34 NP-Completeness

Definition. ***Introduction***

1. ***polynomial-time algorithm*** *on input of size* n *the worst case running time is* $O(n^k)$ *for some constant* k.
2. ***Tractable vs Intractable*** *Polynomial time algorithms are tractable, problems requiring super-polynomial time is intractable, or hard*
3. ***NP-complete problems*** *a class of problem for which no polynomial-time algorithm has yet been discovered, and no one able to prove that no polynomial-time algorithm can exist for any one of them. ($N \neq NP$?)*
4. *Pairs of problem, one is solvable in polynomial time and the other is NP-complete*
   (a) ***Shortest vs. longest simple path*** *We can find shortest path in a directed graph* G *in* $O(VE)$ *time. Finding the longest path is difficult. Merely determining if a graph contain a simple path of at least a givne number of edges is NP-complete*
   (b) ***Euler tour vs. Hamiltonian cycle*** *A Euler tour of a connected, directed graph is a cycle that traverses each edge of* G *exactly once (able to visit each vertex more than once). algorithm for Euler tour in* $O(E)$ *time. A hamiltonian cycle of a directed graph* G *is a simple cycle that contains each vertex in* V. *Determining if a directed graph has a hamiltonian cycle is NP-complete.*
   (c) ***2-CNF satisfiability vs. 3-CNF satisfiability***
      i. *A* ***Boolean formula*** *contains*
         A. *variables whose values are 0 or 1*
         B. *boolean connectivs* $\wedge$, $\vee$, $\neg$
         C. *and parentheses*
      ii. *A boolean formula is* ***satisfiable*** *if there exists some assignment of 0 and 1 to its variables that cause it to evaluate to 1.*
      iii. *A boolean formula is in* ***k-conjunctive normal form*** *or* $k-CNF$ *if it is the ANDs of clauses or ORs of exactly* k *variables or their negations. (i.e.* $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ *is in 2-CNF with satisfying assignment of* $x_1 = 1$ $x_2 = 0$ $x_3 = 1$*)*

      We can determine in polynomial time if a 2-CNF formula is satisfiable but not for 3-CNF

Definition. ***Complexity claases: NP-completeness*** *(NPC)**, P **and** NP*

1. *class* ***P*** *consists of problems* ***solvable*** *in polynomial time (i.e. in* $O(n^k)$*)*
2. *class* ***NP*** *consists of problems* ***verifiable*** *in polynomial time, i.e. given a certificate of a solution, we could verify that the certificate is correct in time polynomial in size of input to the problem. For example,*

   (a) *For hamiltonian cycle problem, given a directed graph $G = (V, E)$ a certificate is a sequence $\langle v_1, v_2, \cdots, v_{|V|}\rangle$ of $|V|$ vertices. We could easily check in polynomial time and $(v_i, v_{i+1}) \in E$ for $i = 1, \cdots, |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well.*
   (b) *For 3-CNF, a certificate would be an assignment to variables, we could check in polynomial time that assignment satisfies the boolean formula*

   $P \subseteq NP$, i.e. any problem in P is also in NP, since if problem in P can be solved in polynomial time without even being supplied a certifiate. The open question is if P is a proper subset of NP.

3. *class **NPC** consists of problems that is*
   (a) *in **NP** and*
   (b) *is as hard as any problem in **NP***

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

Idea is if establish that a problem is NPC, proves its intractibility. Then we should spent time developing an approximation algorithm or solve a tractible special case

Definition. ***NPC problems*** *Proving that a problem is NPC is trying to make a statement about how hard a problem is (rather than how easy it is), i.e. no efficient algorithm is likely to exist.*

1. ***Decision problems vs. Optimization problems***
   (a) ***Optimization problems*** *A problem in which each feasible solution has an associated value, and we wish to find a feasible solution with the best value. (i.e.* SHORTEST-PATH *problem, find a path from u to v using fewest edges, is equivalent to single-pair shortest-path problem in unweighted, undirected graph.)*
   (b) ***Decision problem*** *A problem in which the answer is simply yes or no (1 or 0). Note NPC problems are confined to decision problems (i.e.* PATH *problem, where given a directed graph G, vertex u and v and $k \in \mathbb{I}$, ask if a path exist from u to v consisting of at most k edges)*
   (c) *We often can **cast** an optimization problem to a related decision problem. (i.e.* PATH *being a decision problem of* SHORTEST-PATH*).*
   (d) ***A corresponding decision problems is at least no harder than optimization problem*** *When proving an optimization problem is hard, we often can prove instead that the corresponding decision problem is hard. Since decision problem is at least no harder than the optimization problem, implies if an optimization problem is easy, its related decision problem is also easy (we can solve decision problem by solving the optimization problem) (i.e.* PATH *can be solved by solving* SHORTEST-PATH *and then compare number of edges found to the value of decision-problem parameter k. say if the value found is larger than k, then decision-problem yields yes)*

2. ***Reduction***

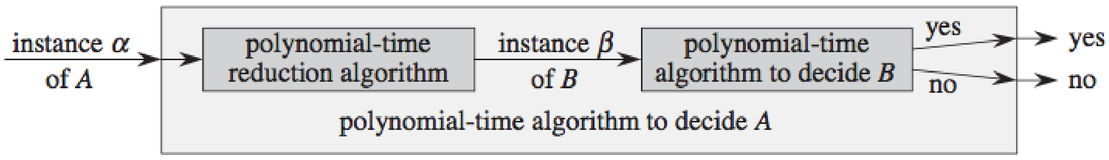
   (a) *Given a decision problem A for which we would want to solve in polynomial time. The input to a particular problem is an **instance** of that problem. (i.e. in* Path*, an instance would be a particular graph G, vertices u, v and $k \in \mathbb{I}$)*

   (b) ***Polynomial-time Decision algorithm*** *Given a decision problem B which already know how to solve in polynomial time.*

   (c) ***Polynomial time Reduction algorithm*** *Suppose we have a procedure that transforms any instance α of A into some instance β of B with*

      i. *transformation takes polynomial time*

      ii. *answers are the same, that is answer for α is yes if and only if the answer for β is also yes.*


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

   (d) *To solve A in polynomial time, we reduce problem A to B*

      i. *Given an instance α of problem A, use reduction algorithm to transform it into an instance β of B*

      ii. *Run polynomial-time decision algorithm for B on instance β Use answer for β as answer for α*

   (e) ***Usage*** *We can use polynomial time reduction in opposite way to **show that a problem is NPC**. Given a decision problem A for which we know no polynomial-time algorithm can exist and a polynomial-time reduction algorithm that transforms instanaces of A to instances of B. Then no polynomial-time algorithm exists for B*

      Proof. Proof by contradiction. Suppose that otherwise B has polynomial time algorithm, then along with the polynomial time reduction algorithm, we would have an algorithm (polynomial-time Algo for reducing A to B and deciding B) that solves A in polynomial time, which contradicts assumption. □

      The catch is we cannot be sure there is no polynomial-time algorithm for A. So we prove B is NPC on the assumption that problem A is also NPC.

