

NP-COMPLETENESS

Concepts

1.1 Introduction

Almost all the algorithms we have studied thus far have been polynomial-time algorithms: on inputs of size n , their worst-case running time is $O(n^k)$ for some constant k . However, not all problems can be solved in polynomial time. For example, there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how much time is provided. There are also problems that can be solved, but not in time $O(n^k)$ for any constant k . Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

1.2 Classes of Problems According to Runtime

P (Polynomial)

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k , where n is the size of the input to the problem. Most of the problems examined in previous chapters are in P.

NP (Nondeterministic Polynomial)

The class NP consists of those problems that are "verifiable" in polynomial time. What we mean here is that if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being given a certificate. We can believe that $P \subseteq NP$.

NPC (NP-Complete)

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.

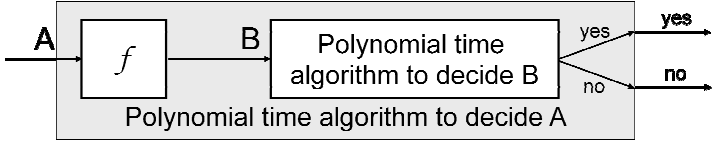
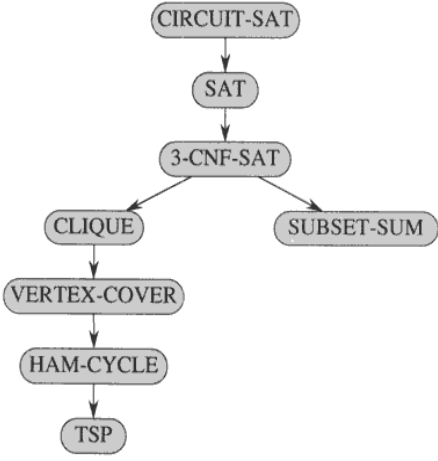
Again, no one has ever proven that no polynomial time algorithm can exist for any NP-Complete problem.

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.

1.3 Optimization Problems and Decision Problems

Many problems of interest are *optimization problems*, in which each feasible (i.e., "legal") solution has an associated value, and we wish to find a feasible solution with the *best* value. For example, in a problem that we call SHORTEST-PATH, we are given an undirected graph G and vertices u and v , and we wish to find a path from u to v that uses the fewest edges.

NP-completeness applies directly not to optimization problems, however, but to *decision problems*, in which the answer is simply "yes" or "no" (or, more formally, "1" or "0"). We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For SHORTEST-PATH, for example, a related decision problem, which we call PATH, is whether, given a directed graph G , vertices u and v , and an integer k , a path exists from u to v consisting of at most k edges.

<p>1.4</p>	<p>Reductions</p> <p>Given two problems A, B, we say that A is reducible to B ($A \leq_p B$) if:</p> <ol style="list-style-type: none"> 1. There exists a function f that <i>converts</i> the inputs of A to inputs of B in polynomial time. 2. $A(i) = \text{YES} \leftrightarrow B(f(i)) = \text{YES}$. [i.e., for an input i, A will be satisfied if and only if $B(f(i))$ satisfies.]
<p>1.5</p>	<p>Polynomial Reduction Algorithm</p> <p>To solve a decision problem A in polynomial time,</p> <ol style="list-style-type: none"> 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 
<p>1.6</p>	<p>Proving a Language to be NP-Complete</p> <ol style="list-style-type: none"> 1. Prove $L \in \text{NP}$. 2. Select a known NP-complete language L'. 3. Describe an algorithm that computes a function f mapping every instance $x \in \{0, 1\}^*$ of L' to an instance $f(x)$ 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.
<p>1.7</p>	<p>NP-completeness proof structure</p> 
<p>1.8</p>	<p>Circuit Satisfiability Problem</p> <p>Given a Boolean combinational circuit composed of AND, OR, and NOT, is it satisfiable?</p> <p>$\text{CIRCUIT-SAT} = \{ \langle C \rangle : C \text{ is a satisfiable Boolean circuit} \}$</p>
<p>1.9</p>	<p>Formula Satisfiability Problem</p> <p>We formulate the (formula) satisfiability problem in terms of the language SAT as follows. An instance of SAT is a Boolean formula ϕ composed of</p> <ol style="list-style-type: none"> 1. n Boolean variables: x_1, x_2, \dots, 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), \leftrightarrow (if and only if); and 3. parentheses. (Without loss of generality, we assume that there are no redundant parentheses, i.e., there is at most one pair of parentheses per Boolean connective.)

