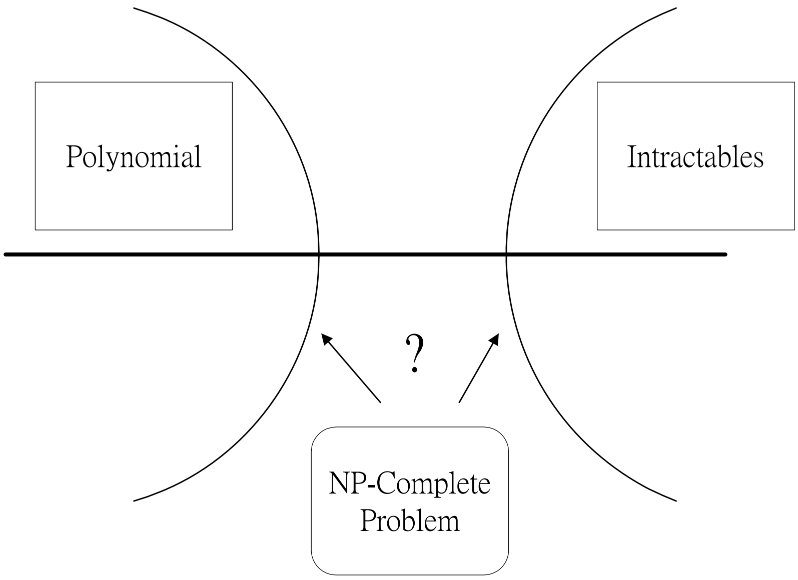
34.NP Completeness

Hsu, Lih-Hsing

Polynomial time algorithms: on inputs of size n, their worst-case running time is $O(n^k)$.

It is natural to wonder whether all problems can be solved in polynomial time. The answer is no. For example, the ***Halting Problem***.

Generally, we think of problems that are solvable by polynomial-time algorithms are being tractable, and problems that requires superpolynomial time are being intractable.



The subject of this chapter, however, is an interesting class of problems, called the “ NP-complete” problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an NP-computer problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. This so-called P ≠ NP question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

NP-complete problem: status are unknown.

If any single NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial time algorithm.

To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness.

The difference between these problems

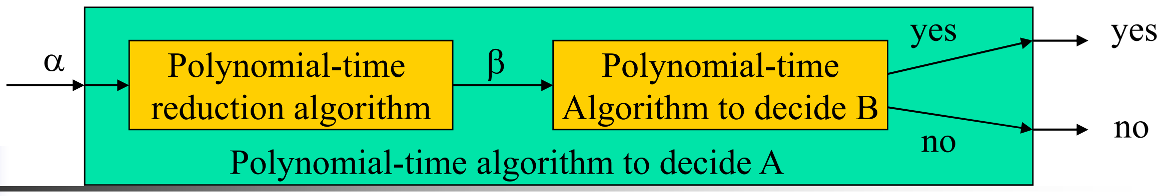
- **Shortest vs. longest simple paths:**
- **Euler tour vs. hamiltonian cycle:**
- **2-CNF satisfiability vs. 3 CNF satisfiablility**
- **NP-completeness and the classes P and NP**
- **Overview of showing problems to be NP-complete**
- **Decision problems vs. optimization problems**

Reductions

Suppose that there is a different decision problem, say B, that we already know how to solve in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

1. The transformation takes polynomial time.
2. The answer are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."

Computer Theory Lab.



We can call such a procedure a polynomial-time reduction algorithm and, it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

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

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, can be solved in one model can be solved in polynomial in another.
- Polynomial-time solvable problems has a nice closure property.

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

Abstract Problems: An abstract problem Q is a binary relation on a set of problem *instances* and a set S of problem *solutions*.

Decision problems: those having yes/no solution.

Optimization problems: recast by imposing a bound on the value to be optimized.

An ***encoding*** of a set S of abstract objects is a mapping e from S to the set of binary string, for example:

$\{0,1,2,3,\ldots\}=\{0,1,10,11,\ldots\}$

We call a problem whose instance sets is the set of binary strings a ***concrete problem***.

We say that an algorithm ***solves*** a concrete problem in time $O(T(n))$ if when it is provided a problem instance i if length $n=|i|$, the algorithm can produce the solution in at most $O(T(n))$ time.

A concrete problem is ***polynomial-time solvable*** if there exists an algorithm to solve it in time $O(n^k)$ for some constant k.

The ***complexity class P*** is the set of concrete decision problems that are solvable in polynomial time.

Abstract problem $\rightarrow$ concrete problem

$$e: I \xrightarrow[\text{encoding}]{} \{0,1\}^*$$

Problem	input k	complexity O(k)
unary	$k \rightarrow 11..1$	$\Theta(k)$
binary	$n = \lfloor \lg k \rfloor$	$\Theta(k) = \Theta(2^n)$

We say that a function $f:\{0,1\}^* \to \{0,1\}^*$ is ***polynomial-time computable*** if there exists a polynomial-time algorithm A that given any $x \in \{0,1\}^*$, produces as output $f(x)$.

For any set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomial related*** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

Lemma 34.1. Let Q be an abstract decision problem on an instance set I, let e_1 and e_2 be polynomially related encodings on I. Then $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.

Using ***reasonable encoding*** to neglect the distinction between abstract and concrete problems.

A formal-language framework

- An ***alphabet*** Σ is a finite set of symbols.
- A ***language*** L over Σ is any set of strings made up of symbols from Σ.
- ***empty string:*** ε.
- ***empty language:*** ϕ.
- Σ^*
- Let L_1, L_2 be two languages. We can define

$L_1 \cup L_2$ (union)

$L_1 \cap L_2$ (intersection)

$\overline{L}$ (complement)

$L_1 L_2 = \{x_1 x_2 \mid x_1 \in L_1 \, and \, x_2 \in L_2\}$

(concatenation)

The closure (Kleen star) of L:

$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$

Computer Theory Lab.

