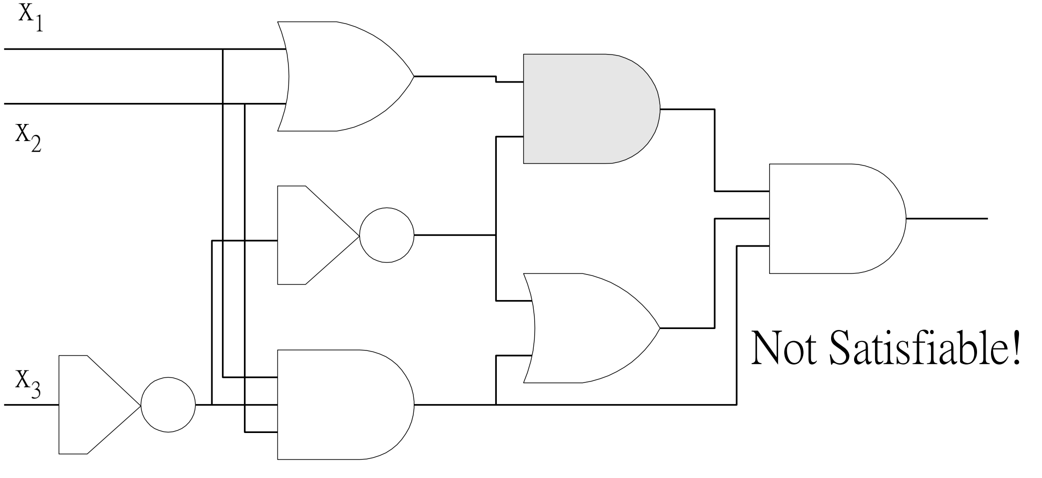
- Boolean combinational circuit
  - Boolean combinational elements, wired together
  - Each element, inputs and outputs (binary)
  - Limit the number of outputs to 1.
  - Called *logic gates*: NOT gate, AND gate, OR gate.
  - *true table*: giving the outputs for each setting of inputs
  - true assignment: a set of boolean inputs.
  - **satisfying assignment**: a true assignment causing the output to be 1
  - **A circuit is satisfiable if it has a satisfying assignment.**

Circuit Satisfiability Problem: definition

- Circuit satisfying problem:
  – given a boolean combinational circuit composed of AND, OR, and NOT, is it stisfiable?
- CIRCUIT-SAT={<C>: C is a satisfiable boolean circuit}
- Implication:
  – in the area of computer-aided hardware optimization, if a subcircuit always produces 0, then the subcircuit can be replaced by a simpler subcircuit that omits all gates and just output a 0.

Circuit Satisfiability Problem





Solving circuit-satisfiability problem

- Intuitive solution:
  - for each possible assignment, check whether it generates 1.
  - suppose the number of inputs is k, then the total possible assignments are 2^k. So the running time is $\Omega(2^k)$. When the size of the problem is $\Theta(k)$, then the running time is not poly.

Circuit Satisfiability: Theorem

- Lemma 34.5.
  - The circuit-satisfiability problem belongs to the class NP.
- Lemma 34.6.
  - The circuit-satisfiability problem is NP-hard.
- Theorem 34.7.
  - The circuit-satisfiability problem is NP-Complete.

Circuit-satisfiability problem is NP-complete

- *Lemma 34.5*: CIRCUIT-SAT belongs to NP.
- Proof:
  – CIRCUIT-SAT is poly-time verifiable.
  – Given (an encoding of) a CIRCUIT-SAT problem C and a certificate, which is an assignment of boolean values to (all) wires in C.
  – The algorithm is constructed as follows: just checks each gates and then the output wire of C:
    - If for every gate, the computed output value matches the value of the output wire given in the certificate and the output of the whole circuit is 1, then the algorithm outputs 1, otherwise 0.
    - The algorithm is executed in poly time (even linear time).

Circuit-satisfiability problem is NP-complete (cont.)

- *Lemma 34.6:* (page 991)
  - CIRCUIT-SAT is NP-hard.
- Proof:
  - Difficult to proof
  - If you are interested, read from the book

NPC proof –Formula Satisfiability (SAT)

- SAT definition
  - n boolean variables: $x_1, x_2, \ldots, x_n$.
  - M boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$ and
  - Parentheses.
- A SAT ϕ is satisfiable
  - if there exists a true assignment which causes ϕ to evaluate to 1.

E.g.: $\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$

Certificate: $x_1 = 1$, $x_2 = 0 \Rightarrow \Phi = 1 \wedge 1 \wedge 1 = 1$

- SAT={< ϕ>: ϕ is a satifiable boolean formula}.
- The historical honor of the first NP-complete problem ever shown.

SAT is NP-complete

- *Theorem 34.9:* (page 997)
  - SAT is NP-complete.
- Proof:
  - SAT belongs to NP.
    - Given a satisfying assignment, the verifying algorithm replaces each variable with its value and evaluates the formula, in poly time.
  - SAT is NP-hard (show CIRCUIT-SAT$\leq_p$ SAT).

SAT is NP-complete (cont.)

- **CIRCUIT-SAT$\leq_p$ SAT**, i.e., any instance of circuit satisfiability can be reduced in poly time to an instance of formula satisfiability.
- Intuitive induction:
  - Look at the gate that produces the circuit output.
  - Inductively express each of gate's inputs as formulas.
  - Formula for the circuit is then obtained by writing an expression that applies the gate's function to its input formulas.

•Unfortunately, this is not a poly reduction

–Shared formula (the gate whose output is fed to 2 or more inputs of other gates) cause the size of generated formula to grow exponentially.

SAT is NP-complete (cont.)

- Correct reduction:
  - For every wire x_i of C, give a variable x_i in the formula.
  - Every gate can be expressed as $x_o \leftrightarrow (x_{i_1} \theta\, x_{i_2} \theta \ldots \theta\, x_{i_l})$
  - The final formula ϕ is the AND of the circuit output variable and conjunction of all clauses describing the operation of each gate. (example Figure 34.10)
- Correctness of the reduction
  - Clearly the reduction can be done in poly time.
  - C is satisfiable if and only if ϕ is satisfiable.
    - If C is satisfiable, then there is a satisfying assignment. This means that each wire of C has a well-defined value and the output of C is 1. Thus the assignment of wire values to variables in ϕ makes each clause in ϕ evaluate to 1. So ϕ is 1.
    - The reverse proof can be done in the same way.

