Polynomial and Non-Polynomial Time

January 12, 2021

1 Introduction

In most of the analysis of algorithms class, we have been looking and studying polynomial time algorithms. The polynomial time algorithm class P are algorithms that on inputs of size n have a worst case running time of $O\left(n^k\right)$ for some constant k. Thus, informally, we can say that the Non-Polynomial (NP) time algorithms are the ones that cannot be solved in $O\left(n^k\right)$ for any constant k.

2 The NP Problem

In **Nondeterministically Polynomial** NP problems, there is a theorem that allow us to ask the question $P \neq NP$? Until now no polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. For this reason, we dedicate our time to these kind of problems. The following figure explain the two possibilities.

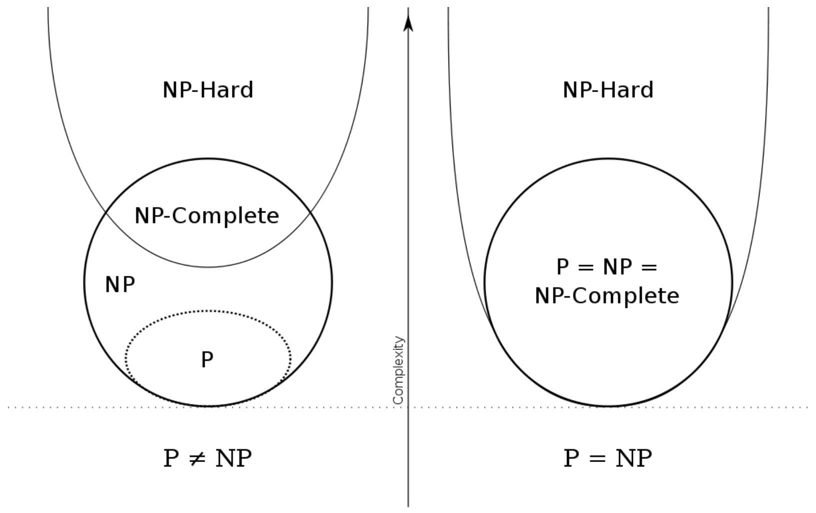


Figure 1: Two views of the world

Examples of how a simple change in what we are asking to a problem can change a polynomial time problem into a NP-problem are:

- Shortest vs. longest simple paths.
- Euler tour vs. Hamiltonian cycle. Here Euler is over the edges and Hamiltonian is over the vertices.
- 2-CNF satisfiability vs. 3-CNF satisfiability.

3 NP-completeness

NP problems are the kind of problems that can be "verified" in polynomial time. What does this mean? That somebody, an oracle, has given us a "certificate" of a solution, thus, we can in polynomial time if the certificate is correct. An example would be, for the Hamiltonian cycle, a sequence of vertices $\left\langle v_1, v_2, ..., v_{|V|}\right\rangle$ from a graph $G = (V, E)$. Then, we check in polynomial time if $(v_i, v_{i+1}) \in E$ for $i = 1, 2, ..., |V| - 1$ and that $\left(v_{|V|}, v_1\right) \in E$. Using this idea, it is possible to define informally that a NP-Complete problem is a problem that is NP and it is as "hard as any NP". Actually, we will define this better later.

4 Basic Notions for the NP

4.1 Decision problems vs. optimization problems

There are two main problems that we attack in computer science:

1. **The optimization problems**. For this problems we wish to find a desirable solution with the best possible value. Example of this are the Dynamic Programming Problems.
2. **The decision problems** are the ones where it is possible to answer "yes" or "no." For example, Given a directed graph G, with vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges? This is actually the framework used by the NP-problems.

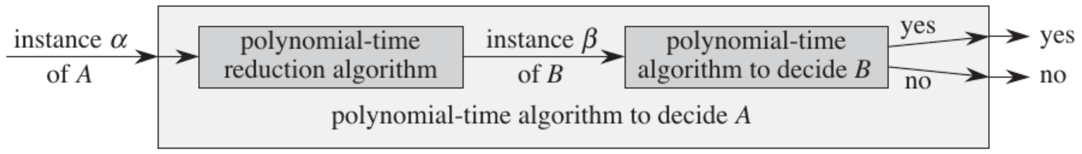
4.2 Reductions

Given A an instance of a problem that we would like to solve in polynomial time. For example, in PATH an instance would be a a particular graph G, with vertices u and v. and integer k. Now assume that a problem B exists that can be solved in polynomial time. In addition, theres is a procedure that can transform $\alpha \in A$ into $\beta \in B$ with the following characteristics:

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

The following figure gives us a way to solve our α problem.