The set of instances of any decision problem Q is the set of Σ^*, where $\Sigma = \{0,1\}$. Since Q is entirely characterized by those problem instances that produces a 1 (yes) answer. We can view Q as the language L over Σ^*, where $L = \{x \in \Sigma^* \mid Q(x) = 1\}$.

Algorithm A ***accepts*** a string $x \in \{0,1\}^*$ if the given input x, the algorithm output $A(x)=1$.

The language ***accepts by an algorithm*** A is the set $L = \{x \in \Sigma^* \mid A(x) = 1\}$.

The algorithm A ***rejects*** a string x if $A(x)=0$.

Even if language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm may loop forever.

A language L is ***decided*** by an algorithm A if every binary string is either accepted or rejected by the algorithm.

A language L is ***accepted in polynomial time*** by an algorithm A if for any length n string $x \in L$, the algorithm accepts x in time $O(n^k)$ for some constant k.

A language L is ***decided in polynomial time*** by an algorithm A if for any length n string $x \in \{0,1\}^*$, the algorithm decides x in time $O(n^k)$ for some constant k.

Example:

PATH PROBLEM:

PATH=$\{<G,u,v,k> \mid G=(V,E)$ is an undirected graph, $u,v \in V, k \geq 0$ is an integer, and there is a path from u to v whose length is at most $k\}$.

- Can be accepted in polynomial time.
- Can be decided in polynomial time.

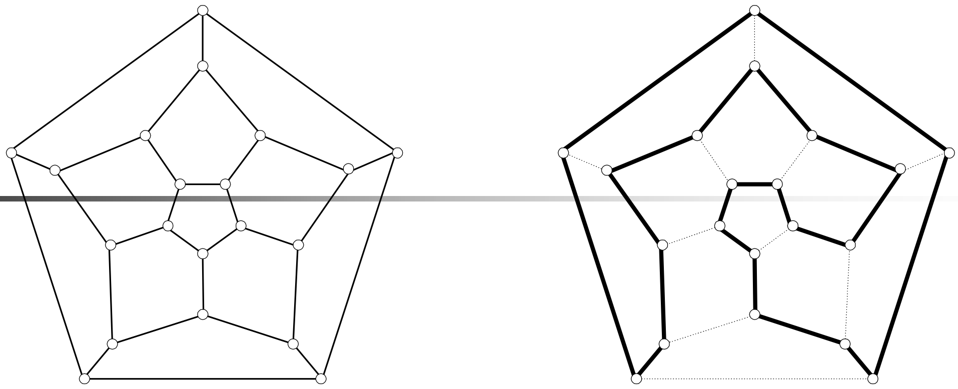
HALTING PROBLEM:

There exists an accepting algorithm, but no decision algorithm exists.

We can informally define a ***complexity class*** as a set of languages, membership in which is determined by a ***complexity measure***, such as running time, on an algorithm that determines whether a string x belongs to language L.

We define the complexity class P as: $P = \{L \subseteq \{0,1\}^* \mid$ there exists an algorithm A that decides L in polynomial time$\}$.

Theorem 34.2. $P = \{L \mid L$ is accepted by a polynomial algorithm$\}$.



HAMILTONIAN CYCLE PROBLEM:

HAM_CYCLE={<G> | G is a hamiltonian graph}

verification: polynomial

decision problem: ?

34.2 Polynomial-time verification

PATH PROBLEM:

PATH={<*G*,*u*,*v*,*k*> | *G*=(*V*,*E*) is an undirected graph, $u,v \in V, k \geq 0$ is an integer, and there is a path from *u* to *v* whose length is at most *k*}.

verification: linear time.

Decision problem: polynomial

naive algorithm:

input size: If we use the reasonable encoding of a graph as its adjacency matrix, the number *m* of vertices is $\Omega(\sqrt{n})$, where n = |<G>| is the length of the encoding of G. There are m! possible permutations of the vertices. Therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$. This is not a polynomial algorithm.

Verification algorithms:

A ***verification algorithm*** is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***. A two-argument algorithm A ***verifies*** an input x if there exists a certificate y such that $A(x,y)=1$. The ***language verified*** by a verification algorithm A is

$$L=\{x\in\{0,1\}^* \mid \exists y\in\{0,1\}^*\ s.t.\ A(x,y)=1\}.$$

The complexity class NP

The ***complexity class NP*** is the class of languages that can be verified by a polynomial-time algorithm. More precise, a language L belongs to NP if and only if there exists a two-input polynomial-time algorithm A and a constant c such that

$L=\{x\in\{0,1\}^* \mid$ there exists a certificate y with $|y|=O(|x|^c)$ such that $A(x,y)=1\}$.

- $NP\neq\phi$ (HAM_CYCLE $\in$ NP.)
- $P\subseteq NP$.

Problem:

1. $P \neq NP?$

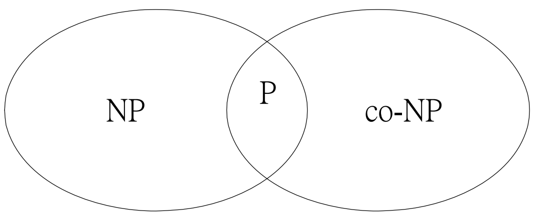
2. Complexity class co-NP

$$co-NP = \{ L \mid \overline{L} \in NP \}.$$

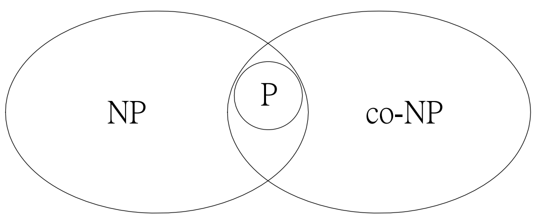
$$NP = co-NP?$$

3. Obviously $P \subset NP \cap co-NP$.

$$P = NP \cap co-NP?$$



$P = NP \cap co-NP$



$P \subset NP \cap co-NP$

34.3 NP-completeness and reducibility

NP-completeness problem: if any one NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time solution, that is NP=P.