As in Boolean combinational circuits, a *truth assignment* for a Boolean formula φ is a set of values for the variables of φ , and a *satisfying assignment* is a truth assignment that causes it to evaluate to 1. A formula with a satisfying assignment is a *satisfiable formula*. The satisfiability problem asks whether a given Boolean formula is satisfiable; in formal-language terms,

$$\text{SAT} = \{ \langle \varphi \rangle : \varphi \text{ is a satisfiable Boolean formula} \}$$

As an example, the formula

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

has the satisfying assignment $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$, since

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

and thus this formula φ belongs to SAT.

Theorem: Satisfiability of Boolean formulas is NP-complete.

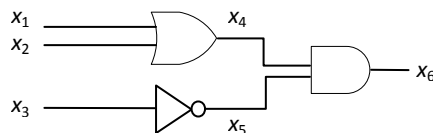
Proof:

1. SAT \in NP.

To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula φ can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression. This task is easily done in polynomial time. If the expression evaluates to 1, the formula is satisfiable.

\therefore SAT \in NP.

2. CIRCUIT-SAT \leq_P SAT.



The figure above illustrates the basic idea of the reduction from CIRCUIT-SAT to SAT. For each wire x_i in the circuit C , the formula φ has a variable x_i . The proper operation of a gate can now be expressed as a formula involving the variables of its incident wires.

For example, the operation of the output of AND gate is $x_6 \leftrightarrow (x_4 \wedge x_5)$.

The formula φ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate.

For the circuit in the figure, the formula is

$$\varphi = x_6 \wedge (x_4 \leftrightarrow (x_1 \wedge x_2)) \wedge (x_5 \leftrightarrow \neg x_3) \wedge (x_6 \leftrightarrow (x_4 \wedge x_5))$$

Given a circuit C , it is straightforward to produce such a formula φ in polynomial time.

Now, circuit C is satisfiable exactly when the formula φ is satisfiable. If C has a satisfying assignment, each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, the assignment of wire values to variables in φ makes each clause of φ evaluate to 1, and thus the conjunction of all evaluates to 1. Conversely, if there is an assignment that causes φ to evaluate to 1, the circuit C is satisfiable by an analogous argument.

Thus, we have shown that CIRCUIT-SAT \leq_P SAT, which completes the proof.

1.10 3-CNF Satisfiability

Many problems can be proved NP-complete by reduction from formula satisfiability. The reduction algorithm must handle any input formula, though, and this requirement can lead to a huge number of cases that must be considered. It is often desirable, therefore, to reduce from a restricted language of Boolean formulas, so that fewer cases need be considered. Of course, we must not restrict the language so much that it becomes polynomial-time solvable. One convenient language is 3-CNF satisfiability, or 3-CNF-SAT.

We define 3-CNF satisfiability using the following terms. 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.

For example, the Boolean formula

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

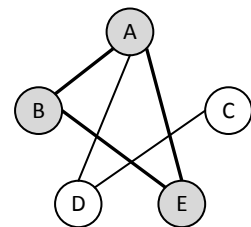
is in 3-CNF. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals x , $\neg x$, and $\neg x$.

1.11 The Clique Problem

A *clique* in an undirected graph $G = (V, E)$ is a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph).