Example of reduction of CIRCUIT-SAT to SAT

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

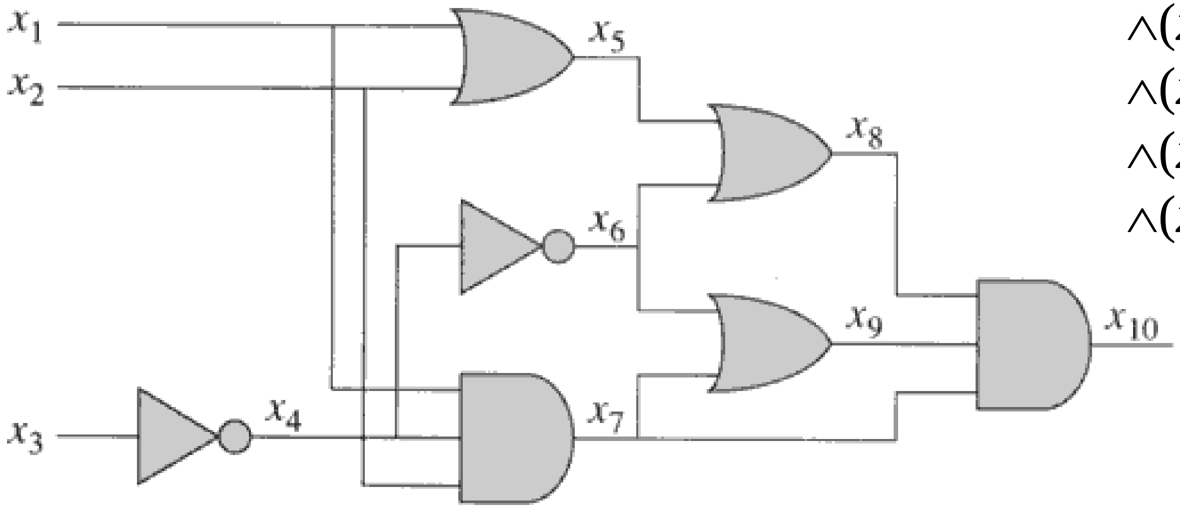


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

INCORRECT REDUCTION: $\phi = x_{10} = x_7 \wedge x_8 \wedge x_9 = (x_1 \wedge x_2 \wedge x_4) \wedge (x_5 \vee x_6) \wedge (x_6 \vee x_7)$
$= (x_1 \wedge x_2 \wedge x_4) \wedge ((x_1 \vee x_2) \vee \neg x_4) \wedge (\neg x_4 \vee (x_1 \wedge x_2 \wedge x_4)) = \ldots.$

NP-completeness proof structure

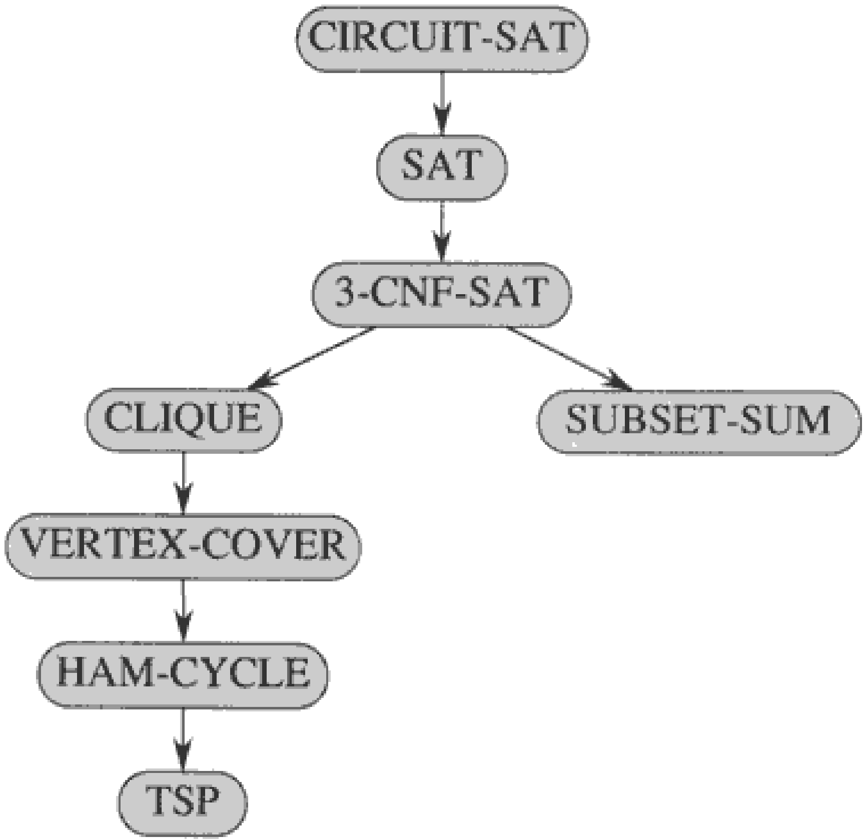


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

3-CNF Satisfiability

3-CNF Satisfiability Problem:

- n boolean variables: $x_1, x_2, \ldots, x_n$
- **Literal**: x_i or $\neg x_i$ (a variable or its negation)
- **Clause:** c_j = an **OR** of **three literals** (m clauses)
- Formula: $\Phi = c_1 \wedge c_2 \wedge \ldots \wedge c_m$

- *E.g.:*

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

- **3-CNF** is NP-Complete

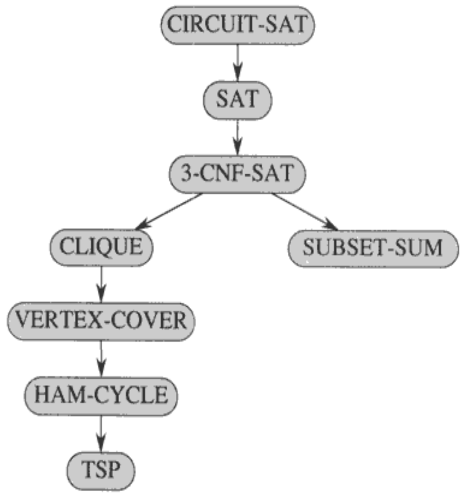


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

Clique

Clique Problem:

– Undirected graph G = (V, E)

– **Clique:** a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph)

– **Size of a clique:** number of vertices it contains

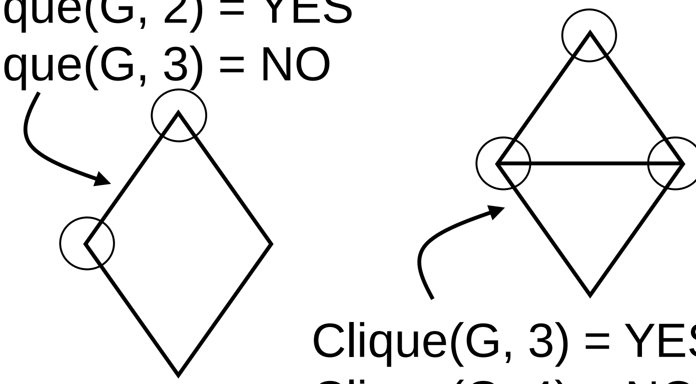
Optimization problem:

– Find a clique of maximum size

Decision problem:

– Does G have a clique of size k?

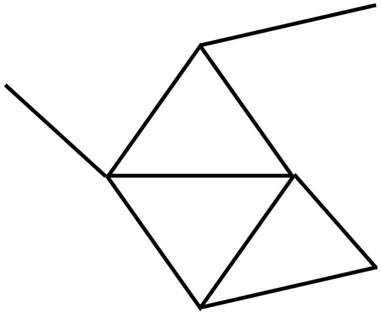
Clique(G, 2) = YES
Clique(G, 3) = NO



Clique(G, 3) = YES
Clique(G, 4) = NO

Clique Verifier

- **Given**: an undirected graph G = (V, E)
- **Problem**: Does G have a clique of size k?
- **Certificate**:
  - A set of k nodes



- **Verifier**:
  - Verify that for all pairs of vertices in this set there exists an edge in E