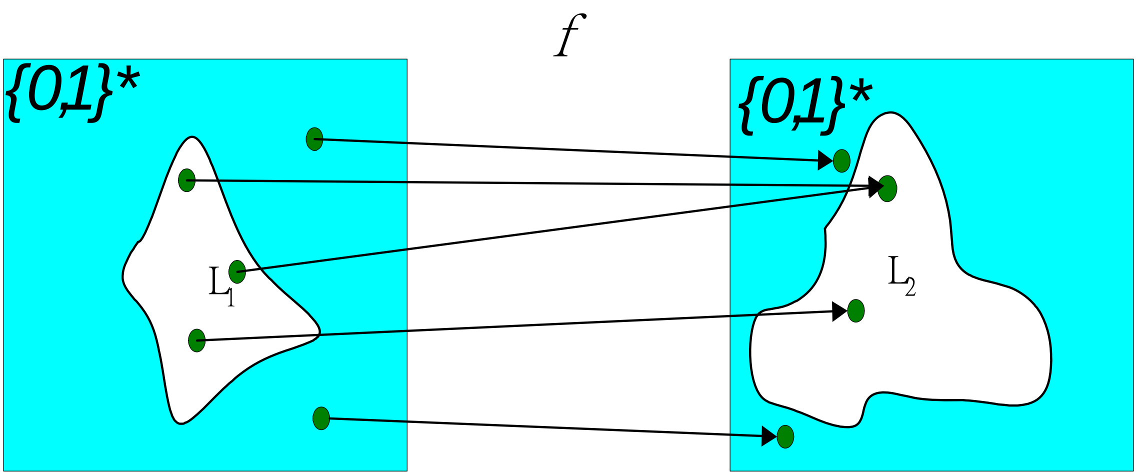
Reducibility:

$$ax + b = 0$$

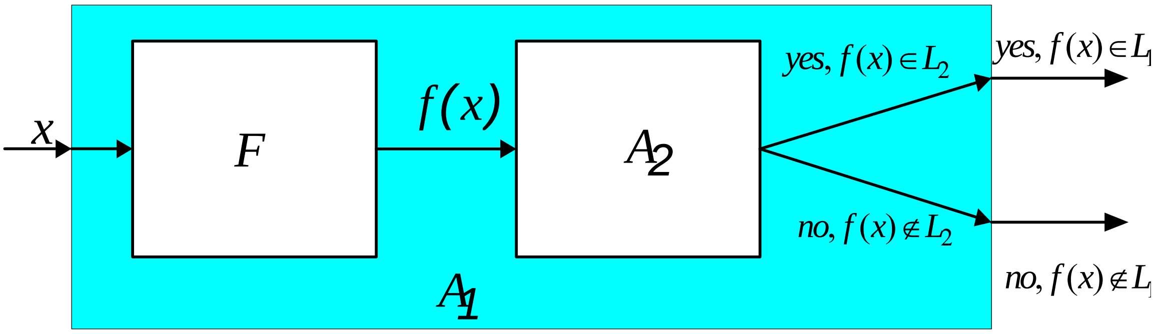
$$ax^2 + bx + c = 0$$

A language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_P L_2$ if there exists a polynomial-time computable function $f:\{0,1\}^* \to \{0,1\}^*$ such that for all $x \in L_1$ if and only if $f(x) \in L_2$. We call the function f the ***reduction function***, and a polynomial algorithm F that computes f is called a ***reduction algorithm***.

Computer Theory Lab.



Lemma 34.3. If L_1, $L_2 \in \{0,1\}^*$ are languages such that $L_1 \le_P L_2$, then $L_2 \in P$ implies $L_1 \in P$.



NP-Completeness

A language $L \in \{0,1\}^*$ is ***NP-complete*** if

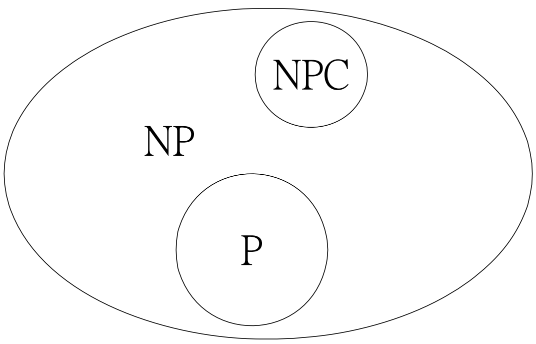
1. $L \in NP$, and
2. $L' \le_P L$ for every $L' \in NP$

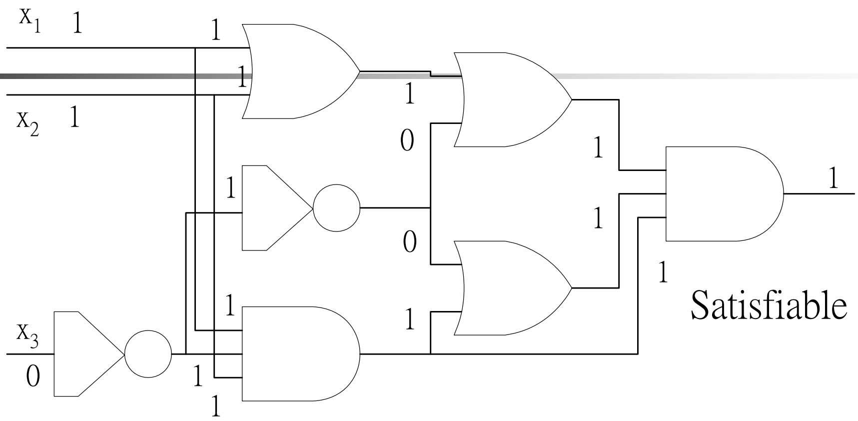
- If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard***.