The *size* of a clique is the number of vertices it contains.

For example, in the graph beside, the nodes A, B and E forms a clique of size 3.



The *clique problem* is the optimization problem of finding a clique of maximum size in a graph. As a decision problem, we ask simply whether a clique of a given size k exists in the graph. The formal definition is:

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

Theorem: The clique problem is NP-complete.

Proof:

1. $\text{CLIQUE} \in \text{NP}$.

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

2. $3\text{-CNF-SAT} \leq_p \text{CLIQUE}$.

The reduction algorithm begins with an instance of 3-CNF-SAT.

Let $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$ be a Boolean formula in 3-CNF with k clauses.

For $r = 1, 2, \dots, k$, 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 .

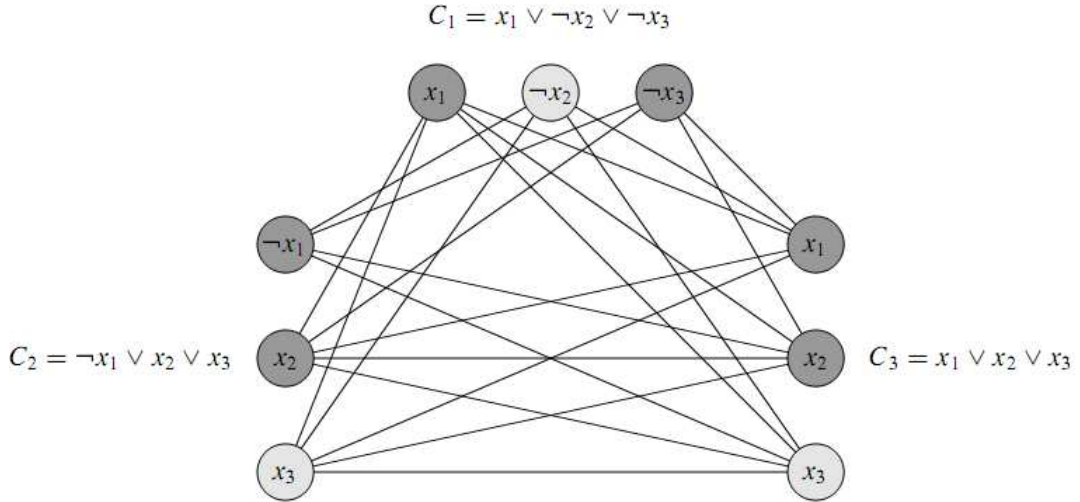
The graph $G = (V, E)$ is constructed as follows. For each clause $C_r = l_1^r \vee l_2^r \vee l_3^r$ in ϕ , we place a triple of vertices v_1^r, v_2^r and v_3^r into V . We put an edge between two vertices v_i^r and v_j^s if both of the following hold:

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

This graph can easily be computed 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 the following figure.



Now, we must show that this transformation of ϕ into G is a reduction.

[সহজ বাংলায়, আমাদেরকে প্রমাণ করতে হবে যে, ফর্মুলাতে যে কয়টা clause আছে, clique-এর সাইজ তত। এখন, ফর্মুলাতে দেখা যাচ্ছে যে, clause গুলো AND অবস্থায় আছে। তার মানে, প্রতিটা clause যদি 1 না হয়, তাহলে কখনোই ফর্মুলাটা satisfy করবে না (satisfy করবে না মানে ফর্মুলার রেজাল্ট 1 হবে না)। তাহলে, যদি প্রত্যেক clause-কে 1 হতে হয়, তবে কোন clause-এর মধ্যে অন্ততপক্ষে একটা literal-এর ভ্যালু অবশ্যই 1 হতে হবে। আমরা গ্রাফে প্রত্যেক clause থেকে exactly একটা করে নোড নিয়েছি যেটার corresponding literal-এর ভ্যালু 1 (এমনকি যদি দুটো literal-এর ভ্যালু 1 হয়ে থাকে, তারপরও আমরা কেবল একটাই নিয়েছি)। তাহলে নিশ্চিতভাবে যতগুলো clause আছে, clique-এর সাইজ তত। এখন, আমাদেরকে আবার প্রমাণ করতে হবে যে, এটা একটা clique, অর্থাৎ, এখানে নোডগুলো প্রত্যেকে প্রত্যেকের সাথে কানেক্টেড। দেখা যায় যে, আমরা all possible edge নিয়েছি (সেগুলো ছাড়া, যেখানে কোন literal তার complement literal-এর সাথে কানেক্ট করেছে; অর্থাৎ x -এর সাথে $\neg x$ কানেক্ট করে এমন edge নেওয়া হয় নাই)। সুতরাং, গ্রাফটা কানেক্টেড। অতএব, এটা একটা clique যার সাইজ number of clause-এর সমান।