3. ***A first NP-complete problem*** *We need a first NPC problem to prove a different problem as NPC with reduction. The problem is the **circuit-satisfiability** problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and wich to know if there exists some set of boolean inputs to this circuit that causes its output to 1.*

34.1 Polynomial time

Definition. *We consider polynomial time solvable problem tractible for philosophical reasons*

1. *Many problem has a small exponent $O(n^c)$ where c is relatively small*
2. *polynomial-time solvable problems usually remains solvable in polynomial time cross different models of computations (random-access machine, turing machine, parallel computer)*
3. *polynomials are closed under addition, multiplication, and composition, i.e. composite polynomial-time algorithm remains to be polynomial*

Definition. ***Abstract Problems***

1. ***Abstract problem*** *Q is a binary relation on a set I of problem* ***instances*** *and a set S of problem* **solutions**. *(i.e. an instance for* SHORTEST-PATH *is a triple (G, u, v), with solution as a sequence of vertices in graph. The problem itself is the relation that associates each instance to a shortest path in graph connecting the two vertices)*
2. ***Decision problem*** *is a function that maps the instance set I to the solution set $\{0, 1\}$ (i.e. for decision problem* PATH, *if $i = \langle G, u, v, k\rangle$ is an instance, then* $\text{PATH}(i) = 1$ *(yes) if a shortest path from u to v has at most k edges, and* 0 *(no) otherwise)*
3. ***Optimization problem*** *in which require some value to be minimized or maximized. We can recast an optimization problem as a decision problem that is no harder.*

Definition. ***Encoding***

1. ***Encoding*** *of a set S of abstract objects is a mapping e from S to thet set of binary strings.*
   (a) *Computer programs solves abstract decision problem by taking an encoding of a problem instance as input.*
   (b) *We use encodings to map abstract problems to concrete problems. Given an abstract decision problem $Q : I \to \{0, 1\}$, an encoding $e : I \to \{0, 1\}^*$ can induce a related concrete decision problem, which we denote $e(Q)$. If solution to abstract instance $i \in I$ has solution $Q(i) \in \{0, 1\}$, then solution to the concrete problem instance $e(i) \in \{0, 1\}^*$ is also in $Q(i)$*
   (c) *The concrete problem produces the same solutions as the abstract problem on binary-string instances that represent the encodings of abstract-problem instances.*
2. ***Concrete Problem*** *A problem whose instance set I is the set of binary strings.*
3. *An algorithm* **solves** *a concrete problem in $O(T(n))$ if when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$ time.*

4. ***Polynomial-time solvable*** *A concrete problem for which there exists an algorithm to solve it in $O(n^k)$ for some constant $k \in \mathbb{R}$*

5. ***Complexity class*** *P is a set of concrete decision problems that are polynomial-time solvable.*

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

7. ***Polynomially related*** *Two encoding e_1 and e_2 are polynomial related if there exists two polynomial-time computable function f_{12} and f_{21} such that for any $i \in I$ we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$. That is a polynomial time algorithm can compute the encoding $e_2(i)$ from encoding $e_1(i)$ and vice versa*

Lemma. *Let Q be an abstract decision problem on 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 $e_2(Q) \in P$*

Definition. ***A formal-language framework***

1. ***Alphabet*** *Σ is a finite set of symbols*

2. ***Language*** *L over Σ is any set of strings made up of symbols from Σ (i.e. if $\Sigma = \{0,1\}$, the set $L = \{10, 11, 101, 111\}$ is a language of binary representation of binary numbers)*

3. ***empty string*** *denoted by ϵ.* ***empty language*** *denoted by $\emptyset$ and* ***language of all strings*** *over Σ is denoted by $L = \Sigma^*$*

4. ***Complement*** $\overline{L} = \Sigma^* \setminus L$

5. ***Concatenation*** $L_1L_2 = \{x_1x_2 : x_1 \in L_1 \wedge x_2 \in L_2\}$

6. ***Closure (kleene star)*** *of a language L is $L^* = \{\epsilon\} \cup L \cup L^2 \cup \cdots$, where L^k is concatenation of L to itself k times*

7. *For any decision problem Q, we have the set of instances represented by Σ^* where $\Sigma = \{0,1\}$. Since Q is entirely characterized by input instances that produce a 1 (yes) we have*

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

   For example, Path *has the corresponding language*

$$\begin{aligned} \textsc{Path} = \{\langle G, u, v, k\rangle : & G \textit{ undirected graph} \\ & u, v \in V \\ & k \in \mathbb{Z} \\ & \textit{exists path from } u \textit{ to } v \textit{ in } G \\ & \textit{consisting of at most } k \textit{ edges}\} \end{aligned}$$

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

9. *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*

10. *A language L is* ***accepted in polynomial time*** *by 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 $O(n^k)$*

11. *A language L is* ***decided in polynomial time*** *by an algorithm A if there exsits a constant k such that for any length-n string $x \in \{0,1\}^*$, the algorithm correctly decides if $x \in L$ in time $O(n^k)$ (so an algorithm accepting a language needs only produce answer when provided string in L, but must accept or reject every string in $\{0,1\}^*$ if were to decide a language)*

12. ***alternative definition for class*** *P*

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

34.2 Polynomial-time verification

Example. For PATH problem, we can check if a certificate p in G has length of at most k. Verifying the certificate can be done in linear time, which takes as long as solving the problem from scratch (PATH). Since $Path \in P$, so verifying a certificate doesnt buy much time. But there are problems with no polynomial time algorithm for which verifying a certificate is easy.