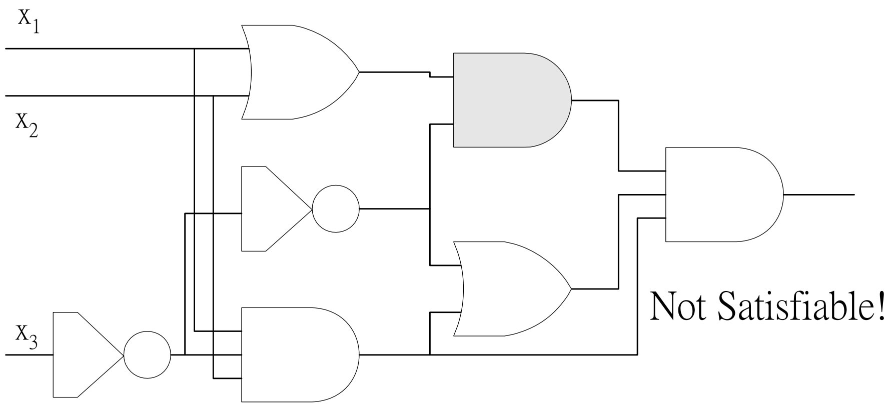
- We also define ***NPC*** to be the class of NP-complete language.

Theorem 34.4. If any NP-complete problem is polynomial-time solvable, then NP=P. If any problem is not polynomial-time solvable, then all NP-complete problem are not polynomial-time solvable.

Proof. By Lemma 34.3.







Circuit-satisfiability problem: Given a boolean combinational circuits composed of AND, OR, or NOT gates, is it satisfiable?

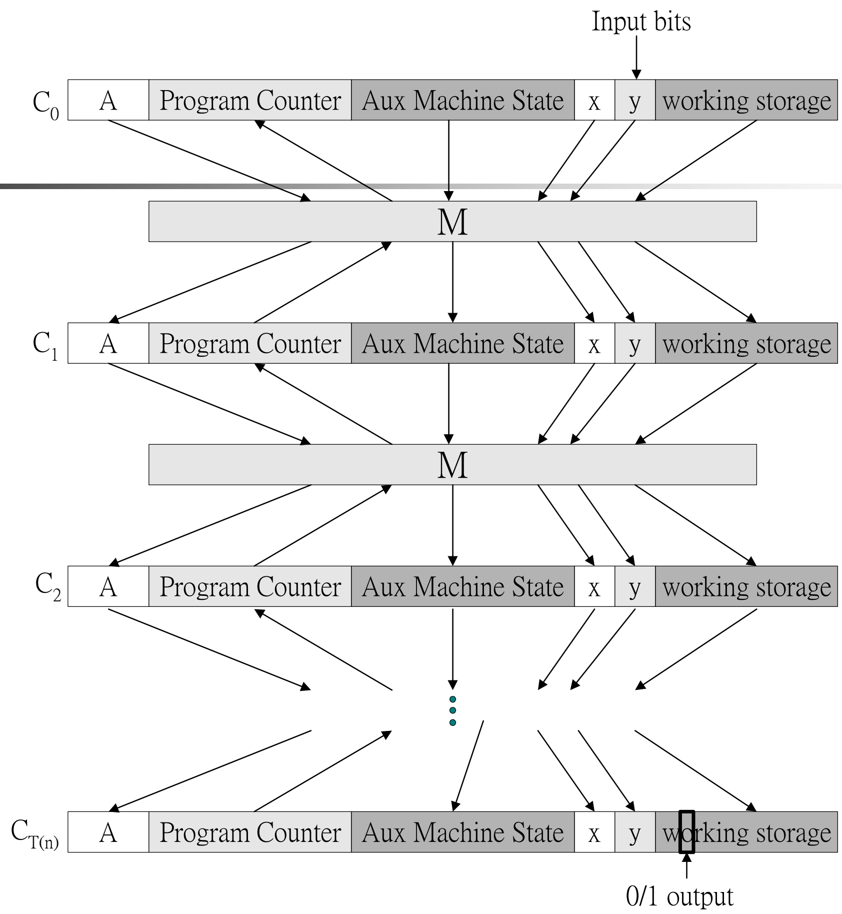
CIRCUIT_SAT={<*C*> | *C* is a satisfiable boolean combinational circuit}.

Lemma 34.5. The circuit-satisfiability problem belongs to the class NP.

Lemma 34.6. The circuit-satisfiability problem is NP-hard.

Proof. $L \leq_P CIRCUIT\_SAT \quad \forall L \in NP.$

Theorem 34.7. The circuit-satisfiability problem is NP-Complete.



34.4NP-Completeness Proof

Lemma 34.8. If L is a language such that $L' \leq_P L$ for some $L' \in NPC$, then L is NP-hard. Moreover, if $L \in NP$ then $L \in NPC$.

Method for proving a language L is NPC:

1. Prove $L \in NP$.
2. Select a known NPC language L'
3. Describe an algorithm that computes a function f mapping every instance of L' to an instance of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
5. Prove that the algorithm computing f runs in polynomial time.

Formula satisfiability:

An instance of SAT is a boolean formula φ composed of

1. boolean variables: $x_1, x_2, \ldots$
2. boolean connectives: any boolean function with one or two input and one output
3. parentheses