CLIQUE is NP-complete

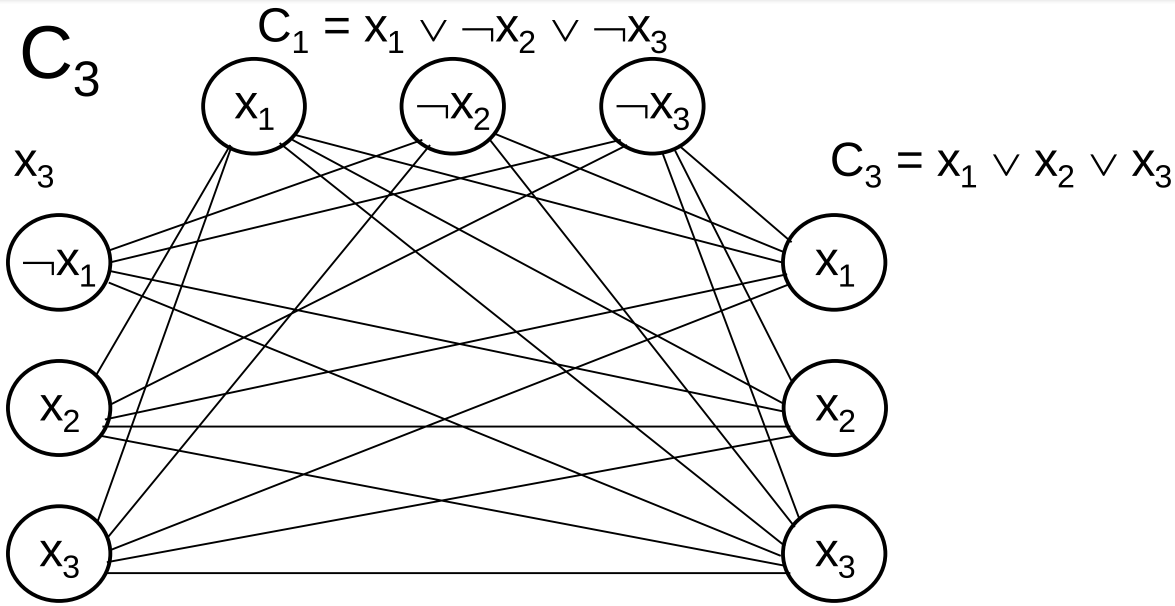
- *Theorem 34.11*: (page 1003)
  - CLIQUE problem is NP-complete.
- Proof:
  - **CLIQUE $\in$NP:**
    - given G=(V,E) and a set V'$\subseteq$V as a certificate for G. The verifying algorithm checks for each pair of $u,v\in$V', whether <u,v> $\in$E. time: $O(|V'|^2|E|)$.
  - **CLIQUE is NP-hard:**
    - show 3-CNF-SAT $\leq_p$CLIQUE.
    - The result is surprising, since from boolean formula to graph.

3-CNF $\leq_p$ Clique

- Start with an instance of 3-CNF:
  - $\Phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ (k clauses)
  - Each clause C_r has three literals: $C_r = l_1^r \vee l_2^r \vee l_3^r$
- **Idea:**
  - Construct a graph G such that Φ is satisfiable only if G has a clique of size k

3-CNF $\leq_p$ Clique

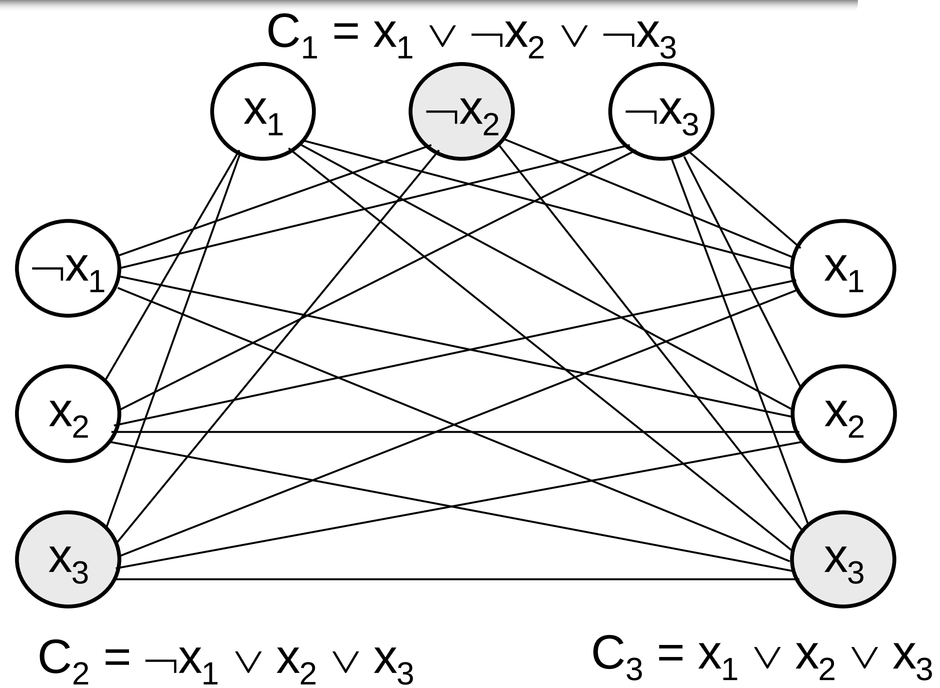
$\Phi = C_1 \wedge C_2 \wedge C_3$



- For each clause $C_r = l_1^r \vee l_2^r \vee l_3^r$ place a triple of vertices v_1^r, v_2^r, v_3^r in V
- Put an edge between two vertices v_i^r and v_j^s if:
  - v_i^r and v_j^s are in different triples
  - l_i^r is not the negation of l_j^s (consistent correspondent literals)

3-CNF $\leq_p$ Clique

$\Phi = C_1 \wedge C_2 \wedge C_3$

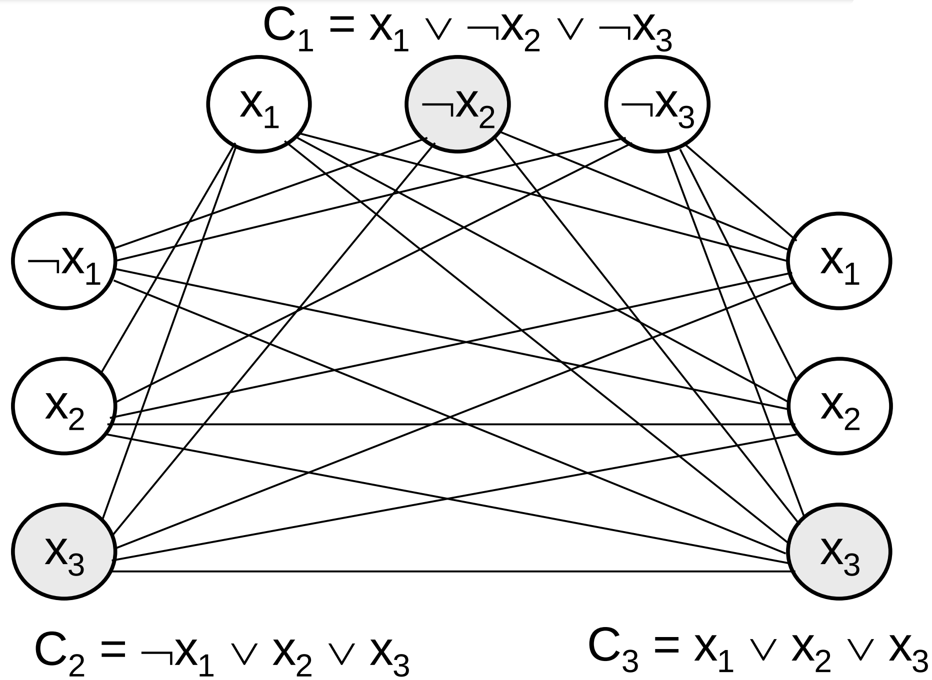


- Suppose Φ has a satisfying assignment
  - Each clause C_r has a literal assigned to 1 – this corresponds to a vertex v_i^r
  - Picking one such literal from each $C_r \Rightarrow$ a set V' of k vertices
- Claim: V' is a clique
  - $\forall\ v_i^r, v_j^s \in$ V' the corresponding literals are 1 $\Rightarrow$ cannot be complements
  - by the design of G the edge $(v_i^r, v_j^s) \in E$