Definition. ***Hamiltonian cycles***

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

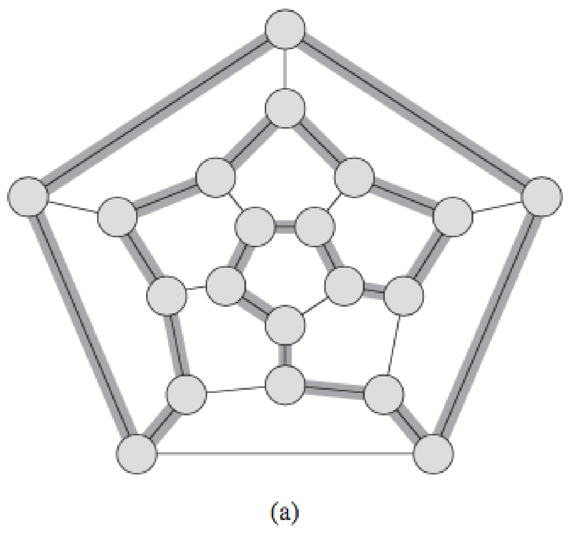


Figure 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

2. *A graph containing a hamiltonian cycle is* ***hamiltonian****(dodecahedron), otherwise it is* ***nonhamiltonian****.*

3. ***Hamiltonian cycle problem*** *asks if a graph G have a hamiltonian cycle*

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

4. ***Permutation algorithm*** *list all permutation of vertices of G, check each permutation to see if its a hamiltonian path. Runs in exponential time $\Omega(2^{\sqrt{n}})$*

Definition. ***Verification algorithm***

1. ***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* ***certificate***

2. *A* ***verifies*** *an input string x if there exists a certificate y such that $A(x, y) = 1$.*

3. *The* ***language verified*** *by a verification algorithm A is*

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

4. *Given a sequence of vertiecs (path), its easy to verify if the cycle is hamiltonian by checking whether*

(a) *it is a permutation of the vertices of V*

(b) *consecutive edges along the cycle actually exists in graph G*

The algorithm can be impl in $O(n^2)$ time. The certificate to the problem is a list of vertices in some hamiltonian cycle. If a graph is hamiltonian, the cycle itself can be verified.

Definition. ***Complexity class NP***

1. ***NP*** *is the class of languages that can be verified by a polynomial time algorithm. More preciesely, 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*

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

2. *We say that algorithm A* ***verifies*** *the language L in polynomial time*

3. Ham-Cycle $\in NP$

4. *if $L \in P$ then $L \in NP$, i.e. $P \subseteq NP$ (decidability implies verifiability)*

   Proof. If there is a polynomial time algorithm to decide L, the algorithm can be converted to a two-argument verification algorithm that ignores any certificates and accepts exactly those input strings it determines to be in L (i.e. just use the decision algorithm's output as verification) $\square$

5. *Unknown if $P = NP$, most believe they are not same class ($\exists L \in NP$ s.t. $L \notin P$). Its intuitive to think that its more difficult to solve a problem from scratch than to verify a clearly presented solution.*

6. ***co-NP*** *is a complexity class such that a problem $L \in$ co-NP if and only of $\overline{L} \in NP$. It is a class of problems for which there is a polynomial-time algorithm that can verify* ***no*** *instances*

7. *There is another open question as to if NP is closed under complement, i.e. if $NP =$ co-NP*

8. *P is closed under complement.*

Theorem. $P \subseteq$ co-NP

NP-Completeness and Reducibility

Definition. ***Intuition***

1. *belief in $P \neq NP$ comes from existence of NPC problems*

2. *If **any** NPC problem can be solved in polynomial time, then **every** problem in NP can be solved in polynomial-time*

3. HAM-CYCLE *is a NPC problem. If we can decide* HAM-CYCLE *in polynomial time, then we could solve every problem in NP in polynomial time. Moreover, if $NP - P \neq$, then* HAM-CYCLE $\in NP - P$

4. *NPC problems are the hardest language in NP.*

Definition. ***Reducibility***

1. *a problem Q can be reduced to another problem Q' if any instance of Q can be easily rephrased as an instance of Q', the solution to which provides a solution to the instance of Q. Q in a sense is **no harder to solve** than Q'*

2. ***polynomial-time reducible** A langauge 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 $x \in \{0,1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$.*

   (a) *The function f is the **reduction function**. The reduction function provides a polynomial-time mapping such that if $x \in L_1$ then $f(x) \in L_2$ and if $x_2 \notin L_1$ then $f(x) \notin L_2$*

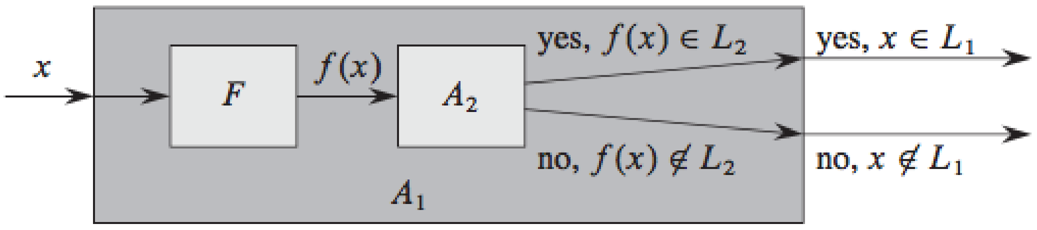
   (b) *The reduction function maps any instance x of the decision problem represented by language L_1 to an instance $f(x)$ of the problem represented by L_2. Implying providing an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$*

   (c) *a polynoimal-time algorithm F that computes f is a **reduction algorithms***

3. *Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polyonmial-time factor.*

   If $L_1 \leq_p L_2$,then L_1 is not more than a polynomial factor harder than L_2

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



Proof.
Let A_2 be a polynomial-time algorithm that decides L_2, and let F be a polynomial-time reduction algorithm that computes the reduction function f. Now we construct a polynomial-time algorithm A_1 that decides L_1. For any given input $x \in \{0,1\}^*$, algorithm A_1 uses F to transform x into $f(x)$ and then it uses A_2 to test whether $f(x) \in L_2$. The algorithm A_1 takes the output from algorithm A_2 and produces that answer as its own output. The correctness follows from the previous lemma. x and $f(x)$ are two different encodings of problem Q connected by a polynomial-time related encodings. Therefore, the solution set for A_2 is the solution set for A_1 □

Definition. ***NP-completeness***

1. ***NP-completeness*** *A language $L \subseteq \{0,1\}^*$ is NPC if*
   (a) $L \in NP$
   (b) $L' \leq_p L$ *for every* $L' \in NP$ *(NP-hard)*
2. ***NP-hard*** *A language that satisfies second property but not necessarily first property, then L is NP-hard.*
3. *A proof that a problem is NPC provides that it is intractible*
4. *We need to prove at least 1 problem as NPC, then we use polynomial-time reducibility as a tool to prove other problems to be NPC*

Theorem. *If any NP-complete problem (NPC) is polynomial-time solvable, then $P = NP$. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.*

Proof. Suppose $L \in P$, and also $L \in NPC$. For any $L' \in NP$, we have $L' \leq_p L$. By previous lemma, since $L \in P$ we have $L' \in P$. Therefore if any NPC L is polynomial-time solvable, we can reduce every NP problem to L and solve them in polynomial-time. Hence $P \subseteq NP$ and $NP \subseteq P$ therefore $P = NP$. This prove the first statement. The second statement is contrapositive of the first. □

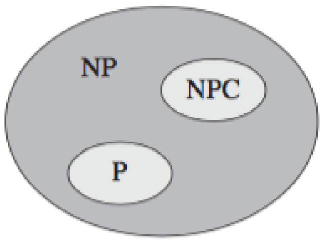


Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and P ∩ NPC = ∅.