উপরের কথাগুলো আবার উল্টোভাবেও বলতে হবে। অর্থাৎ, আমরা উপরে প্রমাণ করতে গিয়ে ফর্মুলা থেকে গ্রাফে এসেছি। এখন আবার গ্রাফ থেকে ফর্মুলাতে যেতে হবে। একই কথাগুলো উল্টো করে বলতে হবে। যেমন: গ্রাফের প্রতিটা নোডের জন্য corresponding একটা মাত্র clause-ই পাওয়া যায়; আবার, গ্রাফের প্রত্যেক নোডের corresponding literal-এর ভ্যালু 1; সুতরাং গ্রাফটা আসলেই ক্লিক এবং ঐ ফর্মুলা satisfied হলে গ্রাফের সাইজ হবে ঐ ফর্মুলার number of clause-এর সমান।

এই কথাগুলো ইংরেজিতে লিখলেই হবে। নিচে বইয়ের টেক্সটটা তুলে দেওয়া হল।]

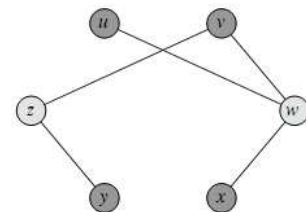
First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r . Picking one such “true” literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s are mapped 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.)

[Note that a satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 may be either 0 or 1 (don't care). This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3 , corresponding to the clique with lightly shaded vertices.]

1.12 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). 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.

For example, the graph in the figure beside has a vertex cover $\{w, z\}$ of size 2.

The *vertex-cover problem* is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to determine whether a graph has a vertex cover of a given size k . As a language, we define

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

1.13 The Hamiltonian Cycle Problem

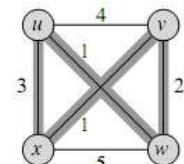
A *Hamiltonian cycle* of a directed graph $G = (V, E)$ is a simple cycle that contains each vertex in V .

The *Hamiltonian cycle problem* is to find an ordering of the vertices such that each vertex is visited exactly once.

1.14 The Traveling-Salesman Problem

In the traveling-salesman problem, which is closely related to the Hamiltonian-cycle problem, a salesman must visit n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a tour, or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is an integer cost $c(i, j)$ to travel from city i to city j , and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. For example, in the figure below, a minimum-cost tour is $\langle u, w, v, x, u \rangle$, with cost 7. The formal language for the corresponding decision problem is

$$\text{TSP} = \{ \langle G, c, k \rangle : G = (V, E) \text{ is a complete graph, } c \text{ is a function from } V \times V \rightarrow \mathbb{Z}, k \in \mathbb{Z}, \text{ and } G \text{ has a traveling-salesman tour with cost at most } k \}$$



Theorem: The Traveling-Salesman problem is NP-complete.

Proof:

1. TSP \in NP.

Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k . This process can certainly be done in polynomial time.