3-CNF $\leq_p$ Clique

$\Phi = C_1 \wedge C_2 \wedge C_3$

- Suppose *G* has a clique of size k
  - No edges between nodes in the same clause
  - Clique contains only one vertex from each clause
  - Assign 1 to vertices in the clique
  - The literals of these vertices cannot belong to complementary literals
  - Each clause is satisfied $\Rightarrow$ Φ is satisfied



NP-completeness proof structure

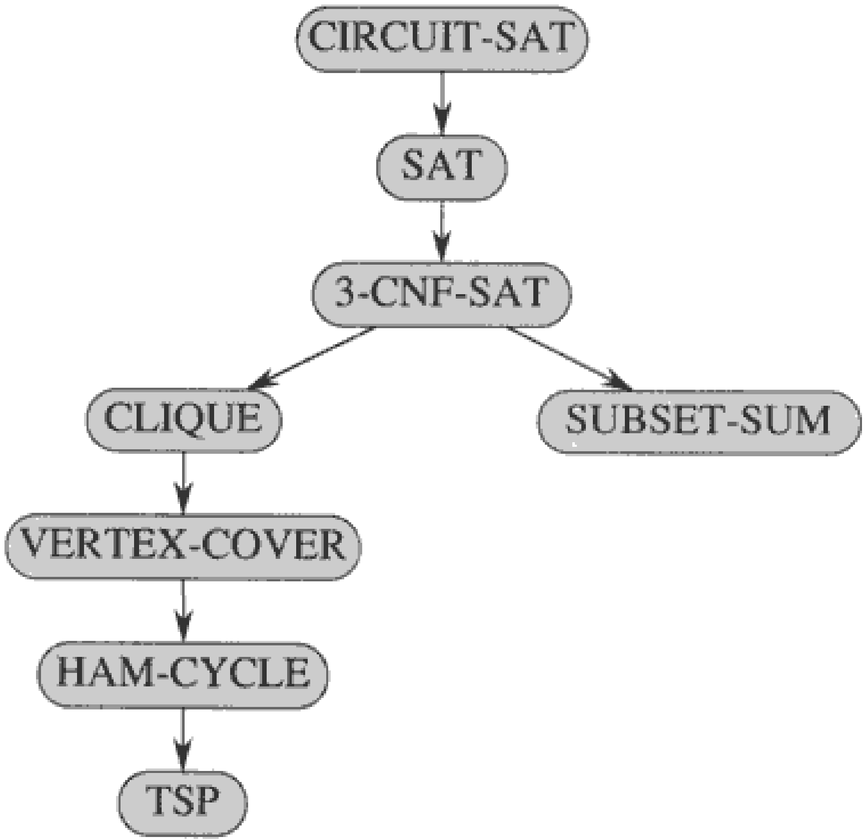
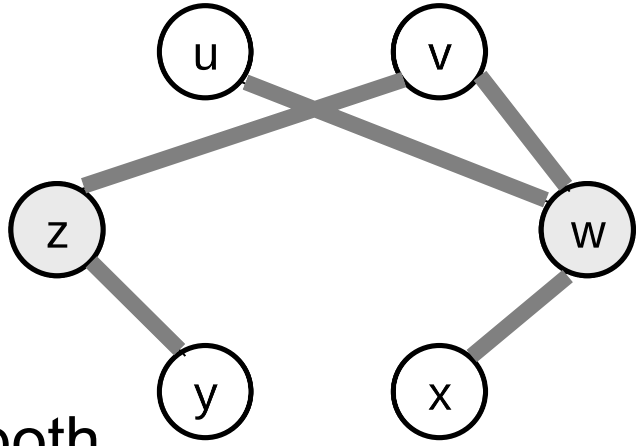


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

Vertex Cover

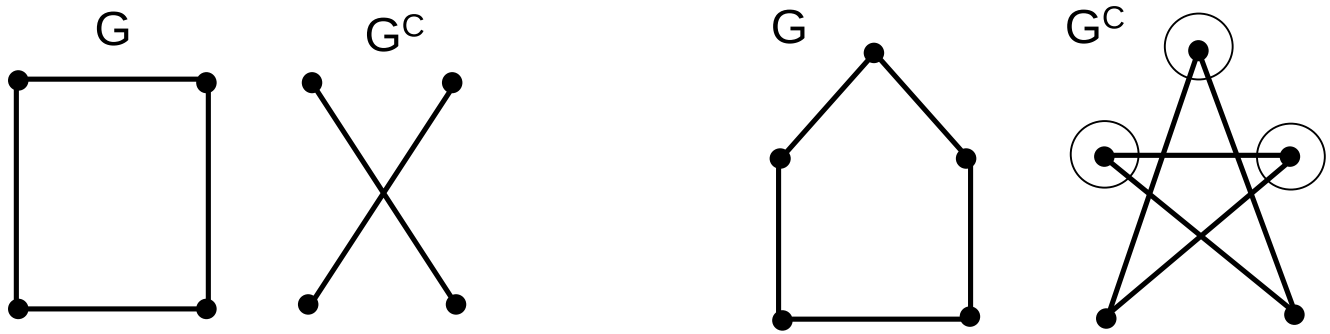


- G = (V, E), undirected graph
- **Vertex cover** = a subset V' ⊆ V such that covers all the edges
  - if (u, v) ∈ E then u ∈ V' or v ∈ V' or both.
- **Size** of a vertex cover = number of vertices in it

Problem:

- Find a vertex cover of minimum size
- Does graph G have a vertex cover of size k?

Clique $\leq_p$ Vertex Cover

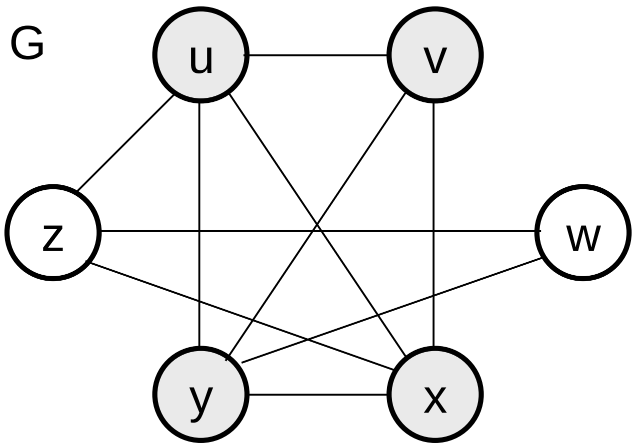


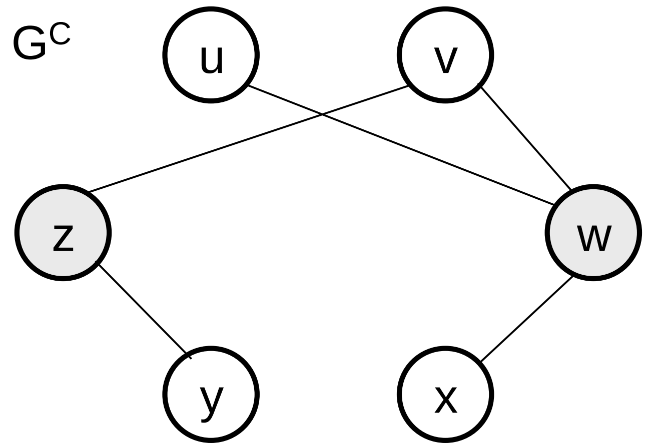
Clique = 2 VxCv = 2 Clique = 2 VxCv = 3

$$\text{Size[Clique](G)} + \text{Size[VxCv]}(G^C) = n$$

- G has a **clique** of size k $\Leftrightarrow$ G^C has a **vertex cover** of size n – k
- S is a clique in G $\Leftrightarrow$ V – S is a vertex cover in G^C