SAT={<φ>| φ is a satisfiability formula}

Example:

$$\varphi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

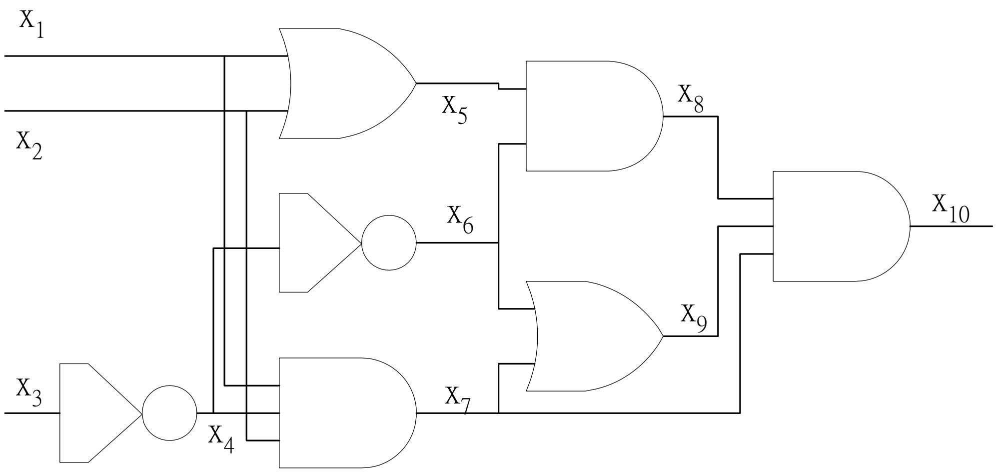
$$x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$$

$$\begin{aligned}\varphi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1\end{aligned}$$

Theorem 34.9 Satisfiability of boolean formula is NP-complete.

Proof.

- $SAT \in NP$
- $CIRCUIT\_SAT \leq_P SAT$



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

3-CNF satisfiability

- literal
- ***conjunction normal form*** (CNF)
- ***3-conjunction normal form*** (3-CNF)

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

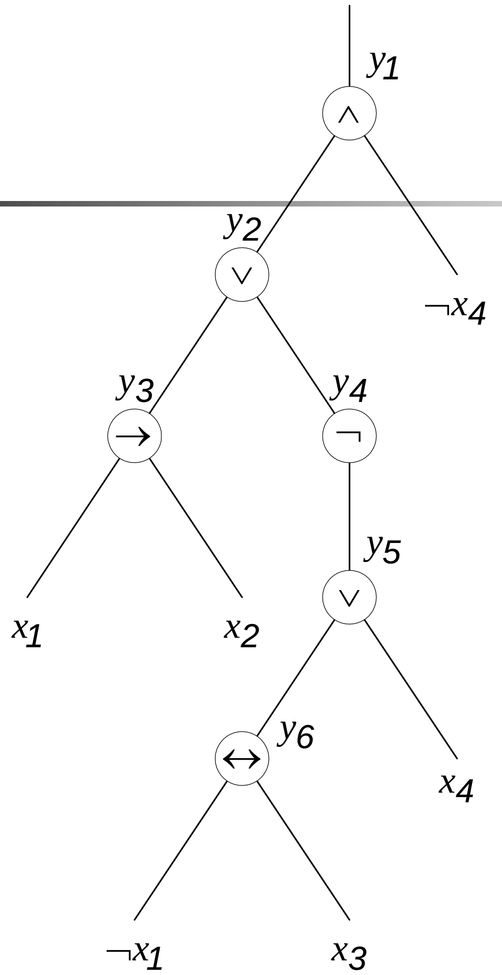
Computer Theory Lab.

Theorem 34.10. Satisfibility boolean formula in 3-CNF is NP complete.

Proof.

- $3-CNF-SAT \in NP$
- $SAT \leq_P 3-CNF-SAT$

$\varphi = ((x_1 \to x_2) \vee ((\neg x_1 \leftrightarrow x_3) \vee x_4) \wedge \neg x_4$



$$\varphi = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_4)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4))$$
$$\wedge (y_3 \leftrightarrow (x_1 \to x_2)) \wedge (y_4 \leftrightarrow \neg y_5)$$
$$\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))$$

$$\varphi_1 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$$

Truth Table $\Downarrow$

$$\neg\varphi_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2)$$
$$\vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

De Morgan rule $\Downarrow$

$$\varphi_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2)$$
$$\wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

$|Ci| = 3 \quad C_i$

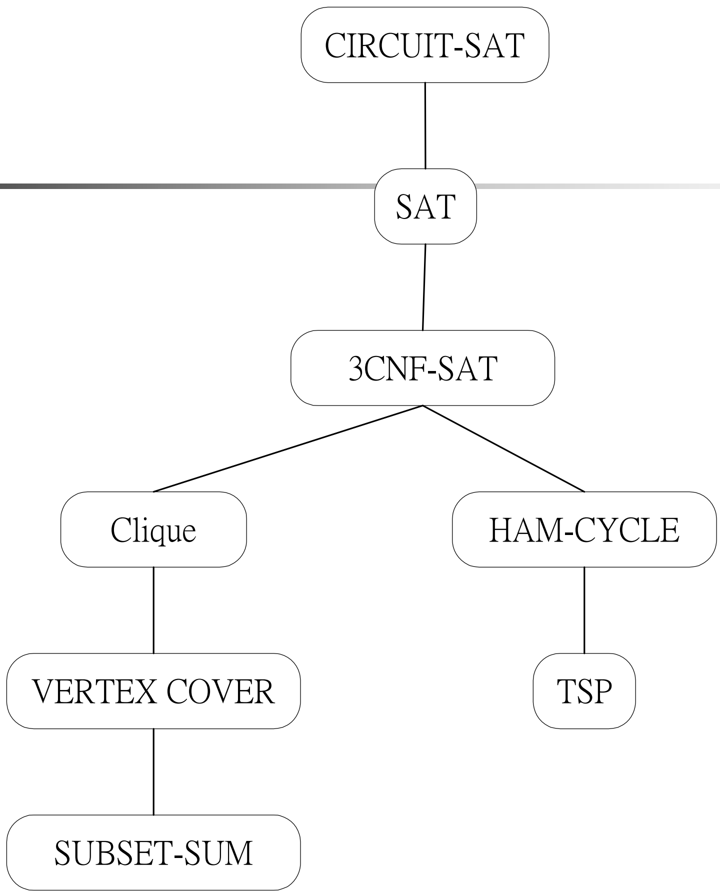
$|Ci| = 2$

$$C_i = l_1 \vee l_2 = (l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$$

$|Ci| = 1$

$$C_i = l = (l \vee p \vee q) \wedge (l \vee p \vee \neg q)$$
$$(l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$$

34.5 NP-Complete Problems



34.5.1 The clique problem

A ***clique*** in a undirected graph $G = (V,E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. The ***size*** of a clique is the number of vertices it contains. The ***clique problem*** is the optimization problem of finding a clique of maximum size in a graph.

