NP-completeness

Chapter 34

Sergey Bereg

April 2015

Examples

Some problems admit *polynomial time algorithms*, i.e. $O(n^k)$ running time where n is the input size. We will study a class of *NP-complete problems* which will be defined later. No polynomial time algorithm is known for *any* NP-complete problem.

Shortest vs. longest simple paths. Given a directed graph $G = (V, E)$ with non-negative edge weights, the *shortest* path from between two vertices can be found in $O(VE)$ time. Finding the *longest* simple path is a NP-complete problem.

What if we modify Bellman-Ford algorithm by changing "$>$" to "$<$" in the relaxation condition?

```
RELAX(u, v, w)
// relax edge (u, v) using weight w(u, v)
1    if d[v] > d[u] + w(u, v) then
2        d[v] = d[u] + w(u, v)
3        π[v] = u
```

Examples

Some problems admit *polynomial time algorithms*, i.e. $O(n^k)$ running time where n is the input size. We will study a class of *NP-complete problems* which will be defined later. No polynomial time algorithm is known for *any* NP-complete problem.

Shortest vs. longest simple paths. Given a directed graph $G = (V, E)$ with non-negative edge weights, the *shortest* path from between two vertices can be found in $O(VE)$ time. Finding the *longest* simple path is a NP-complete problem.

There is a graph G and a vertex $s \in G$ such that a modified Bellman-Ford algorithm does not find the longest simple paths.

Definitions

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

Example. An instance for SHORTEST-PATH is a graph $G = (V, E)$ and two vertices. A solution is an ordered sequence of vertices (can be empty sequence if no path exists). A given instance may have more than one solution.

A *decision problem* is a problem having yes/no solution. So $S = \{0, 1\}$. Many problems are *optimization problems* where some value must be minimized or maximized. Usually it is easy to cast an optimization problem as a decision problem.

Example. Decision problem for SHORTEST-PATH is the problem PATH: Given a graph $G = (V, E)$, vertices u and v, and an integer k, is there a path of from u to v consisting at most k edges?

Encodings

Encoding of a set S is a mapping $e : S \to \{0,1\}^*$ from S to the set of binary strings. *Example*: $S = \mathbb{N}$, then binary code $e : \mathbb{N} \to \{0,1\}^*$.

A problem whose instance set is $\{0,1\}^*$ is called a *concrete problem*. We say that an algorithm *solves* a concrete problem in time $O(T(n))$ if it takes $O(T(n))$ time for a problem instance i of length $n = |i|$. If $T(n) = O(n^k)$ then we say that the problem is *polynomially solvable*.

Complexity class P is a class of concrete decision problems that are polynomially solvable.

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.
- *Example*: if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, \dots\}$ is the language of binary representations of prime numbers.
- We denote the *empty string* by ε and the *empty language* by $\emptyset$.
- So, $L \subseteq \Sigma^*$ where Σ^* is the language of all strings on Σ.

Operations on languages:
union $L_1 \cup L_2$,
intersection $L_1 \cap L_2$,
complement of L is $\Sigma^* - L$,
concatenation of L_1 and L_2 is $\{x_1 x_2 : x_1 \in L_1, x_2 \in L_2\}$,
closure or Kleene star $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$ where L^k is the language obtained by concatenating L to itself k times.

Formal language framework

- An algorithm A *accepts* a string $x \in \{0,1\}^*$ if, given input x, the algorithm's output is 1, i.e. $A(x) = 1$.
- A *rejects* x if $A(x) = 0$.
- The language *accepted* by an algorithm A is the set of strings $L = \{x \in \{0,1\}^* : A(x) = 1\}$.

Question: Is it true that A rejects a string $x \notin L$?

Formal language framework

- An algorithm A *accepts* a string $x \in \{0,1\}^*$ if, given input x, the algorithm's output is 1, i.e. $A(x) = 1$.
- A *rejects* x if $A(x) = 0$.
- The language *accepted* by an algorithm A is the set of strings $L = \{x \in \{0,1\}^* : A(x) = 1\}$.

Question: Is it true that A rejects a string $x \notin L$?

No, A may not reject a string $x \notin L$ (A may loop forever on x).

Formal language framework

- A language L is *decided* by an algorithm A if A accepts/rejects every string x in/not in L.
- A language L is *accepted in polynomial time* by an algorithm A if, for any $x \in L, |x| = n$, it accepts $x \in L$ in $O(n^k)$ time for some fixed k.
- A language L is *decided in polynomial time* by an algorithm A if, for any $x \in \{0,1\}^*, |x| = n$, it decides whether $x \in L$ in $O(n^k)$ time for some fixed k.