Definition. ***Circuit satisfiability***

1. ***the first proved NPC problem***
2. ***boolean combinational element*** *any circuit element that has a constant number of boolean inputs and outputs that performs a well-defined function*
3. ***logic gates*** *are boolean combinational elements in circuit satisfiability problem (NOT, AND, OR gates). We can describe operation of each gates with a* ***truth table***
4. ***boolena combinational circuit*** *a circuit consisting of one or more boolean combinational elements interconnected by* ***wires****. A wire can connect the output of one element to input of another*
   (a) *contains no cycles*
   (b) ***Fan-out*** *is greatest number of inputs of gates of the same type to which the output can be safely connected.*
   (c) ***circuit input*** *If no element output is connected to a wire, the wire is a circuit input*
   (d) ***circuit output*** *If no element input is connected to a wire, the wire is a circuit output*
5. ***truth assignment*** *A truth assignment for a boolean combinational circuit is a set of boolean input values*
6. ***satisfiable*** *A one-output boolean combinational circuit is satisfiable if it has a satisfying assignment: a truth assignment that causes the output of the circuit to be 1*
7. ***circuit satisfiability problem*** *Given a boolean combinational circuit composed of AND, OR, NOT gate and ask if it is satisfiable.*

8. ***Naive solution*** *Check all possible assignments (2^n) to inputs (n) and see if at least one is a truth assignment that causes output to be 1.*

9. ***Configuration*** *Computer's memory holds entire state of computation (program, program counter, storage, ...) Any particular state of an instruction is a configuration. Execution of an instruction is a mapping from one configuration to another*

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

Proof. We shall provide a two-input, polynomial-time algorithm A that can verify CIRCUIT-SAT.

1. One of the inputs to A is an encoding of boolean combinational circuit C.

2. The other being a **certificate** corresponding to an assignment of boolean values to wires in C.

Construct algorithm A as follows

1. For each logic gates, checks value provided by certificate on each output wire provided by the certificate is computed correcty as a function of values on input wires

2. If output of entire circuit is 1, then the algorithm outputs 1. Otherwise 0.

For any satisfiable circuit C as input to A there exists a certificate whose length is polynomial in size of C and that causes A to output a 1. Whenever an unsatisfiable circuit is input, no certificate can fool A into believing that the circuit is satisfiable. A runs in polynomial time. Hence we can verify CIRCUIT-SAT in polynomial-time, hence CIRCUIT-SAT $\in NP$ □

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

Proof. Want to prove that $L \leq_p$ CIRCUIT-SAT for every $L \in NP$. Let L be any language in NP, we now describe a polynomial-time algorithm F computing a reduction function f that maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT. Since $L \in NP$, then must exists algorithm A that verifies L in polynomial time. Let $T(n)$ be worst case running time of algorithm A on length-n input string and let $k \geq 1$ be a constant such that $T(n) = O(n^k)$ and length of the certificate is $O(n^k)$. The verification algorithm A can be represented as a sequence of configurations. Each configuration can consists of

1. program for A

2. proram counter

3. auxiliary machine

4. input x

5. certificate y
6. working storage

The boolean combinational circuit M maps each configuration c_i to the next c_{i+1} starting at initial configuration c_0. A writes the output (0 or 1) to a designated location and the value never changes afterwards. So if the algorithm runs for at most $T(n)$ steps, the output appears as one of the bits in $c_{T(n)}$.

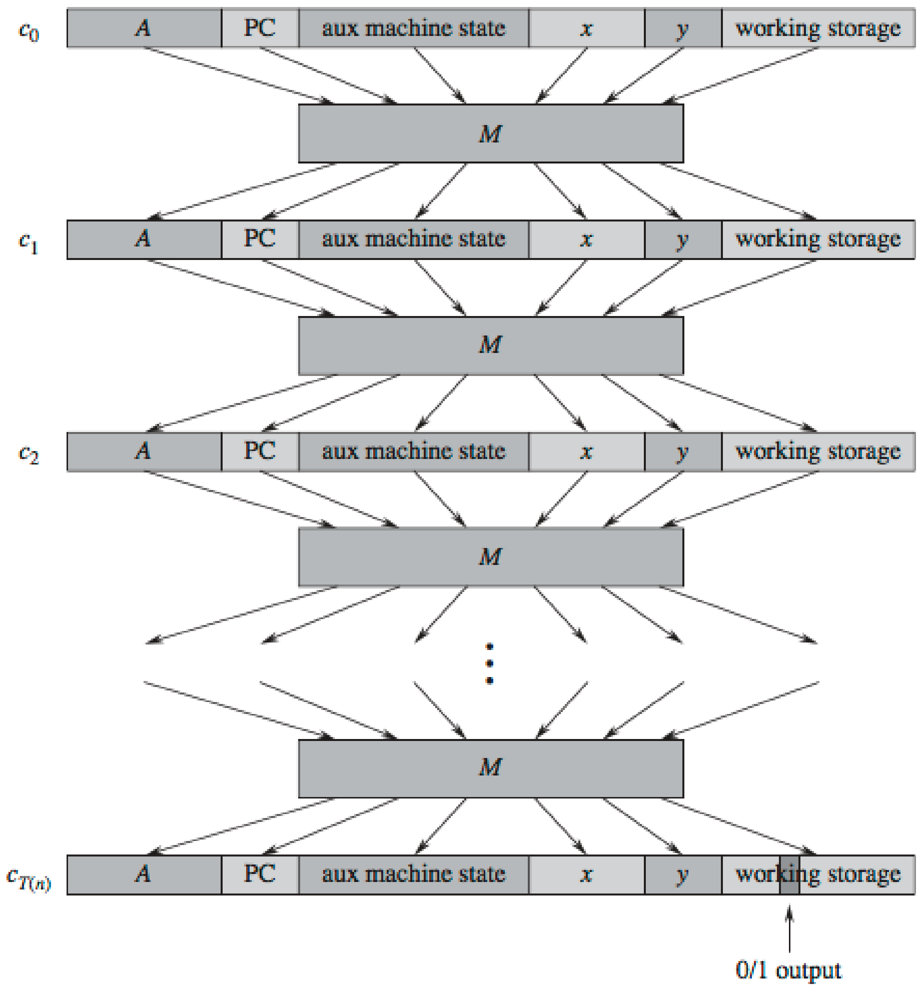


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. A boolean combinational circuit M maps each configuration to the next configuration. The output is a distinguished bit in the working storage.