CLIQUE={<*G*, *k*> |*G* is a graph with clique size *k*}

naïve algorithm: $\Omega(k^2 \binom{|V|}{k})$

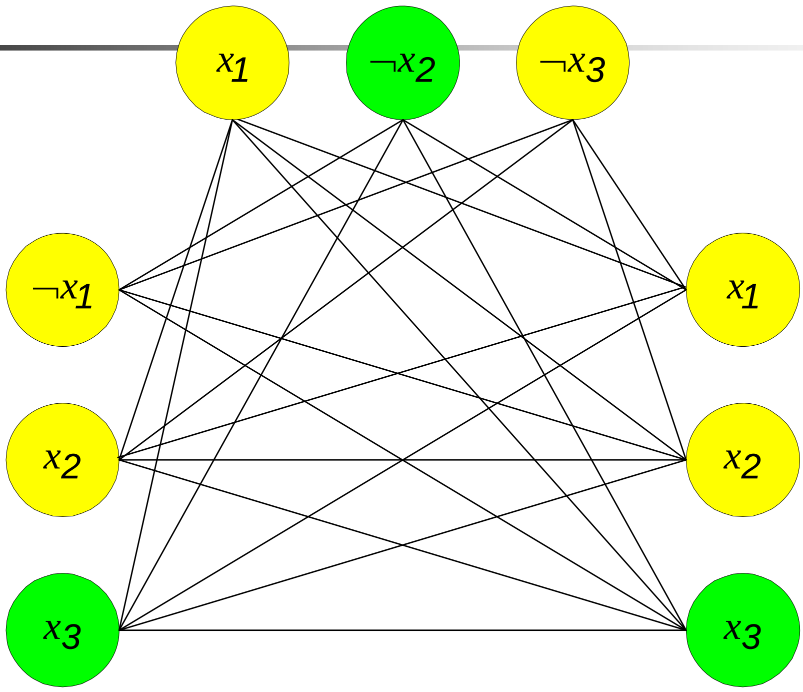
Theorem 34.11. The clique problem is NP-complete.

Proof.

- $clique \in NP$
- $3-CNF-SAT \leq_P clique$

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

$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$



$C_2 = \neg x_1 \vee x_2 \vee x_3$ $C_3 = x_1 \vee x_2 \vee x_3$

- $\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$
- $(v_i^r, v_j^s) \in E \Leftrightarrow \begin{matrix} (1) & r \neq s \\ (2) & l_i^r \neq \neg l_j^s \end{matrix}$
- clique size k

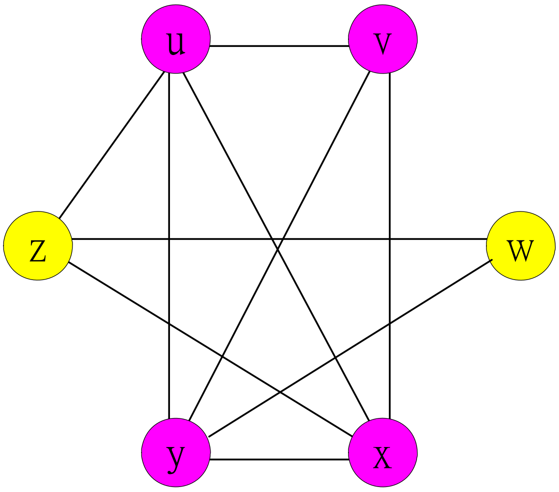
34.5.2 The vertex-cover problem

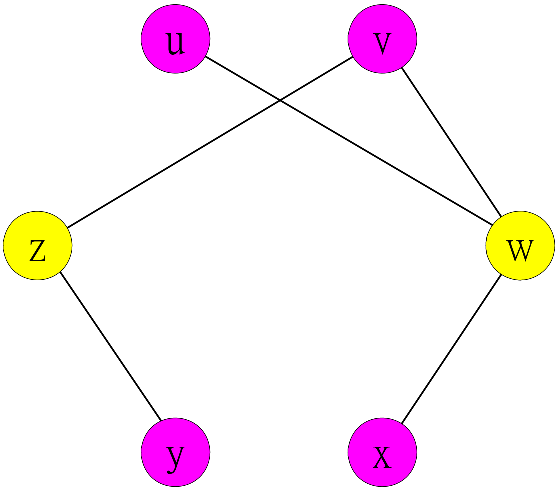
- A ***vertex cover*** of an undirected graph $G=(V,E)$ is a subset $V' \subseteq V$ such that if $(u,v) \in E$ then $u \in V'$ or $v \in V'$ (or both).
- The ***vertex cover problem*** is to find a vertex cover of minimum size in a given graph.
- VERTEX-COVER={<G, k> | graph G has a vertex cover of size k}.

Theorem 34.12. The vertex-cover problem is NP-complete.

Proof.