$\mathrm{P} = \{L \subseteq \{0,1\}^* : \exists A$ that decides L in polynomial time$\}$.

Theorem. $\mathrm{P} = \{L \subseteq \{0,1\}^* : \exists A$ that accepts L in polynomial time$\}$.

Formal language framework

Theorem. $\mathrm{P} = \{L \subseteq \{0,1\}^* : \exists A$ that accepts L in polynomial time$\}$.

Proof. If a language L is decided by a polynomial-time algorithm A, then L is accepted by A.

Suppose that a language L is accepted by a polynomial-time algorithm A with running time at most $T(n) = cn^k$.
We use a "simulation" argument and create a new algorithm A'.
For an input string x, the algorithm A'

1. Computes the length $n = |x|$.
2. Simulates the action of A only for time $T(n)$.
   If A accept/rejects x then A' reports it and stops.
3. Rejects x.

Verification algorithms

Hamiltonian cycle problem HAM-CYCLE. Given an undirected graph G, is there a simple cycle that contains all the vertices of G?

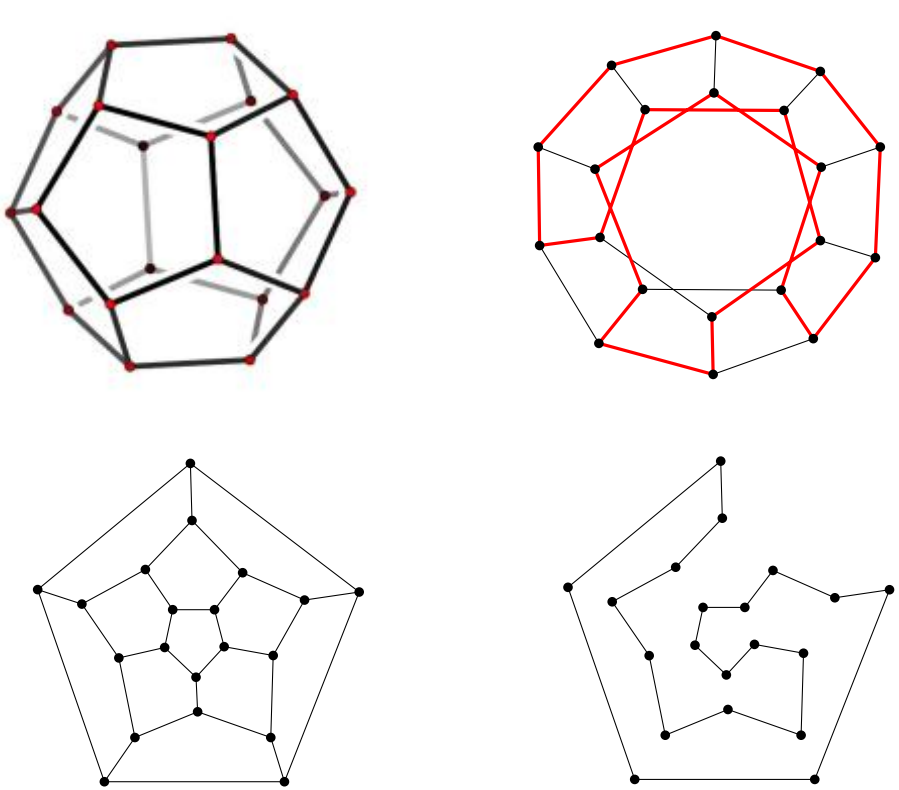
An algorithm: encode graph with adjacency matrix, the number of vertices $m \geq \sqrt{n}$. There are $m!$ permutations and the running time to check them is at least $\Omega(m!) = \Omega((\sqrt{n})!) = \Omega(2^{\sqrt{n}})$. Not polynomial time.

To verify a solution of HAM-CYCLE we need a *certificate*, for example, the hamiltonian cycle. A *verification algorithm* has two arguments - input string and certificate string. A two-argument algorithm A *verifies* an input string x if there is a certificate y such that $A(x, y) = 1$.
The *language verified* by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* \text{ such that } A(x, y) = 1\}.$$

Hamiltonian Game

Is there a walk along a closed path on dodecahedron such that every vertex is visited one time?