The reduction algorithm F works by constructing a single combinational circuit that computes all configurations by a given initial state. Idea is to paste together $T(n)$ copies of circuit M. The output of i-th circuit feeds to input of $(i + 1)$-th circuit. So configurations, rather than being stored in memory, simply reside as values on wires connecting copies of

M. Now we prove that the reduction algorithm F computes a circuit $C = f(x)$ that is satisfiable if and only if there exists a certificate y such that $A(x, y) = 1$. When F obtains input x, it computes $n = |x|$ and constructs a combinational circuit C' consisting of $T(n)$ copies of M. The input to C' is an initial configuration corresponding to a computation on $A(x, y)$ and output is configuration $c_{T(n)}$. F modifies upon C' to construct $C = f(x)$ by

1. wiring inputs to C' corresponding to the program for A. the initial PC, input x, initial state of memory, and also a certificate y.

2. ignores all outputs from C', except for the one bit of $c_{T(n)}$ corresponding to output of A.

The resulting circuit C computes $C(y) = A(x, y)$ for any input y of length $O(n^k)$. Now we need to prove that

1. F correctly computes a reduction function f, i.e. show C is satisfiable if and only if there exists a certificate such that $A(x, y) = 1$

   Proof. Suppose there exists a certificate y of length $O(n^k)$ such that $A(x, y) = 1$. Then if apply the bits of y to inputs of $C = f(x)$, the output of C is $C(y) = A(x, y) = 1$. So if a certificate exists, then C is satisfiable. For other direction, suppose C is satisfiable. Then there exists an input y to C such that $C(y) = 1$, from which we conclude $A(x, y) = 1$ $\square$

2. Show F runs in polynomial time.

   Proof. the number of bits required to represent a configuration is polynomial in n.

   (a) The program for A itself has constant size, independent of length of its input x
   (b) length of input x is n
   (c) length of certificate y is $O(n^k)$
   (d) Since algorithm runs for at most $O(n^k)$ steps (verifier), the amount of work storage is polynomial in n as well

   The combinational circuit M implementing the hardware has size polynomial in length of a configuration, which is $O(n^k)$ hence size of M is polynomial in n. Since C consists of at most $t = O(n^k)$ copies of M, the reduction algorithm F can construct C from x in polynomial time. $\square$

So language Circuit-SAT is at least as hard as any language in NP. This is because we have directly reduced any language $L \in NP$ to Circuit-SAT $\square$

Theorem. *The circuit-satisfiability problem is NP-complete*

34.4 NP-completeness proofs

Lemma. *If L is a language such that $L' \leq_p L$ for some $L' \in NPC$, then L is NP-hard. If, in adidtion, $L \in NP$, then $L \in NPC$*

Proof. Since L' is NPC, for all $L'' \in NP$ we have $L'' \leq_p L'$. By transitivity, we have $L'' \leq_p L' \leq_p L$, so shown L is NP-hard. If $L \in NP$, we also have $L \in NPC$ by definition of NP-completeness. □

Definition. ***NPC proof techniques*** *By previous lemma, by reducing a known NPC language L' to L, we implicitly reduce every language in NP to L.*

1. *Prove $L \in NP$*
2. *Prove L is NP-hard, i.e. $L' \leq_p L$ for some $L' \in NPC$*
   (a) *Select a known NP-complete problem L'*
   (b) *Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L*
   (c) *Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$*
   (d) *Prove the algorithm computing f runs in polynomial time.*

Definition. ***Formula Satisfiability***

1. ***truth assignment*** *for a boolean formula ϕ is a set of values for the variables of ϕ*
2. ***satisfying assignment*** *is a truth assignment that causes it to evaluate to 1*
3. ***Satisfiable formula*** *is a formula with a satisfying assignment.*
4. ***Formula satisfiability problem*** *in terms language* SAT *as follows*
   (a) *n boolean variables $x_1, \cdots, x_n$*
   (b) *m boolean connectives: any boolean function with one or two inputs and one outputs, such as $\wedge \vee \neg \rightarrow \iff$*
   (c) *parentheses.*

   We can encode a boolean formula ϕ in a length that is polynomial in $n+m$. We ask if whether a given boolena formula is satisfiable.

$$\text{SAT} = \{\langle\phi\rangle : \phi \textit{ is satisfiable boolean formula }\}$$

5. ***A naive exponential time algoriothm*** *Check every single assignment (2^n) for a formula $\langle\phi\rangle$ of n variables. If length of $\langle\phi\rangle$ is polynomial in n, then checking every assignment requires $\Omega(2^n)$ time*

Theorem. *Satisfiability of boolean formulas is NP-complete Two part proof*

1. *Prove* SAT $\in NP$

   Proof. Need to show that a certificate consisting of a satisfying assignment from 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 can be done in polynomial time. If the expression evaluates to 1, then algorithm has verified that the formula is satisfiable. Hence SAT $\in NP$ $\square$

2. *Prove* SAT *is NP-hard*

   Proof. We can prove this by showing that CIRCUIT-SAT $\leq_p$ SAT. In other words, we need to show a reduction algorithm that reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time. The reduction algorithm which look at the gates that produces the circuit output and inductively express each of gates' inputs as formulas does not work. The problem arise from shared-subformula, in which gates whose output wires have fan-out of 2 or more can cause size of generated formula to grow exponentially. Have to find a smarter solution. Each wire x_i in circuit C has a variable x_i. We express how each gates operate as a small formula invovling varaibles of its incident wires. (i.e. AND gate $z \iff (x \wedge y)$) These small formulas are **clauses** The reduction algorihtm generates formula ϕ as the AND of

   (a) circuit-output variable (output -¿ 1), and
   (b) conjunction of clauses describing operation of each gates (so that gate logic is valid)

   Now we show that circuit C is satisfiable if and only if ϕ is satisfiable. If C has a satisfying assignment, then each wire of circuit has a well-defined value, and output of circuit is 1. Therefore, when we assign wire values to variables to ϕ, each clause of ϕ evaluates to 1, and thus conjunction of all clauses (for each gate) along with the final output evaluates to 1. Conversely, if some assignment causes ϕ to evaluate to 1, then circuit C is satisfiable by an analogous argument. Hence CIRCUIT-SAT $\leq_p$ SAT $\square$

Definition. ***3-CNF satisfiability***

1. ***motivation*** *formula satisfiability is very useful in proving many other NPC problems. But reduction algorithm must handle any input formula and this can lead to many number of cases, so might want a restrricted langauge of boolean formulas*

2. ***Literal*** *in a boolean formula is an occurrence of a variable or its negation.*

3. ***Conjunctive normal form (CNF)*** *A boolean formula is in conjunctive normal form if it is expressed as an AND of clauses, each of which is the OR of one or more literals.*

4. ***3-CNF*** *A boolean formula is in 3-CNF if each clause has exactly three distinct literals. For example*
$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4)$$

5. ***3-CNF satisfiability problem*** *Ask if a given boolean formula ϕ in 3-CNF is satisfiable.*

Theorem. *Satisfiability of boolean formulas in 3-CNF is NP-complete*

1. *Prove* 3-CNF-SAT $\in NP$

   Proof. The certificate of the problem is a truth assignment for the input formula ϕ. We provide the verifying algorithm as simply replacing each variable in the formula with values of the variable and evaluates the expression. This can be done in polynomial time. If the expression evalutes to 1, the algorithm verifies that the formula is satisfiable. Here we have shown that such algorithm can verify 3-CNF-SAT in polynomial time □

2. *Prove* SAT $\leq_p$ 3-CNF-SAT

   Proof. The reduction algorithm is as follows.