Algorithm 1 Polynomial time Algorithm


instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A

Or in steps:

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.

5 Polynomial Time

For the polynomial time problems is necessary to have three things:

- What is an abstract problem?
- What is an encoding?
- A formal-language framework.

5.1 Abstract Problems

Definition 1. An abstract problem Q is a binary relation

$$Q : I \to S$$

I are a set of instances of the problem, and S are a set of possible solutions.

An example of this abstract definition of a problem is the decision problem where a machine needs to say if theres is or there is not a solution. The binary relation of the decision problem can be seen as

$$Q : I \to \{0, 1\} \tag{1}$$

5.2 Encoding

An encoding is necessary in order to represent our problems in a language that the computer machine can understand in order to try to solve it. An example could be the representation of graph $G = (V, E)$.

Example 2. A graph $G = (V, E)$ can be encoded as follow:

- Each vertex i can be encoded using a binary number. For example the set of vertices $\{1, 2, 3\}$ can be seen as the binary numbers 0, 1 and 10.
- Each edge then can be seen as a pair of binary numbers. for example $(1, 2)$ can be seen as $\{0, 1\}$
- An encoding for delimiters, thus you know where each definition starts and ends.

Now, using the encoding of the problem the Turing machine or whatever machine you have at your disposal can solve the problem. This encoded version of the problem is called a **concrete problem**. Thus, a concrete problem is polynomial-time solvable, if there exists an algorithm to solve it in time $O\left(n^k\right)$ for some constant k.

It would be **nice to extend** the polynomial-time solvability from concrete problems to abstract problems by using encodings. However, the definition needs to be independent of any particular encoding, i.e. the efficiency of solving a problem should not depend on how the problem is encoded. Unfortunately, it depends quite heavily on the encoding.

Example 3. Assume k an integer as the input of an algorithm, and the running time is $\Theta(k)$. If k is encoded as a string of 1's, then the running time of the algorithm is $O(n)$ on length-n inputs, which is polynomial time. Now, we use the binary representation of the integer k, then $n = \lfloor \lg k \rfloor + 1$, then we have the following running time $\Theta(k) = \Theta(2^n)$ which is exponential in the size of the input. Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

Then, the way we encode the abstract problem is quite important with respect to the polynomial time solvability. Nevertheless, in practice, if we rule out "expensive encodings" as the unary encoding, the encoding of a problem makes little difference. For this, we have the following definitions.

Definition 4. We say that a function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ is a polynomial-time computable if there exists a polynomial-time algorithm A, such that given any input $x \in \{0,1\}^*$, it produces as output $f(x)$.

This allows to define a type of correlation between encodings.

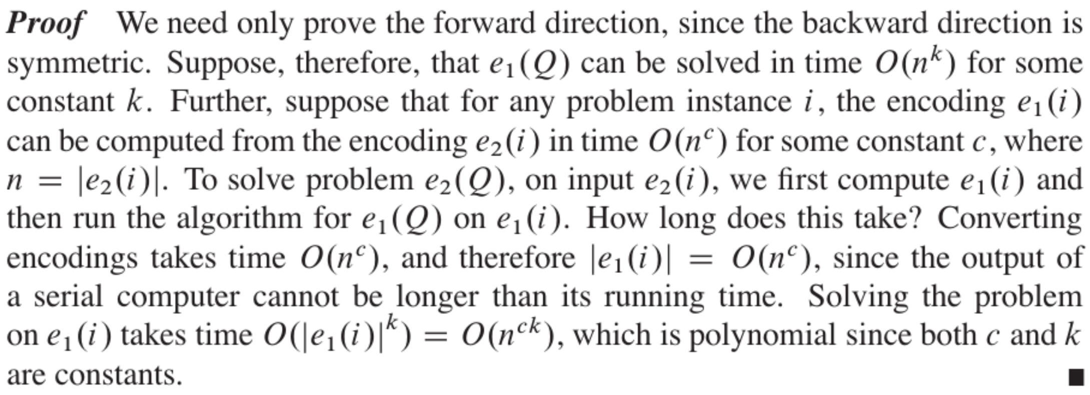
Definition 5. For some set I of problem instances, we say that two encodings e_1 and e_2 are polynomially related if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

Using this definition, we have the following lemma.