Clique $\leq_p$ Vertex Cover





- $G = (V, E) \Rightarrow G^C = (V, E^C)$

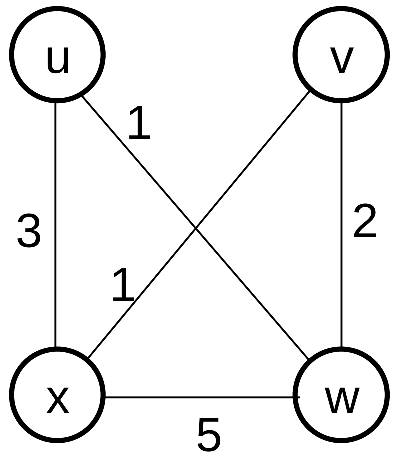
  $E^C = \{(u, v):, u, v \in V, \text{ and } (u, v) \notin E\}$

Idea:

$\langle G, k\rangle$ (clique) $\rightarrow \langle G^C, |V|\text{-}k\rangle$ (vertex cover)

The Traveling Salesman Problem

- $G = (V, E)$, $|V| = n$, vertices represent cities
- **Cost**: $c(i, j)$ = cost of travel from city i to city j
- **Problem**: salesman should make a tour (hamiltonian cycle):
  - Visit each city only once
  - Finish at the city he started from
  - Total cost is minimum
- TSP = tour with cost at most k



$\langle u, w, v, x, u \rangle$

Traveling-salesman problem is NPC

- TSP={<G,*c*,*k*>:

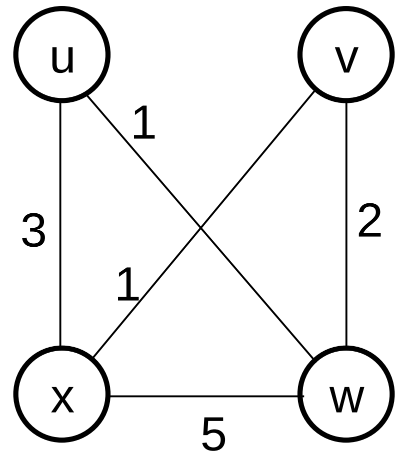
  G=(V,E) is a complete graph,

  c is a function from $V \times V \to Z$,

  $k \in Z$, and G has a traveling salesman tour with cost at most *k*.}
- *Theorem 34.14*: (page 1012)
  - TSP is NP-complete.

TSP $\in$ NP

- **Certificate:**
  - Sequence of n vertices, cost
  - E.g.: $\langle$u, w, v, x, u$\rangle$, 7
- **Verification:**
  - Each vertex occurs only once
  - Sum of costs is at most k



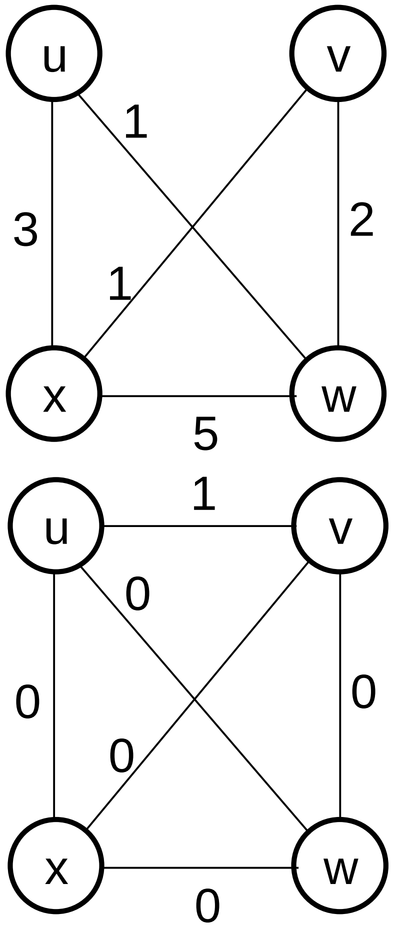
HAM-CYCLE $\leq_p$ TSP

- Start with an instance of Hamiltonian cycle G = (V, E)
- Form the complete graph G' = (V, E')

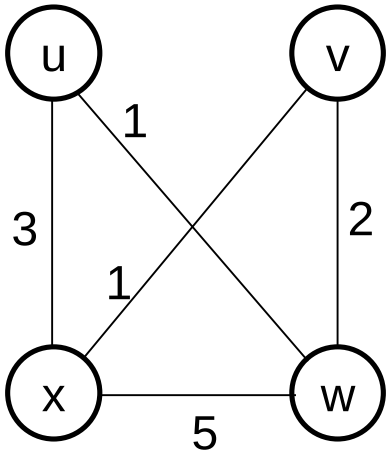
$$E' = \{(i, j): i, j \in V \text{ and } i \neq j\}$$

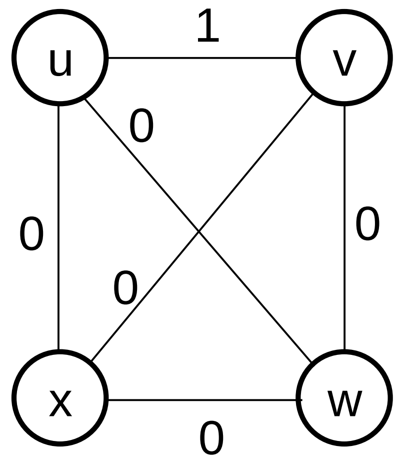
$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \\ 1 & \text{if } (i, j) \notin E \end{cases}$$

- TSP: $\langle G', c, 0 \rangle$
- G has a hamiltonian cycle $\Leftrightarrow$ G' has a tour of cost at most 0