   (a) We construct a binary parse tree for the input formula ϕ, with literals as leaves and connectives as internal nodes. For cluases with more than 2 OR of literals, we can use associativity to parenthesize the expression fully so that every internal node in resulting tree has 1 or 2 children. Now similar to how we proved SAT, we rewrite ϕ as AND of the root variable and a conjunction of clauses describing the operation of each node (we introduce a variable y_i as output of each internal node, output of root as being y_i) Observe that ϕ' obtained is a conjunction of clauses ϕ'_i, each of which has at most 3 literals (2 children and internal node output).

   (b) Now we convert each clause ϕ' by evaluating all possible assignment to its variables. Write up truth table for each clause, and use rows in which outputs 0, we can rewrite the original clause in **disjunctive normal form (DNF)** or OR (handle different rows) of ANDs (handles a single row) of literals (input variables that may be either 1 or 0), which is equivalent to $\neg\phi'_i$. We negate DNF clauses and convert it into a CNF formula ϕ''_i by using demorgan's law to complement all literals, change ORs to ANDs, and change ANDs to ORs. Now we have converted each clause ϕ'_i of formula ϕ' into a CNF formula ϕ''_i and so ϕ' is equivalent to the CNF formula ϕ'' and moreover each has at most 3 literals

   (c) Now we transform the formula so that each clause has exactly 3 clauses. The formula ϕ''' has 2 auxiliary variables p and q/ For each cluase C_i of ϕ'', we

      i. if C_i has 3 distinct literals then simply include C_i as a clause of ϕ'''

ii. If C_i has 2 distinct literals, i.e. $C_i = (x_1 \vee x_2)$ then include $(x_1 \vee x_2 \vee p) \wedge (x_1 \vee x_2 \vee \neg p)$ as clauses of ϕ'''. Now matter values of p, one clause is evaluted to 1 and the other evalutes to $(x_1 \vee x_2)$

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

Now note 3-CNF formula ϕ''' is satisfiable if and only if ϕ is satisfiable. Each step preserves satisfiability. first step preserves satisfiability by construction of ϕ'. second step CNF formula ϕ'' is algebraically equivalent to ϕ'. The 3-CNF formula ϕ''' is effectively equivalent to ϕ'' since any assignment to p and q produces a formula that is algebraically equivalent.
Additionally, we have to show the reduction can be computed in polynomial time

(a) construction of ϕ' from ϕ introduce at most 1 variable and 1 clause per connective in ϕ.

(b) construction of ϕ'' from ϕ' introduce at most 8 clauses (since at most 2^3 rows in a truth table with 3 inputs, and at most 3 literals in a clause) into ϕ'' from ϕ'

(c) construction of ϕ''' from ϕ'' introduce at most 4 clauses (assuming it happens that each clause has only one literal)

So size of the resulting formula ϕ''' is polynomial in length of the original formula ϕ. So each of the constructions can easily be reduced in polynomial time $\square$

34.5 NP-complete problems

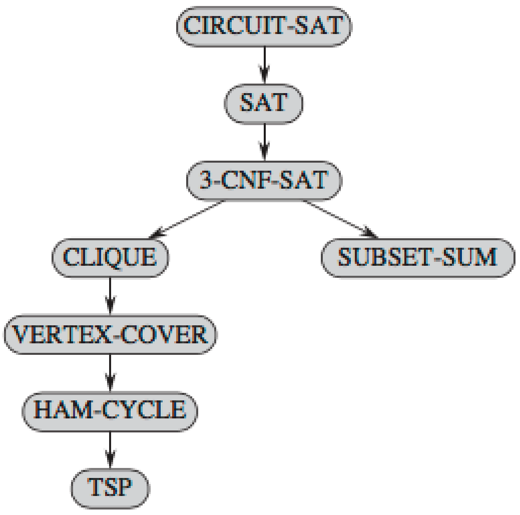


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

Definition. ***The Clique problem***

1. ***Clique*** *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
2. ***Size of Clique*** *is the number of vertices it contains*
3. ***Clique problem*** *is an optimization problem of finding a clique of maximum size in a graph. As a decision problem. we ask whether a clique of a given size* k *exsits in the graph*

$$\textsc{Clique} = \{\langle G, k\rangle : G \textit{ is a grpah containing a clique of size } k\}$$

4. ***Naive super-polynomial algorithm*** *to the decision problem. List all* k*-subsets of* V *and check each one to see whether if any one of them forms a clique. The running time is* $\Omega(k^2\binom{|V|}{k})$ *(*k^2 *because need to check every edge in the subgraph induced by the* k*-subset) If* k *is constant then runtime is polynomial. But in general,* $k \sim \frac{|V|}{2}$*, in which case the algorithm runs in superpolynomial time.*

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

1. *Prove* CLIQUE$\in NP$

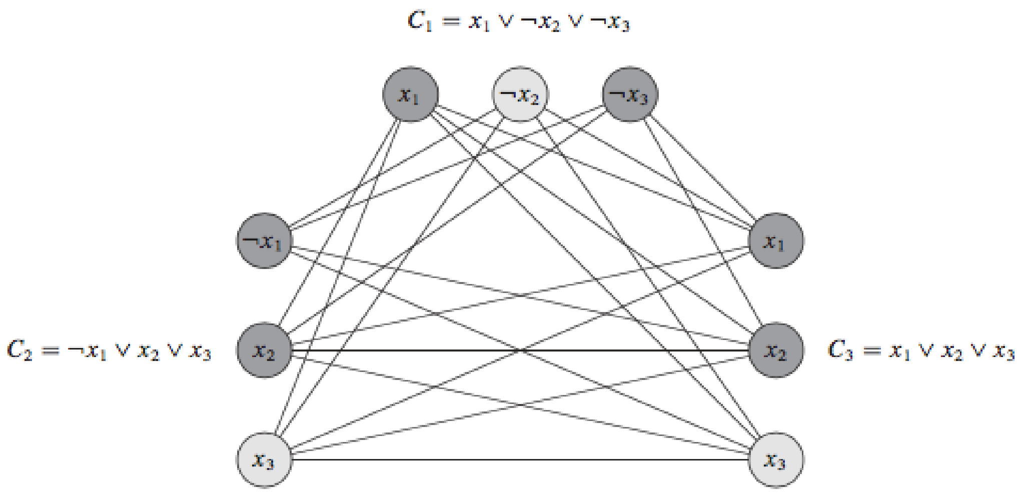
   Proof. For a given graph $G = (V, E)$, use set $V' \subseteq V$ of vertices as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E □