Complexity class NP

Complexity class NP is the class of languages that can be verified by a polynomial time algorithm. So, $L \in \mathrm{NP}$ if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* \text{ such that } |y| = O(|x|^c) \text{ and } A(x,y) = 1\}.$$

We say that algorithm A *verifies* L in *polynomial time*.
Example: HAM-CYCLE $\in \mathrm{NP}$.
If $L \in \mathrm{P}$ then $L \in \mathrm{NP}$ since there is a polynomial-time algorithm to decide L (the verification algorithm ignores the second argument). Thus, $\mathrm{P} \subseteq \mathrm{NP}$. Whether $\mathrm{P} = \mathrm{NP}$ is the famous problem.

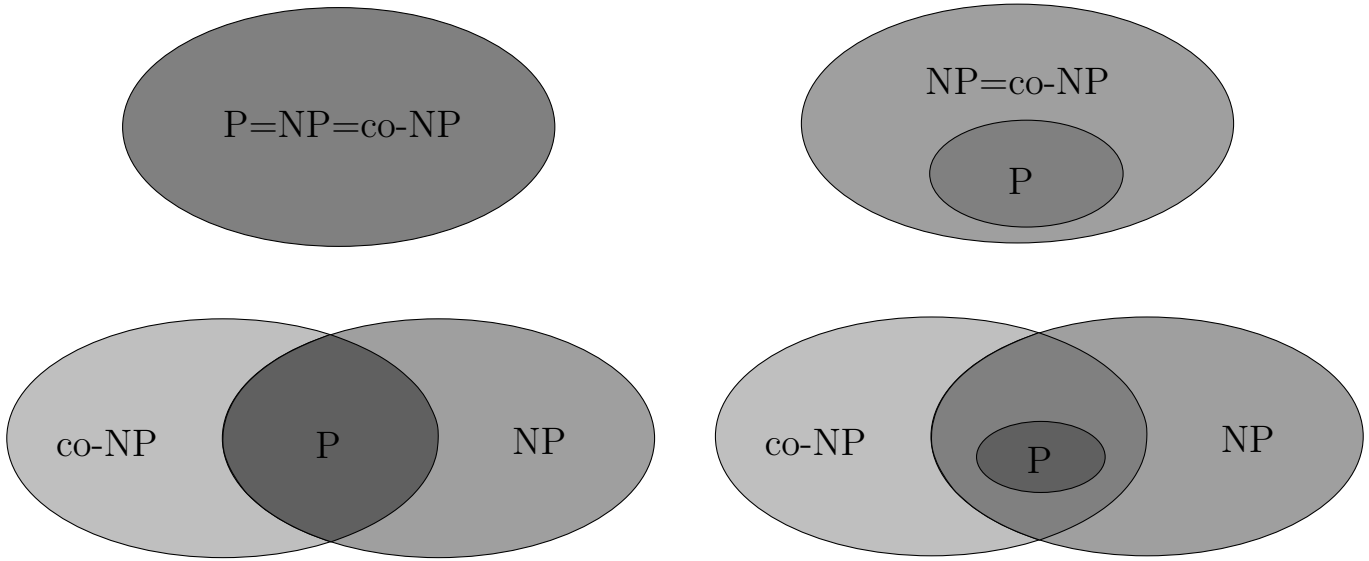
It is not even known if NP is closed under complement.
That is, does $L \in \mathrm{NP}$ imply $\overline{L} \in \mathrm{NP}$?

Complexity class co-NP is the set of languages L such that $\overline{L} \in \mathrm{NP}$.
So, the question is NP=co-NP?

Classes P, NP, and co-NP

Since P is closed under complement, we have $\mathrm{P} \subseteq \mathrm{NP} \cap \text{co-NP}$.

There are 4 possible relations between P, NP and co-NP.



Reducibility

A language L_1 is *polynomial-time reducible* to a language L_2, written $L_1 \leq_{\mathrm{P}} L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$

$x \in L_1$ if and only if $f(x) \in L_2$.

f is the *reduction function* and a polynomial-time algorithm F that computes f is called a *reduction algorithm*.

Example. EQUAL-NUMBERS$\leq_{\mathrm{P}}$SAME-POINTS.

EQUAL-NUMBERS: Given n integer numbers, are there two equal numbers?

SAME-POINTS: Given n points in the plane with integer coordinates, are there two coinciding points?

Reducibility

Lemma. If $L_1, L_2 \in \{0,1\}^*$ are languages such that $L_1 \leq_{\mathrm{P}} L_2$ and $L_2 \in \mathrm{P}$, then $L_1 \in \mathrm{P}$.

Reducibility

Lemma. If $L_1, L_2 \in \{0,1\}^*$ are languages such that $L_1 \leq_{\mathrm{P}} L_2$ and $L_2 \in \mathrm{P}$, then $L_1 \in \mathrm{P}$.