- $VERTEX-COVER \in NP$
- $CLIQUE \le_P VERTEX-COVER$





34.5.3The hamiltonian-cycle problem

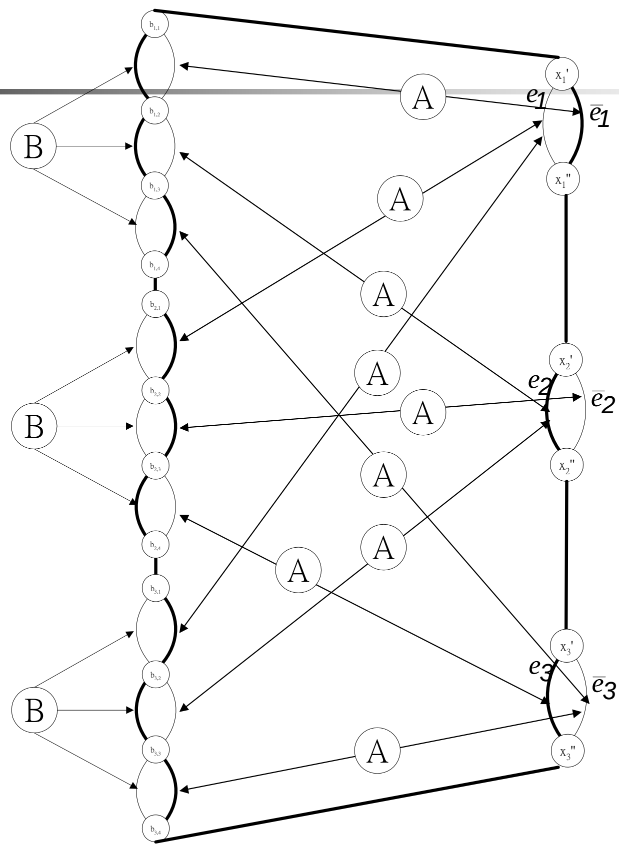
Theorem 34.13. The hamiltonian cycle problem is NP-complete.

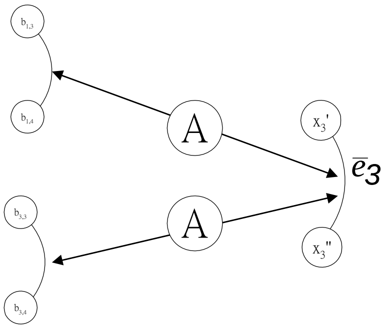
Proof.

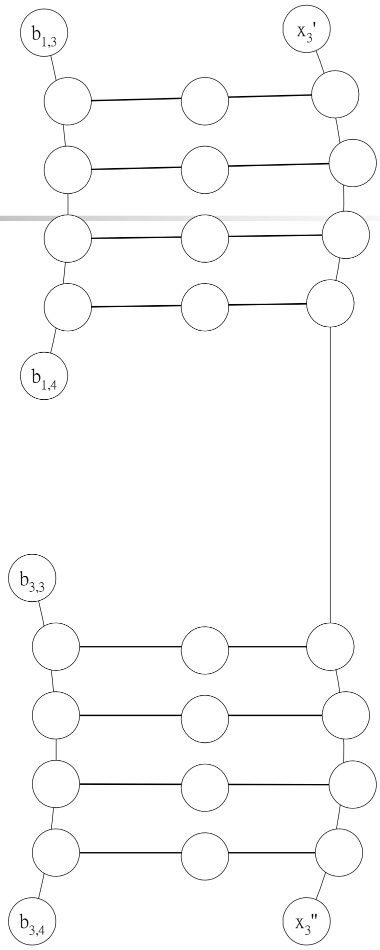
- $HAM - CYCLE \in NP$
- $3CNF - SAT \leq_P HAM - CYCLE$

- kinds of wedges

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

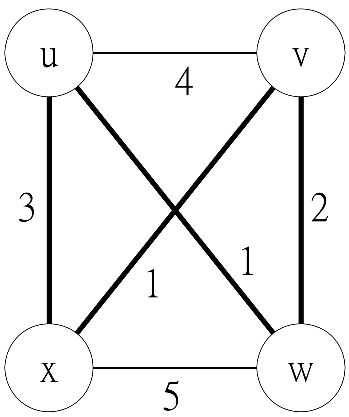






34.5.4 The traveling-salesman problem

TSP={<G, c, k> | G = (V, E) is a complete graph, c is a function from $V \times V$ into Z, $k \in Z$, and G has a traveling salesman tour with cost at most k}.



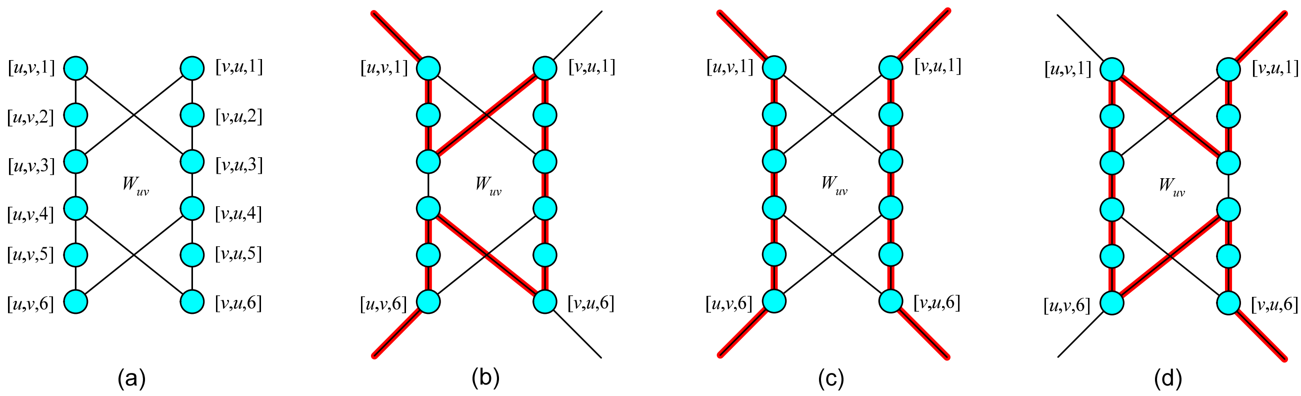
Theorem 34.13

- The hamiltonian cycle problem is NP-complete.

Proof.

- First, show that HAM-CYCLE belongs to NP.
- We now prove that VERTEX-COVER $\leq_p$ HAM-CYCLE, which shows that HAM-CYCLE is NP-complete.
- Given an undirected graph G=(V,E) and an integer k, we construct an undirected graph G'=(V',E') that has a hamiltonian cycle iff G has a vertex cover of size k.