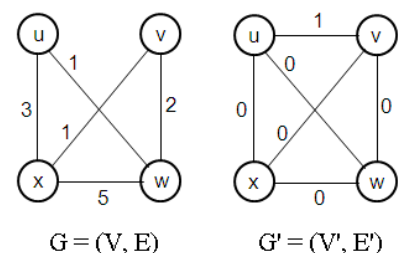
2. HAM-CYCLE \leq_p TSP.

Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V', E')$, where $E' = \{ (i, j) : i, j \in V \text{ and } i \neq j \}$, and we define the cost function c by

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

(Note that because G is undirected, it has no self-loops, and so $c(v, v) = 1$ for all vertices $v \in V$.) The instance of TSP is then $\langle G', c, 0 \rangle$, which is easily formed in polynomial time.

[An example of this construction is shown in the figure]



We now show that graph G has a Hamiltonian cycle if and only if graph G' has a tour of cost at most 0.

[সহজ বাংলায়, আমাদেরকে প্রমাণ করতে হবে যে, G গ্রাফে Hamiltonian cycle থাকবে যদি ও কেবল যদি G' গ্রাফে কোন tour থাকে যার সর্বোচ্চ cost হল 0। এখন, G' গ্রাফটা এমনভাবে তৈরি করা হয়েছে যেন G গ্রাফের Hamiltonian cycle-এ যে যে edge আছে, G' গ্রাফে সেসব edge-এর cost 0। সুতরাং, G গ্রাফের tour-কে G' গ্রাফে 0 cost-এ execute করা যায়।

আবার, বিপরীতভাবে, আমাদেরকে প্রমাণ করতে হবে যে, G' গ্রাফের সর্বোচ্চ 0 cost-এর tour যে সকল edge cover করে, সেসব edge হল G গ্রাফের edge। যেহেতু G' গ্রাফে কেবল 0 আর 1 cost-এর edge-ই আছে, তাই সেখানে কোন tour-এর cost 0 হতে হলে ঐ tour-এর সকল edge-এর cost অবশ্যই 0 হতে হবে। আর 0 cost-এর edge গুলো প্রকৃতপক্ষে G গ্রাফের edge। সুতরাং, G' গ্রাফের সর্বোচ্চ 0 cost-এর tour যে সকল edge cover করে, সেসব edge হল G গ্রাফের edge।

এই কথাগুলো ইংরেজিতে লিখলেই হবে। নিচে বইয়ের টেক্সটটা তুলে দেওয়া হল।]

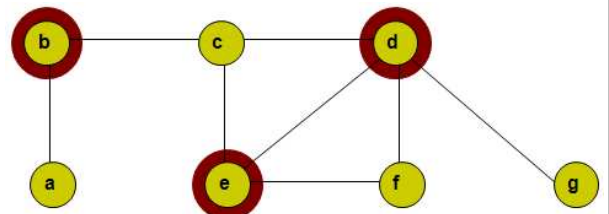
Suppose that graph G has a Hamiltonian cycle h . Each edge in h belongs to E and thus has cost 0 in G . Thus, h is a tour in G with cost 0.

Conversely, suppose that graph G has a tour h' of cost at most 0. Since the cost of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E . We conclude that h' is a Hamiltonian cycle in graph G .