2. *Prove* 3-CNF-SAT $\leq_p$ CLIQUE

   Proof. The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be boolean formula in 3-CNF with k clauses. For $r = 1, 2, \cdots k$, each C_r has exactly 3 distinct literals l_1^r, l_2^r, l_3^r. We now construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k. The reduction algorithm works such that for each clause $C_r = (l_1^r, l_2^r, l_3^r)$ in ϕ place a triple of vertices v_1^r, v_2^r, v_3^r into V, put an **edge between 2 vertices** v_i^r and v_j^s if both of following holds

   (a) v_i^r and v_j^r are in different triples, i.e. $r \neq s$ and

   (b) their corresponding literals are consistent, i.e. l_i^r is not he negation of l_j^s



$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$C_2 = \neg x_1 \vee x_2 \vee x_3$
$\neg x_1$
x_2
x_3
$C_3 = x_1 \vee x_2 \vee x_3$
x_1
x_2
x_3

Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. 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.

The reduction algorithm runs in polynomial time (easy to see). Now we have to show that the transformation from ϕ to G is a reduction. Suppose ϕ has a satisfying assignment. Then each clause C_r has at least one literal l_i^r that is assigned 1, and each such literal corresponds to vertex v_i^r. Picking one such true literal from each clause yields

a set V' of k vertices. Claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literal l_i^r and l_j^s map to 1 by the given satisfying assignment, so the two literals cannot be complements. So by construction of G, edge (v_i^r, v_j^s) belongs to E. Conversely, suppose 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 can assign 1 to each literal l_i^r such that $v_i^r \in V'$. Note this assignment cannot assign 1 to a literal and 0 to its complement, since G contains no edges between inconsistent literals. Each clause is satisfied (by the fact that we have a k clique) so ϕ is satisfied. Note the proof works on Clique of a particular structure, i.e. Clique is NP-hard only in graphs in which the vertices are restricted to occur in triples and that there are no edges between vertices in the same triple. But this is suffice to show that Clique is NP-hard in general graph as well. Since if we had a polynomial-time algorithm that sovles clique on the general graph, it would solve clique on restricted graph.
Also reduction used the instance of 3-CNF-SAT but not the solution. Since we cannot decide the solution to 3-CNF-SAT in polynomial time, we cannot rely on knowing whether the formula ϕ is satisfiable. □

Vertex-Cover Problem

Definition. ***Vertex Cover Problem***

1. ***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. (every edge has at least 1 vertex in the cover) That is, each vertex covers its incident edges, and a vertex cover for G is a set of vertices that covers all edges in E.*

2. ***Size of vertex cover*** *is the number of vertices in it*

3. ***Vertex Cover problem*** *the optimization problem wants to find a vertex cover of minimum size in a given graph. The decision problem wants to determine if a graph has a vertex cover of a given size k.*

$$\textsc{Vertex-Cover} = \{\langle G, k\rangle : \textit{graph } G \textit{ has a vertex cover of size } k\}$$

4. ***Complement of graph*** *The complement of G, $\overline{G} = (V, \overline{E})$ where*

$$\overline{E} = \{(u, v) : u, v \in V \quad \wedge \quad u \neq v \quad \wedge \quad (u, v) \notin E\}$$

In other words, the $\overline{G}$ contains exactly those edges that are not in G

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

1. *Prove* $\textsc{Vertex-Cover} \in NP$

Proof. Given a graph $G = (V, E)$ and integer k, the certificate we choose is a vertex cover $V' \subseteq V$ itself. The verification algorithm checks

(a) $|V'| = k$

(b) for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$

This can be done in polynomial time easily. □

2. *Prove* Vertex-cover *is NPC by showing* Clique $\leq_p$ Vertex-Cover

Proof. The reduction algorithm takes an input instance of Clique problem $\langle G, k\rangle$. The reduction computes the complement $\overline{G}$ and outputs an instance of $\langle \overline{G}, |V| - k\rangle$ of the Vertex-Cover problem. Now we prove that the transformation is a reduction:

the graph G has a clique of size k iff the graph $\overline{G}$ has a vertex cover of size $|V| - k$

(a) Suppose G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\overline{G}$. Let $(u, v) \in \overline{E}$ be arbitrary. Then $(u, v) \notin E$. This implies at least one of u or v does not belong to the clique subset 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 $V - V'$ ($u, v \notin V' \to u, v \in V - V'$), this means that the edge (u, v) is covered by $V - V'$. Since choice of (u, v) is arbitrary, every edge of $\overline{E}$ is covered by a vertex in $V - V'$. set $V - V'$ has size $|V| - k$, which forms a vertex cover for $\overline{G}$

(b) Suppose $\overline{G}$ 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 (by property of vertex cover). The contrapositive is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, the $(u, v) \in E$. In other words, $V - V'$ is a clique, with size $|V| - |V'| = k$

□

Definition. ***The hamiltonian-cycle problem***

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

2. *A graph containing a hamiltonian cycle is* ***hamiltonian****, otherwise it is* ***nonhamiltonian****.*

3. ***Hamiltonian cycle problem*** *asks if a graph G have a hamiltonian cycle*

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

4. ***Permutation algorithm*** *list all permutation of vertices of G, check each permutation to see if its a hamiltonian path. Runs in exponential time $\Omega(2^{\sqrt{n}})$*

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

1. *Prove that* HAM-CYCLE $\in NP$

   Proof. Given a graph $G = (V, E)$, a certificate is a sequence of $|V|$ vertices that make up the hamiltonian cycle. The verification algorithm checks

   (a) the sequence contains each vertex V eactly once

   (b) there is an edge between each pair of consecutive vertices and between the first and last vertex

   The certificate can be verified in polynomial time □

2. *Prove* VERTEX-COVER $\leq_p$ HAM-CYCLE

   Proof. Given $\langle G, k\rangle$, we construt an undirected graph $G' = (V', E')$ that has a hamiltonian cycle if and only if G has a vertex cover of size k. For each edge $(u, v) \in E$ the graph G' will contain one copy of a **widget** (which enforces certain properties). Denote each vertex in W_{uv} by $[u, v, i]$ or $[v, u, i]$ where $1 \leq i \leq 6$ (totals to 12 vertices)