Lemma. *34.1*

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

Proof: *We are going to prove on direction, the other*



Proof We need only prove the forward direction, since the backward direction is symmetric. Suppose, therefore, that $e_1(Q)$ can be solved in time $O(n^k)$ for some constant k. Further, suppose that for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in time $O(n^c)$ for some constant c, where $n = |e_2(i)|$. To solve problem $e_2(Q)$, on input $e_2(i)$, we first compute $e_1(i)$ and then run the algorithm for $e_1(Q)$ on $e_1(i)$. How long does this take? Converting encodings takes time $O(n^c)$, and therefore $|e_1(i)| = O(n^c)$, since the output of a serial computer cannot be longer than its running time. Solving the problem on $e_1(i)$ takes time $O(|e_1(i)|^k) = O(n^{ck})$, which is polynomial since both c and k are constants. ■

6 The Formal Language Framework

This at the slides or your class in Automatons.

7 Using the Formal Language

We can use the formal language framework to represent our problems.

Example 6. $Q$is entirely characterized by instances that produce a yes answer, then

$$L = \{x \in \Sigma^* | Q(x) = 1\}.$$

Another example is the following one.

Example 7. An algorithm A accepts a string $x \in \{0,1\}^*$if, given x, the algorithm's output $A(x)$ is 1. Thus, the language accepted by an algorithm A is the set of strings:

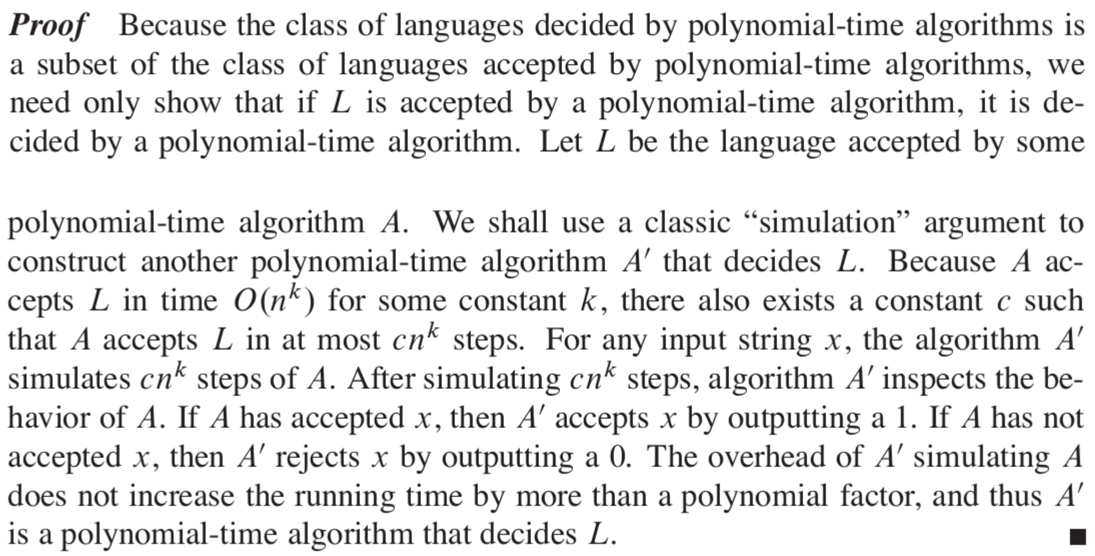
$$L = \left\{x \in \{0,1\}^* | A(x) = 1\right\}.$$

It is important to mention, that even when an algorithm accept strings x from L, the algorithm could not reject a string $x \notin L$ provided as input to it. The algorithm could loop forever. Thus, we are more stringent: **A language L is decided by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A.** It is more: **A language L is accepted in polynomial time by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O\left(n^k\right)$.**

This formal language allows us to define a complexity class as a set of languages, membership in which is determined by a complexity measure. This can be seen by the following theorem.

Theorem. *34.2 The class P of languages that can be accepted in polynomial time or:*

$$P = \{L | L \text{ is accepted by a polynomial-time algorithm}\}$$


Proof Because the class of languages decided by polynomial-time algorithms is a subset of the class of languages accepted by polynomial-time algorithms, we need only show that if L is accepted by a polynomial-time algorithm, it is decided by a polynomial-time algorithm. Let L be the language accepted by some

polynomial-time algorithm A. We shall use a classic "simulation" argument to construct another polynomial-time algorithm A' that decides L. Because A accepts L in time $O(n^k)$ for some constant k, there also exists a constant c such that A accepts L in at most cn^k steps. For any input string x, the algorithm A' simulates cn^k steps of A. After simulating cn^k steps, algorithm A' inspects the behavior of A. If A has accepted x, then A' accepts x by outputting a 1. If A has not accepted x, then A' rejects x by outputting a 0. The overhead of A' simulating A does not increase the running time by more than a polynomial factor, and thus A' is a polynomial-time algorithm that decides L. ■

8 More Formal definitions of NP

Finally, using this concept of "decision," formal languages and Turing machines, we have the following definition:

Definition 8. A problem is said to be **Nondeterministically Polynomial (NP)** if we can find a no-deterministic Turing machine that can solve the problem in a polynomial number of nondeterministic moves.