APPROXIMATION ALGORITHMS

Concepts

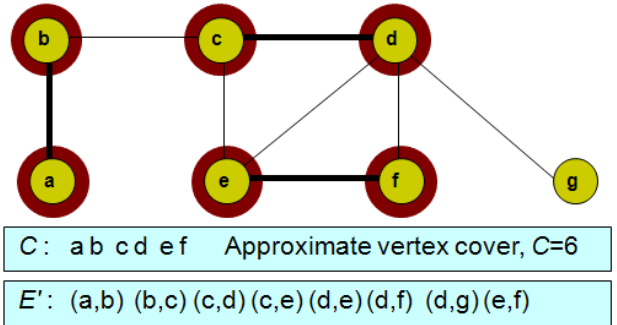
2.1	<p>Approaches to getting around NP-completeness</p> <ul style="list-style-type: none"> ➤ Exponential time may be acceptable for small inputs [Brute Force] ➤ Isolate special cases that can run in polynomial time [DC, GRDY, DP] ➤ Near-optimal solutions may be acceptable [Approximation Algorithms] 														
2.2	<p>Approximation Algorithms</p> <p>An algorithm that returns near-optimal solutions is called an approximation algorithm.</p>														
2.3	<p>Performance Ratios for Approximation Algorithms</p> <p>Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution. Depending on the problem, an optimal solution may be defined as one with maximum possible cost or one with minimum possible cost; that is, the problem may be either a maximization or a minimization problem.</p> <p>We say that an algorithm for a problem has an <i>approximation ratio</i> of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:</p> $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$ <p>We also call an algorithm that achieves an approximation ratio of $\rho(n)$ a $\rho(n)$-approximation algorithm. The definitions of approximation ratio and of $\rho(n)$-approximation algorithm apply for both minimization and maximization problems.</p> <p>For a maximization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.</p> <p>Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.</p> <p>Since all solutions are assumed to have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* < 1$ implies $C^*/C > 1$. Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.</p>														
2.4	<p>Approximation Algorithm for the Vertex-Cover Problem</p> <p>Consider the graph beside. By inspection, the optimal vertex cover is $\{b, d, e\}$ and the size of optimal solution, $C^* = 3$.</p> <p>An approximation algorithm for the vertex-cover problem is as follows:</p> <p style="margin-left: 20px;">APPROX-VERTEX-COVER(G)</p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%; padding: 2px 10px 2px 0;">1 $C \leftarrow \emptyset$</td> <td style="width: 50%; padding: 2px 10px 2px 0;">// C contains the result of vertex cover</td> </tr> <tr> <td style="padding: 2px 10px 2px 0;">2 $E' \leftarrow E[G]$</td> <td style="padding: 2px 10px 2px 0;">// initially all edges are stored here</td> </tr> <tr> <td style="padding: 2px 10px 2px 0;">3 while $E' \neq \emptyset$</td> <td style="padding: 2px 10px 2px 0;">// all edges that are not considered yet</td> </tr> <tr> <td style="padding: 2px 10px 2px 0;">4 do let (u, v) be an arbitrary edge of E'</td> <td style="padding: 2px 10px 2px 0;">// select an edge from E'</td> </tr> <tr> <td style="padding: 2px 10px 2px 0;">5 $C \leftarrow C \cup \{u, v\}$</td> <td style="padding: 2px 10px 2px 0;">// add vertices of the selected edge to C</td> </tr> <tr> <td style="padding: 2px 10px 2px 0;">6 remove from E' every edge incident on either u or v</td> <td style="padding: 2px 10px 2px 0;">// delete all edges related to the vertices u and v</td> </tr> <tr> <td style="padding: 2px 10px 2px 0;">7 return C</td> <td></td> </tr> </table>	1 $C \leftarrow \emptyset$	// C contains the result of vertex cover	2 $E' \leftarrow E[G]$	// initially all edges are stored here	3 while $E' \neq \emptyset$	// all edges that are not considered yet	4 do let (u, v) be an arbitrary edge of E'	// select an edge from E'	5 $C \leftarrow C \cup \{u, v\}$	// add vertices of the selected edge to C	6 remove from E' every edge incident on either u or v	// delete all edges related to the vertices u and v	7 return C	
1 $C \leftarrow \emptyset$	// C contains the result of vertex cover														
2 $E' \leftarrow E[G]$	// initially all edges are stored here														
3 while $E' \neq \emptyset$	// all edges that are not considered yet														
4 do let (u, v) be an arbitrary edge of E'	// select an edge from E'														
5 $C \leftarrow C \cup \{u, v\}$	// add vertices of the selected edge to C														
6 remove from E' every edge incident on either u or v	// delete all edges related to the vertices u and v														
7 return C															