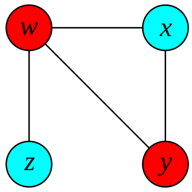
widget



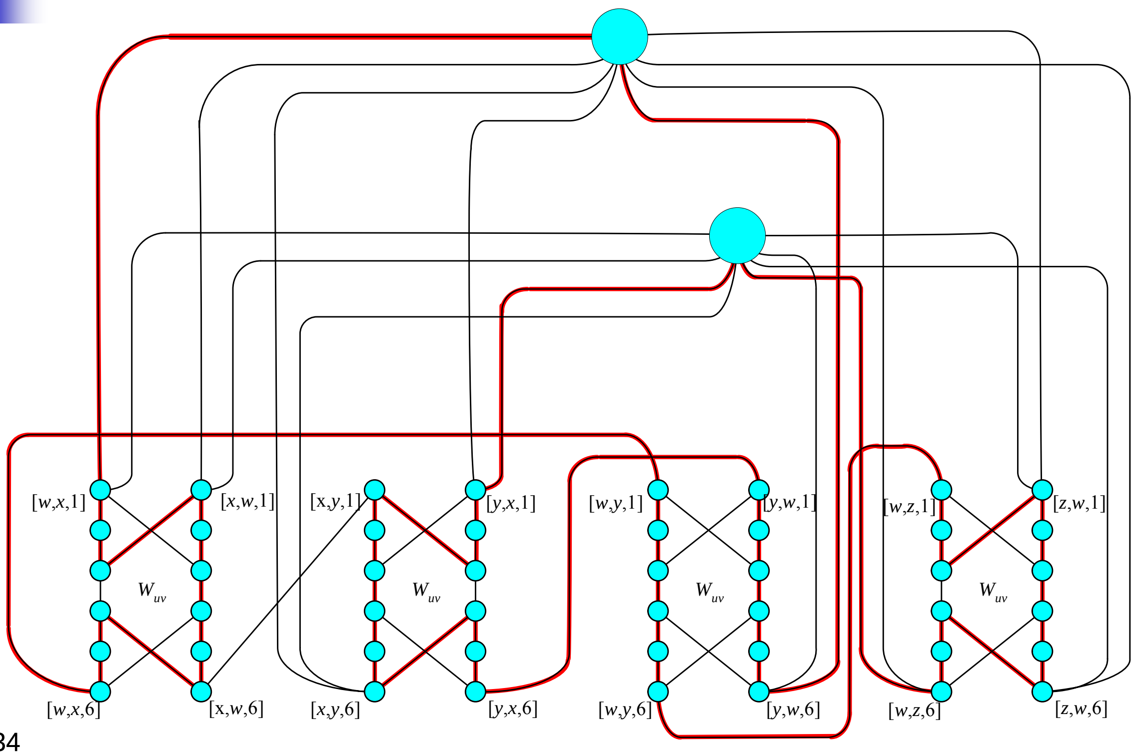
The reduction of an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem.

- (a) An undirected graph G with a vertex of size 2, consisting if the lightly shaded vertices w and y.
- (b) the undirected graph G’ produced by the reduction, with the hamiltonian path corresponding to the vertex cover shaded.
- The vertex cover {w,y} corresponds to edges (s_1,[w,x,1]) and (s_2,[y,x,1]) appearing in the hamiltonian cycle.

(a)



(b)



Three types of edges in E’

1. Edges in widget.
2. $\{([u,u^{(i)},6],[u,u^{(i+1)},1]) : 1 \leq i \leq \text{degree}(u)-1\}$
3. $\{(s_j,[u,u^{(1)},1]) : u \in V \text{ and } 1 \leq j \leq k\} \cup$
   $\{(s_j,[u,u^{(\text{degree}(u))},6]) : u \in V \text{ and } 1 \leq j \leq k\}$

The reduction performed in polynomial time

- $|V'| = 12|E| + k$
  $\leq 12|E| + |V|$

- $|E'| = (14|E|) + (2|E| - |V|) + (2k|V|)$
  $= 16|E| + (2k-1)|V|$
  $\leq 16|E| + (2|V|-1)|V|$

- The transformation from graph G to G' is a reduction.
- That is, G has a vertex cover of size k iff G' has a hamiltonian cycle.

Theorem 34.14. The traveling salesman problem is NP-complete.

Proof.

- $TSP \in NP$

$HAM - CYCLE \leq_P TSP$

34.5.5 The subset-sum problem

S={1,4,16,64,256,1040,1041,1093,1284,1344}

t=3754

S'={1,16,64,256,1040,1093,1284}

SUBSET-SUM={<S, t> | there exists a subset $S' \subset S$ such that $t = \sum_{s \in S'} s$

Theorem 34.15

- The subset-sum problem is NP-complete.

Proof.

- First, show that SUBSET-SUM is in NP.
- We now show that 3-CNF-SAT $\leq_p$ SUBSET-SUM.
- Given a 3-CNF formula ϕ over variables x_1, $x_2, \ldots, x_n$ with clauses $C_1, C_2, \ldots, C_k$, each containing exactly three distinct literals.
- The reduction algorithm constructs an instance <S,t> of the subset-sum problem such that ϕ is satisfiable iff there is a subset of S whose sum is exactly t.

Example

- The formula in 3-CNF is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$, and $C_4 = (x_1 \vee x_2 \vee x_3)$.
- A satisfying assignment of ϕ is $<x_1 = 0, x_2 = 0, x_3 = 1>$.

The reduction of 3-CNF-SAT to SUBSET-SUM

C_4 has no $\neg x_1$

C_4 has x_2

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v_1'	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v_2'	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v_3'	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s_1'	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s_2'	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s_3'	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s_4'	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

The reduction performed in polynomial time

- The set S contains 2n+2k values, each of which has n+k digits, and the time to produce each digit is polynomial in n+k.
- The target t has n+k digits, and the reduction produces each in constant time.

- 3-CNF formula ϕ is satisfiable iff there is a subset $S' \subseteq S$ whose sum is t.