HAM-CYCLE $\leq_p$ TSP

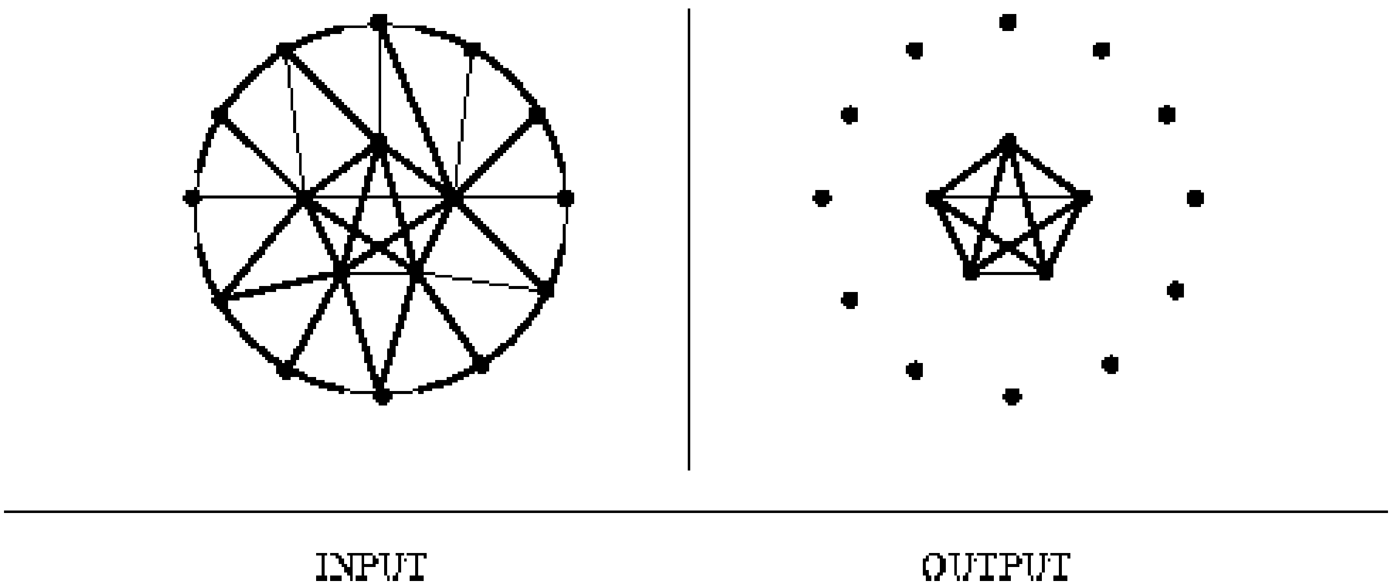




- G has a hamiltonian cycle h
  - $\Rightarrow$ Each edge in $h \in E \Rightarrow$ has cost 0 in G'
  - $\Rightarrow$ h is a tour in G' with cost 0
- G' has a tour h' of cost at most 0
  - $\Rightarrow$ Each edge on tour must have cost 0
  - $\Rightarrow$ h' contains only edges in E

Some popular NP-complete problems

Clique

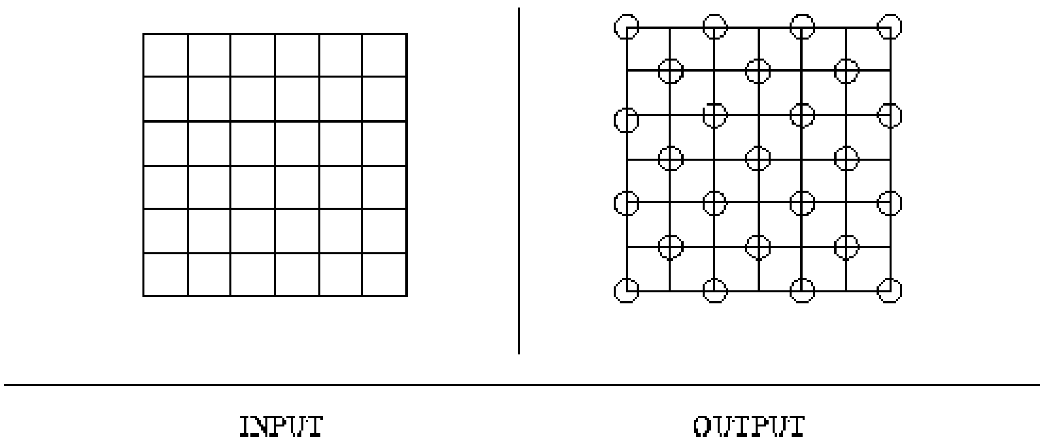


Input description: A graph $G=(V,E)$.

Problem description: What is the largest $S \subset V$ such that for all $x, y \in S$, $(x, y) \in E$?

Some popular NP-complete problems

Independent Set

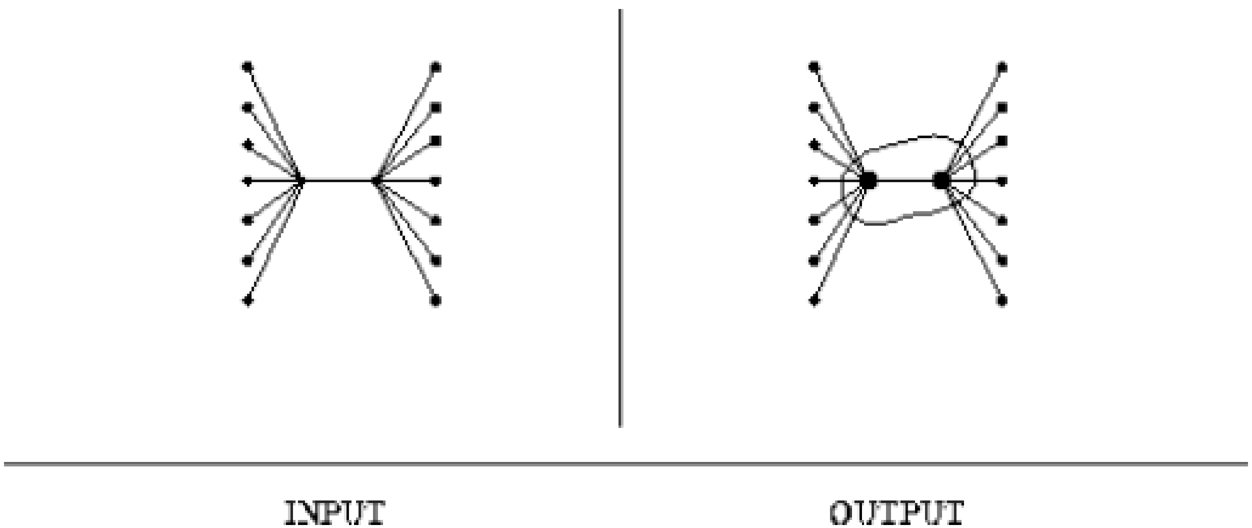


Input description: A graph $G=(V,E)$.

Problem description: What is the largest subset S of vertices of V such that no pair of vertices in S defines an edge of E between them?

Some popular NP-complete problems

Vertex Cover

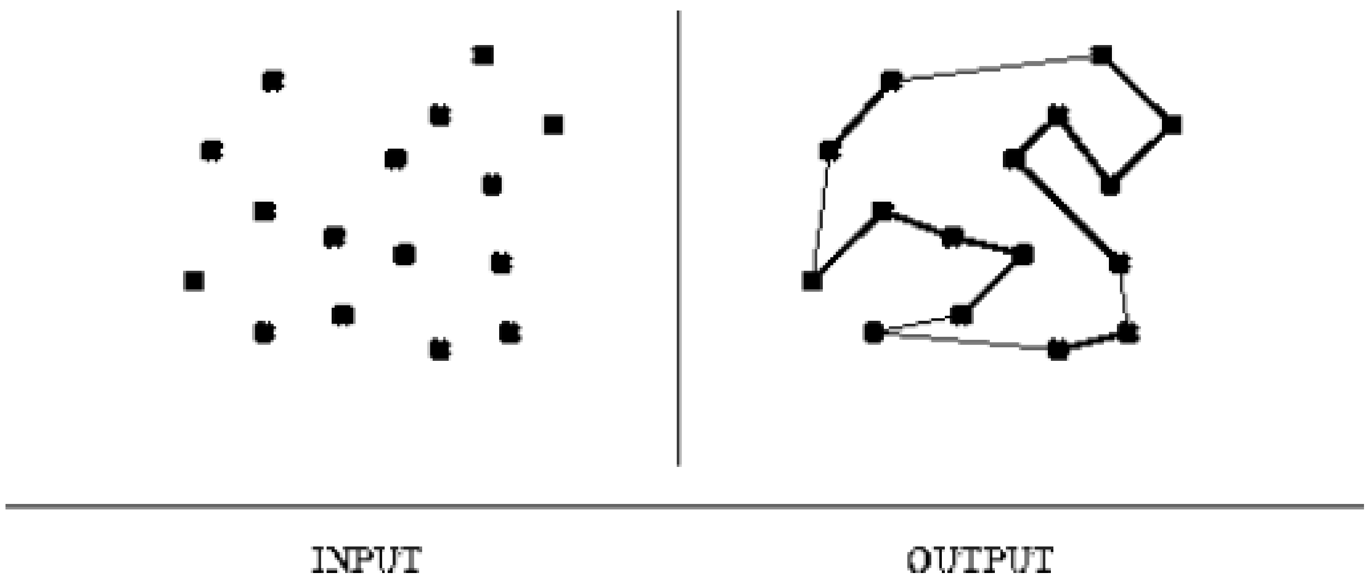


Input description: A graph $G=(V,E)$.