Proof. Consider the algorithm for L_1 that is combination of the reduction algorithm and the algorithm for L_2.

NP-completeness

A language L is NP-*complete* if

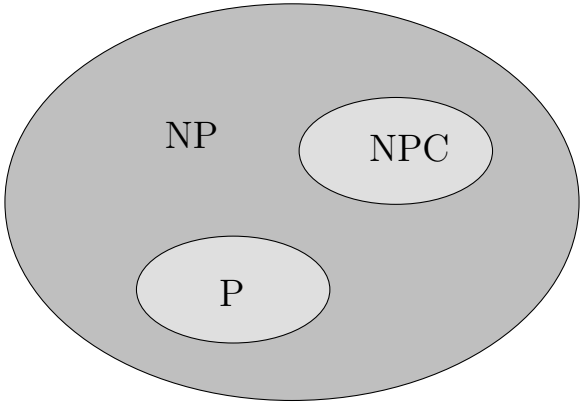
1. $L \in \mathrm{NP}$, and
2. $L' \leq_{\mathrm{P}} L$ for every $L' \in \mathrm{NP}$.

We say that a language is *NP-hard* if it satisfies condition 2 (1 may be satisfied or not). Any NP-complete problem is NP-hard.

Example of NP-hard problem that is not NP-complete (does not satisfy the condition 1): halting problem (given a program and its input, will it run forever?).

Theorem. If any NP-complete problem is solvable in polynomial time, then $\mathrm{P} = \mathrm{NP}$. Equivalently, if some problem in NP is not solvable in polynomial time, then no NP-complete problem is polynomial-time solvable.

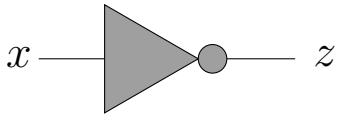
Classes P, NPC



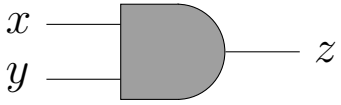
How most theoretical computer scientists view the relationships among P, NP, and NPC.

Circuit Satisfiability

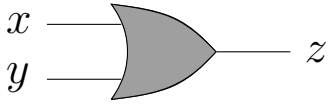
A *boolean combinatorial circuit* is composed of AND, OR, and NOT gates connected by *wires*.
Circuit input is a wire that is not the output of a gate.
Circuit output is a wire that is not the input of a gate; it is unique.
A *truth assignment* of a circuit is a set of boolean input values.



x	$\neg x$
0	1
1	0



x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

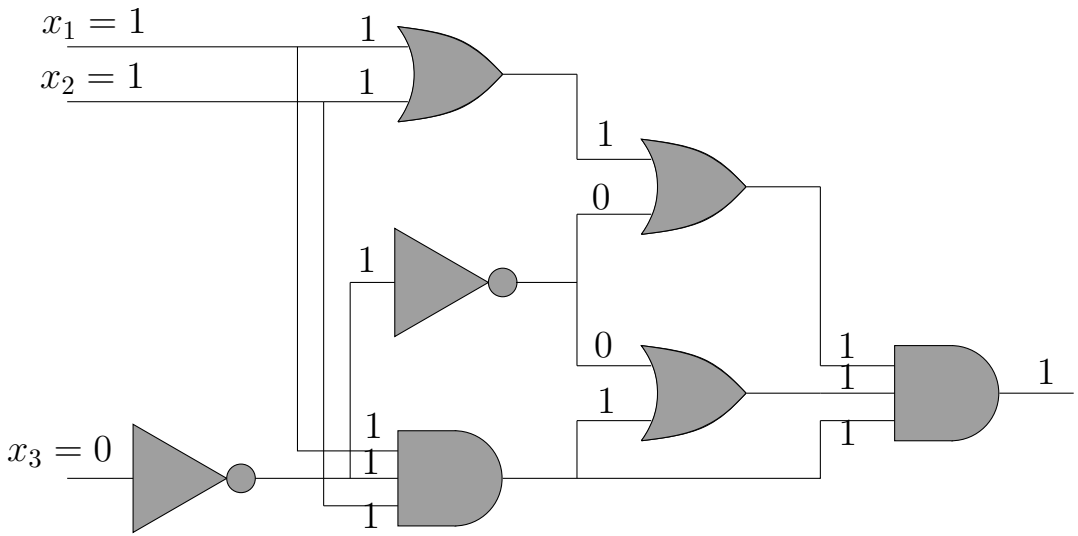


x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

A boolean combinatorial circuit is *satisfiable* if it has a truth assignment that causes the output to be 1.
This assignment is called a *satisfying assignment*.