Applying this algorithm for the graph above, we get the following graph along with the vertex cover (the initial contents of the array E' is also shown):

Analysis of the algorithm

- Running time:
 - $O(V+E)$: if we use adjacency list.
- 2-Approximation Algorithm
 - Minimization problem; $C^* = 3$; $C = 6$
 - Factor = $C/C^* = 2$



2.5 Theorem: APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm

[আমাদের টার্গেট হল এটা প্রমাণ করা যে, $C/C^* = 2$] এখন, general-ভাবে C আর C^* -এর মান কত হতে পারে? যেহেতু এটা একটা minimization problem, সুতরাং, আমাদেরকে জানতে হবে optimal solution তথা C^* -এর মান সর্বনিম্ন কত হতে পারে। সেই সাথে এটাও বের করতে হবে যে, approximate solution তথা C -এর মান সর্বোচ্চ কত হতে পারে (worst case বিবেচনা করতে হবে)। এ দুটো মান বের করার জন্য আমরা C আর C^* -কে edge-এর সাথে সম্পর্কযুক্ত করার চেষ্টা করব, কারণ আমরা প্রত্যেকবার একটা edge নিয়ে সেটা থেকেই vertex cover বের করার চেষ্টা করি।

প্রথমে দেখি C^* -এর মান সর্বনিম্ন কত হতে পারে। ধরি, আমরা প্রত্যেকবার যখন একটা করে edge নিই, তখন সেটা A অ্যাারেতে রাখি। এখন, no two edges in A share an endpoint! (যেমন, আমরা (a, b) edge নিয়ে A -তে রাখলে A -তে আর কোন edge থাকতে পারবে না যেখানে a অথবা b আছে।) কারণ, আমরা একটা edge নেওয়ার পর approximation algorithm-এর ৬ষ্ঠ লাইনে অন্য যত edge-এ ঐ edge-এর endpoint আছে, সেসব edge-কে আর নিই নি। সুতরাং, A -এর মধ্যে যেসব নোডের উল্লেখ থাকবে, সেগুলো unique। আবার, এমনটি কখনো হবে না যে, C^* -এর একটি নোড A -এর একাধিক edge-কে কাভার করে। তাহলে, C^* -এ অবশ্যই A -এর প্রত্যেক edge-এর জন্য কমপক্ষে একটি করে নোড থাকবে। তাহলে C^* -এর মান হবে $|C^*| \geq |A|$ ।

এখন দেখি C -এর মান সর্বোচ্চ কত হতে পারে। approximation algorithm অনুযায়ী A -তে যখন একটা edge নেওয়া হবে, তখন তার জন্য C -তে দুটো নোড নেওয়া হবে। তাহলে C -এর সর্বোচ্চ মান হতে পারে $|C| = 2|A|$ ।

এ দুটো মান combine করে পাওয়া যায়: $\frac{|C|}{|C^*|} \leq 2$]

The loop on lines 3–6 repeatedly picks an edge (u, v) from E' , adds its endpoints u and v to C , and deletes all edges in E' that are covered by either u or v . The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E' . Therefore, APPROX-VERTEX-COVER runs in polynomial time.

The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $E[G]$ has been covered by some vertex in C .

To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that were picked in line 4 of APPROX-VERTEX-COVER. In order to cover the edges in A , any vertex cover — in particular, an optimal cover C^* — must include at least one endpoint of each edge in A . No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6. Thus, no two edges in A are covered by the same vertex from C^* , and we have the lower bound:

$$|C^*| \geq |A| \dots(1) \text{ on the size of an optimal vertex cover.}$$

Each execution of line 4 picks an edge for which neither of its endpoints is already in C , yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2 |A| \dots(2)$$

Combining equations (1) and (2), we obtain

$$|C| = 2 |A| \leq 2 |C^*| ,$$

thereby proving the theorem.