A simpler definition is the following one.

Definition 9. A problem is said to be NP if its solution comes from a finite set of possibilities, and it takes polynomial time to verify the correctness of a candidate solution.

A more formal definition comes from the NTIME concept.

Definition 10. Given that NTIME($f(n)$) is the set of decision problems that can be solved by a **non-deterministic Turing machine** i.e. the set of decision problems that can be solved by a non-deterministic Turing machine which runs in time $O(f(n))$. Thus, the The complexity class NP can be defined as follows:

$$NP = \cup_{k \in N} NTIME\left(n^k\right)$$

9 Polynomial Time Verification

We have the following intuitive definition:

Example 11. Given an instance of a decision problem. For example, given $\langle G, A, F, k\rangle$ an instance of PATH, and a possible solution a path p from A to F. Then, you need to check if the length of p is at most k.

10 3-CNF is NP-Complete

A literal in a boolean formula is an occurrence of 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 three distinct literals.

Theorem. *34.10 Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.*

Proof. The Part of NP uses the same idea from the SAT problem. The interesting part is $SAT \leq_p 3-CNF$. For this, we have the following stages

1. First given an instance of SAT, for example,

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

parsed into

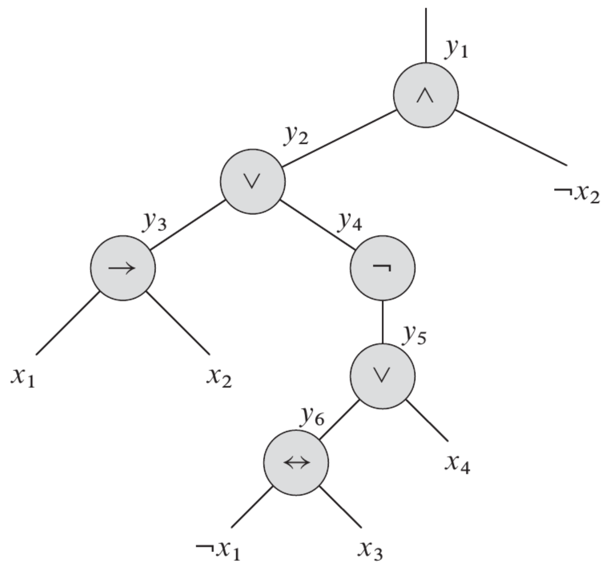


Figure 2: Parsing of the SAT formula

Note: Should the input formula contain a clause such as the OR of several literals, we use associativity to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. We can now think of the binary parse tree as a circuit for computing the function.

Now, we introduce a series of variables y_i and rewrite the original formula as

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

Figure 3: The Parsing into Φ'

Note: Observe that the formula thus obtained is a conjunction of clauses, each of which has at most 3 literals. The only requirement that we might fail to meet is that each clause has to be an OR of 3 literals.

2. Now, we rewrite each element ϕ'_i into a conjuctive normal form. For this we construct the trut table of each ϕ'_i

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

Figure 4: The Truth Table of $(y_1 \longleftrightarrow (y_2 \wedge \neg x_2))$

Then, we use the zero entries to construct the Disjuntive Normal Form (DNF) -an OR of ANDs- that is equivalent to $\neg\phi'_1$. We then negate this formula and conver it tinto a CNF formula ϕ''_1 using the Morgan's laws.

For example, the DNF formula equivalent to $\neg\phi'_1$

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

The conversion is like

$$\phi_1'' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Note Each formula has at most 3 literals.

3. To have all into 3 literals we use the following rules to include the necessary literals:

   For each clause C_i of ϕ'', we include the folloing clauses into ϕ''':

   (a) If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ''' .

   (b) If C_i has 2 distinct literals, that is, if $C_i = (l_1 \vee l_2)$, where l_1 and l_2 are literals, then include$(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of ϕ'''. The literals p and $\neg p$ merely fulfill the syntactic requirement that each clause of ϕ''' has exactly 3 distinct literals. Whether $p = 0$ or $p = 1$, one of the clauses is equivalent to $(l_1 \vee l_2)$, and the other evaluates to 1, which is the identity for AND.

   (c) If C_i has just 1 distinct literal l, then include $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ as clauses of ϕ'''. Regardless of the values of p and q, one of the four clauses is equivalent to l, and the other 3 evaluate to 1.