Circuit Satisfiability

Example. $x_1 = 1, x_2 = 1, x_3 = 0$ is a *satisfying assignment* and the circuit is satisfiable.



Circuit Satisfiability

CIRCUIT-SAT *problem*. Given a boolean combinatorial circuit, is it satisfiable?

Lemma. CIRCUIT-SAT is in NP.

Proof. Make a verification algorithm $A(x, y)$ where x is an encoding of circuit and y is the certificate (satisfying assignment). A must be a polynomial-time algorithm.

Idea: for each logic gate A computes its output based on inputs. At any moment, there is a gate with "ready" inputs. A returns the final output (of entire circuit). □

Circuit Satisfiability

Lemma. CIRCUIT-SAT is NP-hard.

Circuit Satisfiability

Lemma. CIRCUIT-SAT is NP-hard.

Proof. Let L be a language in NP and let A be a verification algorithm for A. We want to design a polynomial-time algorithm that maps every $x \in \{0,1\}^*$ to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT.

Idea: model the execution of the algorithm (code for A, program counter, machine state, and working storage) with a circuit C. There are $T(n) = O(n^k)$ steps in the execution. We put $T(n)$ copies of the circuit M that implements the computer hardware. We "run" the algorithm on the certificate y. □

Boolean formula satisfiability

An instance of CIRCUIT-SAT can be transformed into a *boolean formula* ϕ composed of

1. n boolean variables $x_1, \ldots, x_n$
2. m boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication. like $\geq$), $\leftrightarrow$ (if and only if. like $=$)
3. parentheses

x	y	$x \rightarrow y$	$x \leftrightarrow y$
0	0	1	1
0	1	1	0
1	0	0	0
1	1	1	1

Example: $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Boolean formula satisfiability

A *truth assignment* for a boolean function ϕ is a set of values of variables of ϕ and a *satisfying assignment* is a truth assignment that causes it to evaluate to 1. If a satisfying assignment exists then ϕ is *satisfiable*.

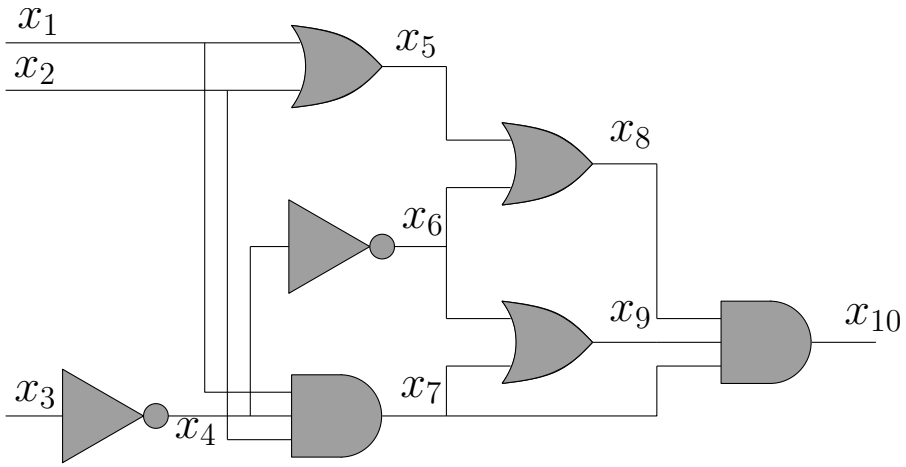
SAT *problem*. Given a boolean formula, is it satisfiable?

Theorem. SAT is NP-complete.

Proof. 1. We show that SAT is in NP: certificate is the truth assignment that satisfies the formula. Can be checked in polynomial time.

Boolean formula satisfiability

2. We show that CIRCUIT-SAT$\leq_{\mathrm{P}}$SAT. Assign a variable for each wire. Formula for SAT is the AND of the circuit output and the conjunction of clauses describing the operation of each gate. The formula can be produced in polynomial time. It is satisfiable if and only if the circuit is satisfiable.



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

3-CNF

A *literal* in a boolean formula is a variable or its negation. A boolean formula is in *conjunctive normal form*, or *CNF*, if it is expressed as an AND of *clauses*, each of which is the OR of one or more literals. A boolean formula is in *3-conjunctive normal form*, or *3-CNF*, if each clause has exactly 3 distinct literals.

Example: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$.

3-CNF-SAT, or *3-SAT*, asks whether a given 3-CNF is satisfiable.

Theorem. 3-SAT is NP-complete.