Problem description: What is the smallest subset of $S \subset V$ such that each $e \in E$ contains at least one vertex of S?

Some popular NP-complete problems

Traveling Salesman Problem

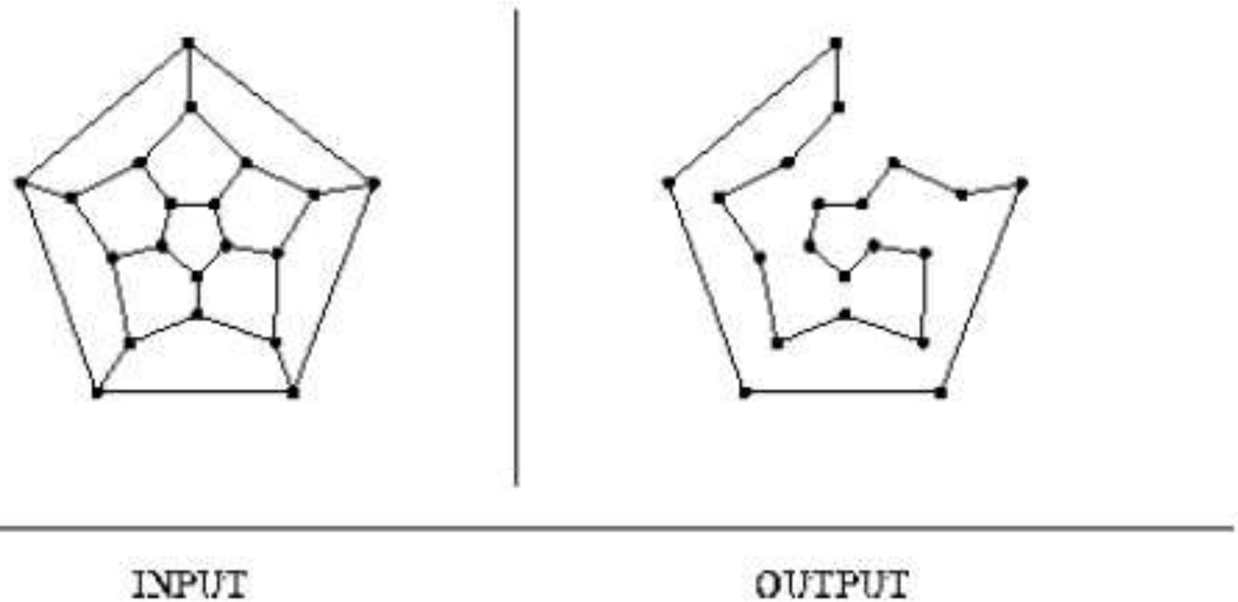


Input description: A weighted graph *G*.

Problem description: Find the cycle of minimum cost that visits each of the vertices of *G* exactly once.

Some popular NP-complete problems

Hamiltonian Cycle

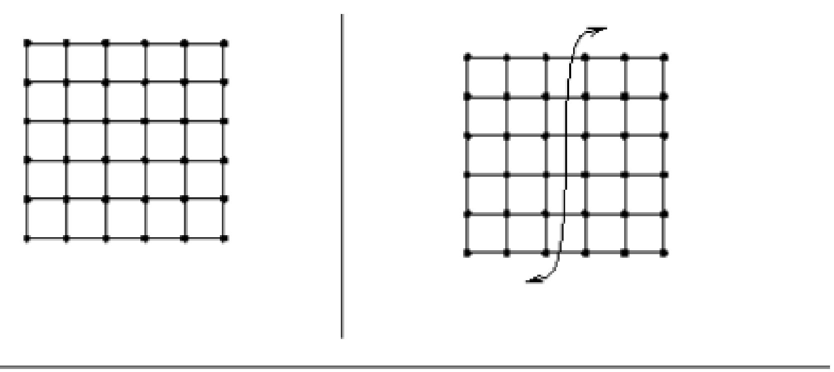


Input description: A graph $G = (V,E)$.

Problem description: Find an ordering of the vertices such that each vertex is visited exactly once.

Some popular NP-complete problems

Graph Partition



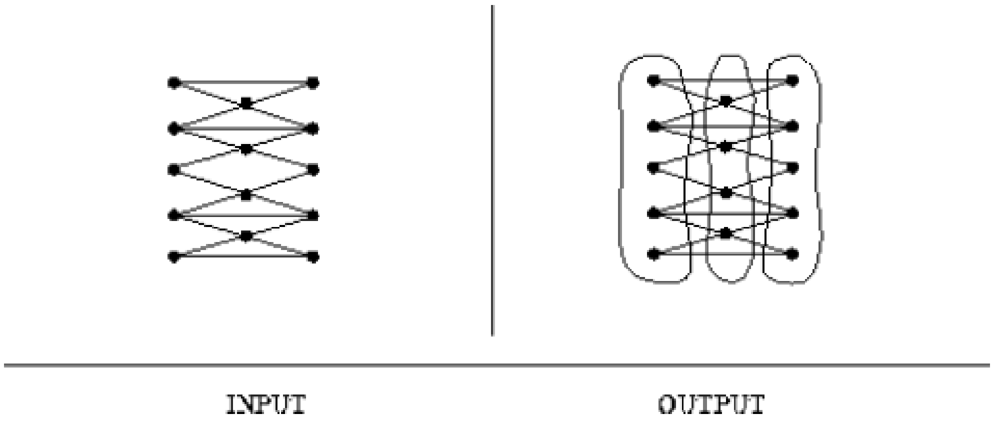
INPUT OUTPUT

Input description: A (weighted) graph $G=(V,E)$ and integers j, k, and m.

Problem description: Partition the vertices into m subsets such that each subset has size at most j, while the cost of the edges spanning the subsets is bounded by k.