We can see that the 3-CNF formula ϕ'''. is satisfiable if and only if is satisfiable by inspecting each of the three steps. Like the reduction from CIRCUIT-SAT to SAT, the construction of ϕ' from ϕ in the first step preserves satisfiability. The second step produces a CNF formula ϕ'' that is algebraically equivalent to ϕ'. The third step produces a 3-CNF formula ϕ''' that is effectively equivalent to ϕ'', since any assignment to the variables p and q produces a formula that is algebraically equivalent to ϕ''.

The polynomial part in the slides. □

11 Clique is NP-Complete

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 vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a Graph. The formal definition is:

$$CLIQUE = \{\langle G, k \rangle \,|G \text{ is a graph containing a clique of size } k\}.$$

A naive way to see if a graph has a clique of size k is to list all k-subsets of size $|V|$ then check if they form a clique. It has the following running time $\Omega\left(k^2 \begin{pmatrix} |V| \\ 2 \end{pmatrix}\right)$.

Theorem. *34.11 The clique problem is NP-complete.*

Proof To show that CLIQUE $\in$ NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

We next prove that 3-CNF-SAT $\leq_{\text{P}}$ CLIQUE, which shows that the clique problem is NP-hard. You might be surprised that we should be able to prove such a result, since on the surface logical formulas seem to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses. For $r = 1, 2, \ldots, k$, each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r. We shall construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k.

We construct the graph $G = (V, E)$ as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, we place a triple of vertices v_1^r, v_2^r, and v_3^r into V. We put an edge between two vertices v_i^r and v_j^s if both of the following hold:

- v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
- their corresponding literals are ***consistent***, that is, l_i^r is not the negation of l_j^s.

We can easily build this graph from ϕ in polynomial time. As an example of this construction, if we have

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

then G is the graph shown in Figure 34.14.

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s map to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so ϕ is satisfied. (Any variables that do not correspond to a vertex in the clique may be set arbitrarily.) ■

12 The vertex-cover problem

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The size of a vertex cover is the number of vertices in it.

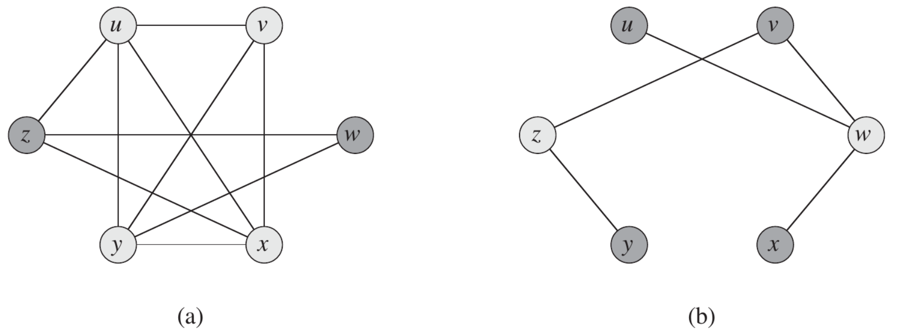


Figure 5: Example

The vertex-cover problem is to find a vertex cover of minimum size in a given graph or in optimization terms

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

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

Proof We first show that VERTEX-COVER $\in$ NP. Suppose we are given a graph $G = (V, E)$ and an integer k. The certificate we choose is the vertex cover $V' \subseteq V$ itself. The verification algorithm affirms that $|V'| = k$, and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. We can easily verify the certificate in polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_{\mathrm{P}}$ VERTEX-COVER. This reduction relies on the notion of the "complement" of a graph. Given an undirected graph $G = (V, E)$, we define the ***complement*** of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$. In other words, $\overline{G}$ is the graph containing exactly those edges that are not in G. Figure 34.15 shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

The reduction algorithm takes as input an instance $\langle G, k \rangle$ of the clique problem. It computes the complement $\overline{G}$, which we can easily do in polynomial time. The output of the reduction algorithm is the instance $\langle \overline{G}, |V| - k \rangle$ of the vertex-cover problem. To complete the proof, we show that this transformation is indeed a reduction: the graph G has a clique of size k if and only if the graph $\overline{G}$ has a vertex cover of size $|V| - k$.

Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\overline{G}$. Let (u, v) be any edge in $\overline{E}$. Then, $(u, v) \notin E$, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, at least one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from $\overline{E}$, every edge of $\overline{E}$ is covered by a vertex in $V - V'$. Hence, the set $V - V'$, which has size $|V| - k$, forms a vertex cover for $\overline{G}$.

Conversely, suppose that $\overline{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then, for all $u, v \in V$, if $(u, v) \in \overline{E}$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words, $V - V'$ is a clique, and it has size $|V| - |V'| = k$. ■