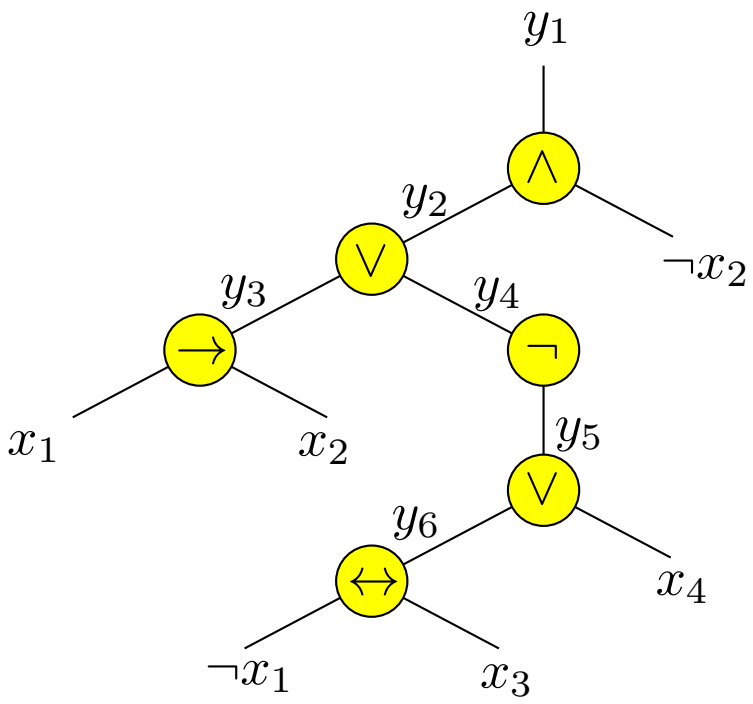
Proof. 3-SAT$\in \mathrm{NP}$ is similar to the proof that SAT$\in \mathrm{NP}$ (every instance of 3-SAT is an instance of SAT).

3-CNF

We prove that SAT$\leq_{\mathrm{P}}$3-SAT.

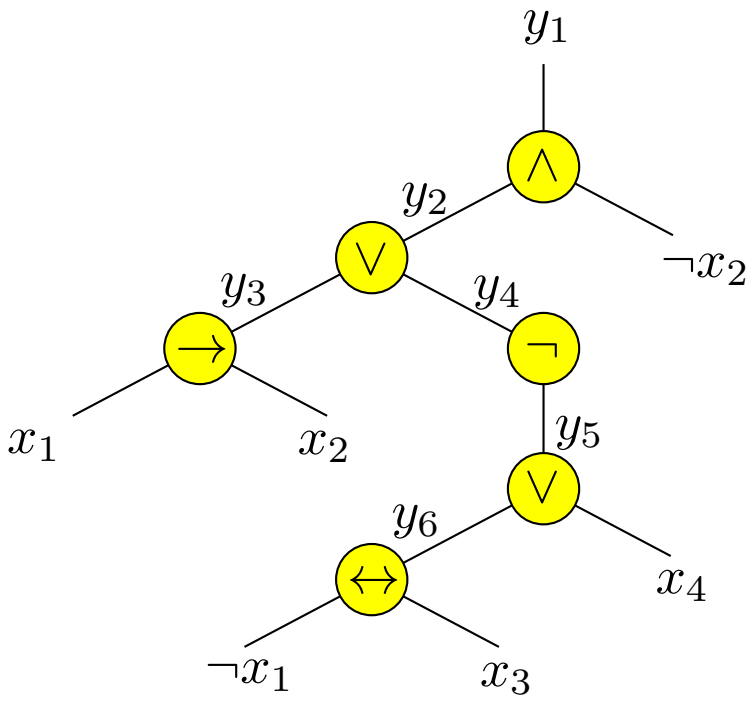
Let ϕ be an input formula for SAT, for example
$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Construct a binary "parse" tree. Introduce new variables y_i.



3-CNF

Transform the tree into a formula ϕ':



$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ &\wedge (y_4 \leftrightarrow \neg y_5) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

3-CNF

ϕ' is AND of $\phi'_i, i = 1, 2, ..$ Transform each ϕ'_i as follows.

Make truth table for ϕ'_i.

Convert $\neg\phi'_i$ using 0's in the table.

For example $\phi'_1 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

y_1	y_2	x_2	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Then $\neg\phi'_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

3-CNF

$$\neg\phi'_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

Applying DeMorgan's laws

$$\phi'_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

This way we create a CNF formula ϕ'' so that each clause has at most 3 literals. We change those with 1 and 2 literals:

If $\phi'_i = l_1 \vee l_2$ then $\phi'_i = (l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \neg x)$.