Some popular NP-complete problems

Vertex Coloring

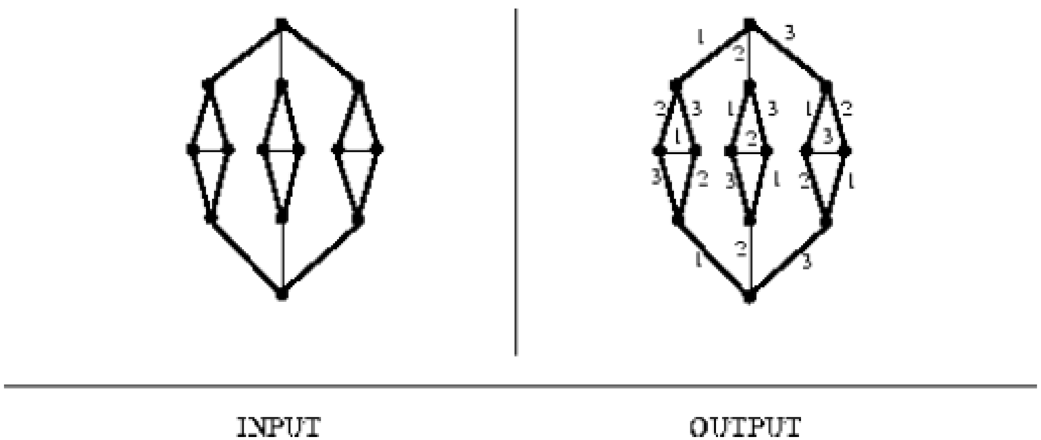


Input description: A graph $G=(V,E)$.

Problem description: Color the vertices of V using the minimum number of colors such that for each edge $(i,j) \in E$, vertices i and j have different colors.

Some popular NP-complete problems

Edge Coloring

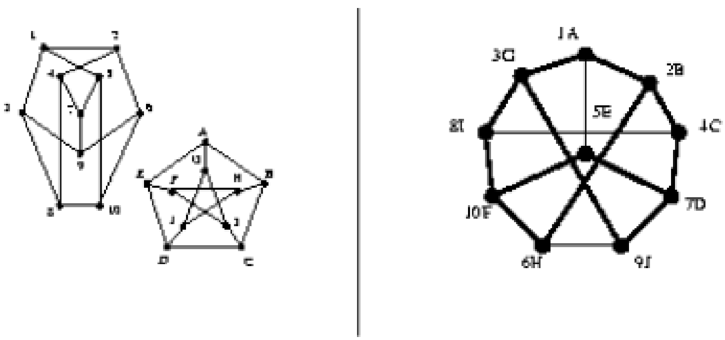


Input description: A graph $G=(V,E)$.

Problem description: What is the smallest set of colors needed to color the edges of E such that no two same-color edges share a vertex in common?

Some popular NP-complete problems

Graph Isomorphism



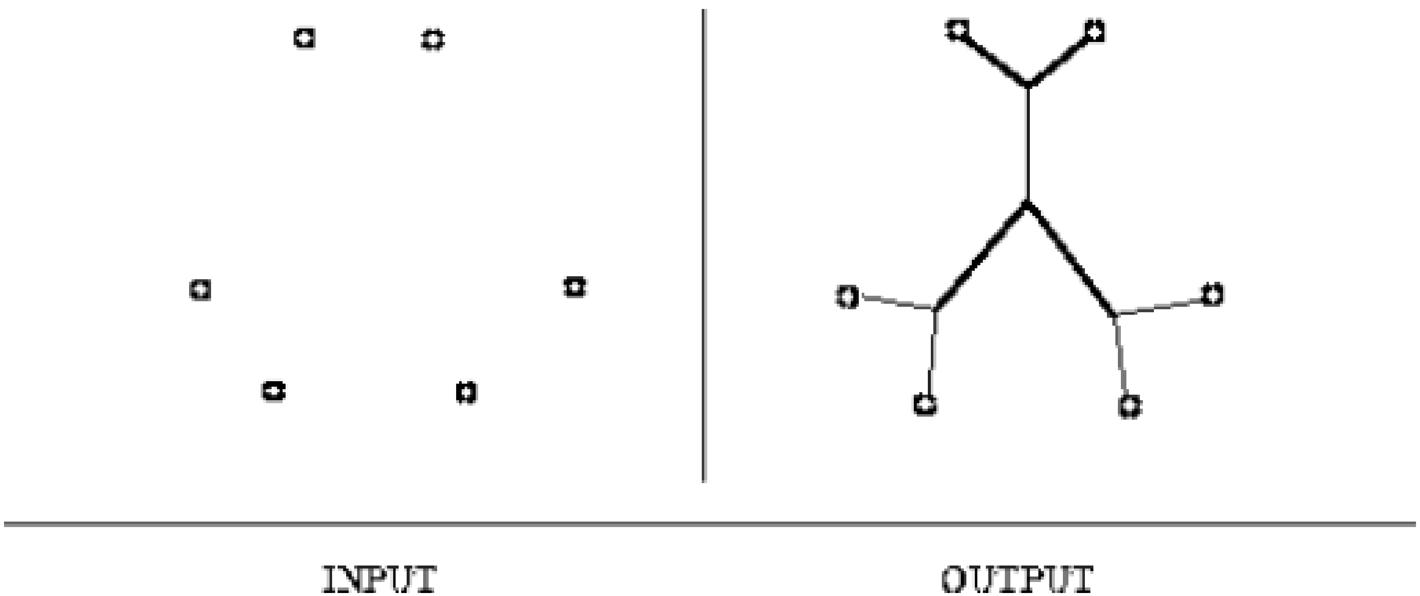
INPUT OUTPUT

Input description: Two graphs, *G* and *H*.

Problem description: Find a (or all) mappings *f* of the vertices of *G* to the vertices of *H* such that *G* and *H* are identical; i.e. (x,y) is an edge of *G* iff $(f(x),f(y))$ is an edge of *H*.

Some popular NP-complete problems

Steiner Tree

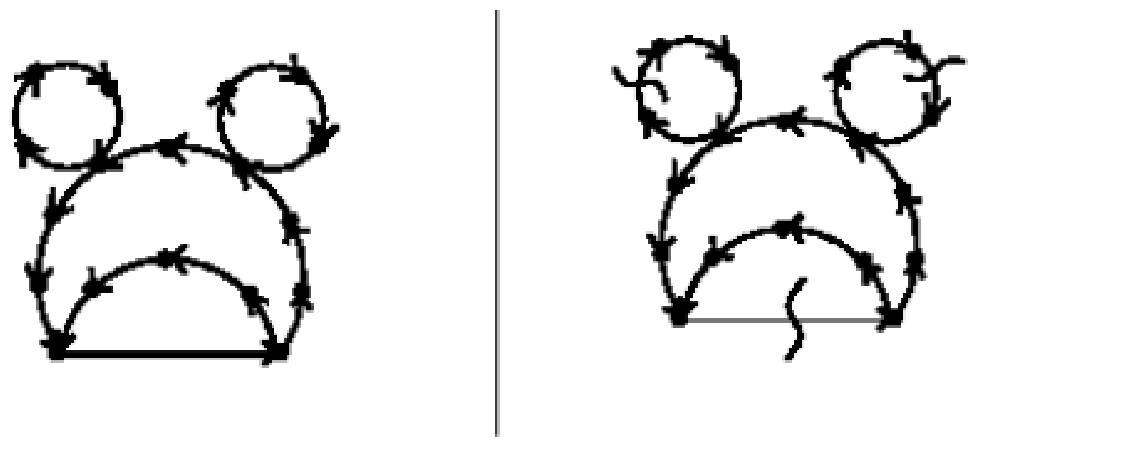


Input description: A graph $G=(V,E)$. A subset of vertices $T \in V$.

Problem description: Find the smallest tree connecting all the vertices of T.

Some popular NP-complete problems

Feedback Edge/Vertex Set



INPUT OUTPUT

Input description: A (directed) graph $G=(V,E)$.

Problem description: What is the smallest set of edges E' or vertices V' whose deletion leaves an acyclic graph?

Readings

- Chapter 34