If $\phi'_i = l_1$ then $\phi'_i = (l_1 \vee x \vee y) \wedge (l_1 \vee x \vee \neg y) \wedge (l_1 \vee \neg x \vee y) \wedge (l_1 \vee \neg x \vee \neg y)$.

□

NP-complete problems

A *clique* in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The *size* of a clique is the number of its vertices. *Clique Problem* is to find a clique of maximum size in G. Decision problem:

CLIQUE$= \{\langle G, k\rangle : G$ is a graph with a clique of size $k\}$.

Naive algorithm needs $\Omega(k^2\binom{|V|}{k})$ time.

Theorem. The clique problem is NP-complete.

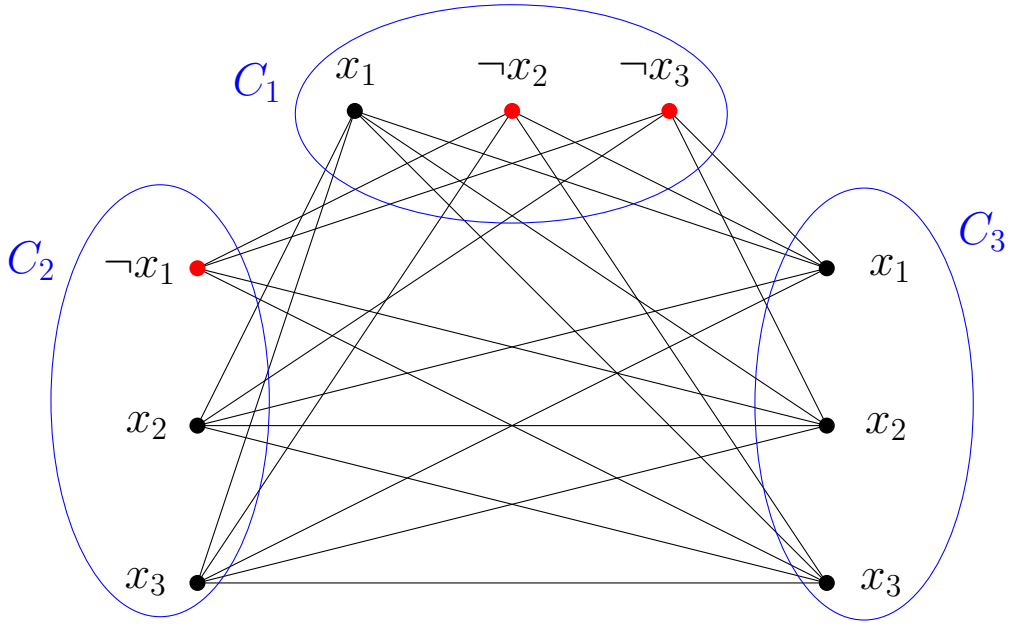
Proof. CLIQUE$\in \mathrm{NP}$ if we choose the clique as the certificate.

CLIQUE is NP-hard if we show 3-SAT$\leq_{\mathrm{P}}$CLIQUE.

Clique Problem

The reduction algorithm begins with an instance of 3-SAT problem $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$. We construct a graph $G(V, E)$. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$, we place 3 vertices v_1^r, v_2^r, v_3^r. We put an edge between v_i^r and v_j^s if $r \neq s$ and literals of v_i^r and v_j^s are not contradictory.

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



Clique Problem

We show that the transformation of ϕ is a reduction.

Suppose that ϕ has a satisfying assignment. Then each clause C^r contains at least one literal l_i^r assigned to 1. These vertices form a clique of size k in G.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, so V' contains exactly one vertex per triple. We assign 1 to each literal in the clique (we do not assign 1 to a literal and its complement). We assign any value to a variable whose literals are not in clique. All clauses are satisfied and ϕ is satisfied. $\square$

Vertex-Cover Problem

A *vertex cover* of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices 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 G.
Decision problem:

VERTEX-COVER$= \{\langle G, k\rangle$: graph G has a vertex cover of size $k\}$.

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

Proof. VERTEX-COVER$\in$ NP if we choose the vertex cover as the certificate.

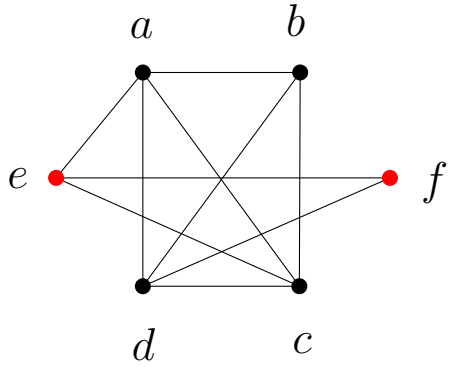
VERTEX-COVER is NP-hard if we show CLIQUE$\leq_P$VERTEX-COVER.

Vertex-Cover Problem

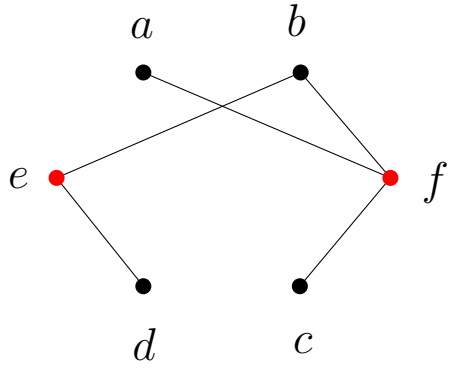
We define the *complement* of G as $\overline{G} = (V, \overline{E})$ where

$$\overline{E} = \{(u, v) : u, v \in V, u \neq v \text{ and } (u, v) \notin E\}.$$

An instance of $\langle G, k \rangle$ of the clique problem reduces to an instance $\langle \overline{G}, |V| - k \rangle$ of the vertex-cover problem. Indeed, a set $V' \subseteq V, |V'| = k$ is the clique of G if and only if $V - V'$ is a vertex cover of $\overline{G}$. The reduction can be done in polynomial time. □



G, clique $\{a, b, c, d\}$



$\overline{G}$, vertex cover $\{e, f\}$

Vertex-Cover Problem

```
APPROX-VERTEX-COVER(G)
1   C = ∅
2   E' = E[G]
3   while E' ≠ ∅
4       let (u, v) be any edge of E'
5       C = C ∪ {u, v}
6       remove from E' every edge incident on either u or v
7   return C
```

Theorem. APPROX-VERTEX-COVER computes a 2-approximation in $O(V + E)$ time.

Hamiltonian-cycle Problem

A *hamiltonian cycle* in a graph $G = (V, E)$ is a simple cycle that contains all vertices of G. G is *hamiltonian* if it contains a hamiltonian cycle. *Hamiltonian-cycle problem* is to detect if a graph G is hamiltonian.

HAM-CYCLE$= \{\langle G \rangle : G$ is a hamiltonian graph $\}$.

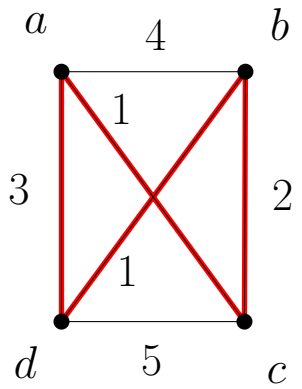
Theorem. The hamiltonian-cycle problem is NP-complete.

Proof. HAM-CYCLE$\in$ NP if we choose the hamiltonian cycle as the certificate.

HAM-CYCLE is NP-hard if we show VERTEX-COVER$\leq_P$HAM-CYCLE. Use *widgets* and *selector vertices* to transform G to G'. Polynomial time. G has a vertex cover V^* if and only if G' has a hamiltonian cycle. □

Traveling-salesman problem

Let G be a complete graph with integer cost $c(i, j)$ to travel from city i to j. *Traveling-salesman problem* is to find a minimum-cost *tour* (hamiltonian cycle).



traveling-salesman tour $acbda$ of costs 7.

TSP$= \{\langle G, c, k\rangle : G$ is a complete graph and has a traveling-salesman tour of cost at most $k\}$.

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

Proof. TSP$\in$ NP if we choose the tour as the certificate.
Hardness: HAM-CYCLE$\leq_P$TSP.

Subset-Sum Problem

Given a finite set $S \subset \mathbb{N}$ (natural numbers or positive integers) and a target t, the *subset-sum problem* is to find a subset $S' \subseteq S$ whose elements sum to t.

Example: $S = \{1, 3, 3, 9, 18\}, t = 10$. Then $S' = \{1, 9\}$ is a solution.

SUBSET-SUM$= \{\langle S, t\rangle$: there exists a subset $S' \subseteq S$ such that $t = \sum_{s \in S'} s\}$.

Theorem. The subset-sum problem is NP-complete.

Proof. SUBSET-SUM$\in$ NP if we choose S' as the certificate.
Hardness: 3-SAT$\leq_P$SUBSET-SUM.

Subset-Sum Problem

Reduction. $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ 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 $C_4 = (x_1 \vee x_2 \vee x_3)$.

		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