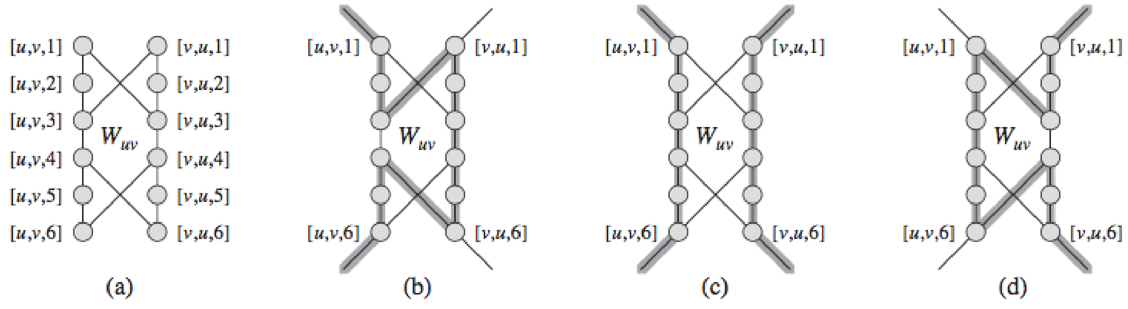


Figure 34.16 The widget used in reducing the vertex-cover problem to the hamiltonian-cycle problem. An edge (u, v) of graph G corresponds to widget W_{uv} in the graph G' created in the reduction. **(a)** The widget, with individual vertices labeled. **(b)–(d)** The shaded paths are the only possible ones through the widget that include all vertices, assuming that the only connections from the widget to the remainder of G' are through vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$.

   (a) limit connection bewteen widget and remainder of graph G', i.e. only $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, $[v, u, 6]$ will have edges incident from outside W_{uv}

   (b) Every hamiltonian cycle must traverse edges of W_{uv} in 3 possible ways

   We also include **selector vertices** $s_1, \cdots, s_k$. Use edges incident on selector vertices in G' to select the k vertices of the cover in G.

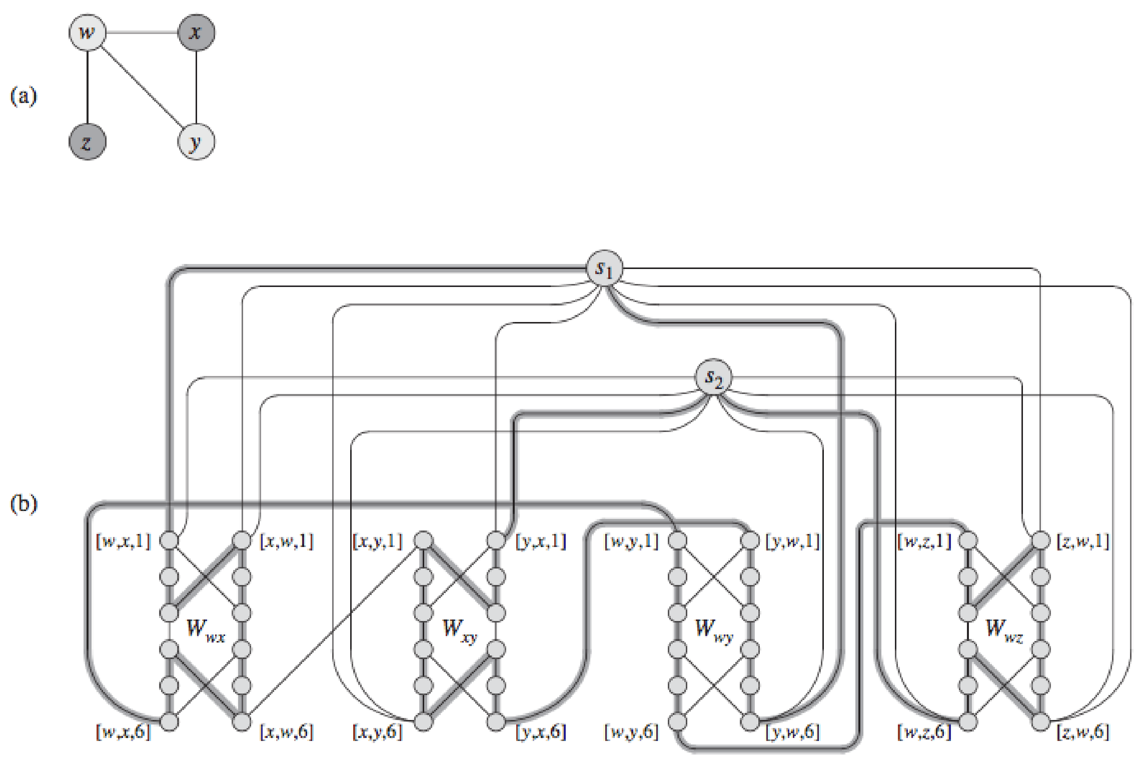


Figure 34.17 Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem. **(a)** An undirected graph G with a vertex cover of size 2, consisting of the lightly shaded vertices w and y. **(b)** The undirected graph G' produced by the reduction, with the hamiltonian 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.

… $\square$

Definition. ***Traveling salesman problem*** *Given a complete graph G with n vertices, a salesman wish to make a* ***tour*** *(hamiltonian cycle) visiting each vertex exactly once and finish at the starting vertex. For each $(i, j) \in E$, there is a nonnegative integer cost $c(i, j)$. The salesman wishes to make the tour whose total cost is minimum, where total cost is sum of individual costs along the edges of the tour*

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

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

1. *Prove* TSP $\in NP$

   Proof. Given an instance $\langle G, c, k\rangle$ or problem. The certificate is the sequence of n vertices in the tour. The verification algorithm checks

   (a) the sequence contains each vertex exactly once

   (b) sums summation of edge costs is at most k

   This can be done in polynomial time □

2. *Prove* Ham-Cycle $\leq_p$ TSP

   Proof. Let $G = (V, E)$ be an instance of Ham-Cycle. Now we construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$, where

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

   with cost function function

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

   The instance of TSP is then $\langle G', c, 0\rangle$. Note the transformation can be achieved in polynomial time. Now have to show that G has a hamiltonian cycle if and only if G' has a tour of cost at most 0.

   (a) Suppose G has a hamiltonian cycle h. Each edge in h belongs to E so has cost 0 in G'. Thus h is a tour in G' with cost 0.

   (b) Suppose G' has a tour h' of cost at most 0. Since the costs of edges in E' are 0 or 1, the cost of tour h' is exactly 0 implies each edge on the tour must have cost 0. Therefore h' contains only edges in E, so h' is a hamiltonian cycle in graph G

   □

3. *Together TSP is NPC. Note that the instance of NPC is in a restricted form, in that k is restricted to 0 and cost function c takes value of 0 or 1 only. This is fine since if we know how to solve the generalized TSP problem, we can solve the restrited TSP problem. By contraposition, if we cant solve the restricted TSP problem, we cannot solve generalized TSP*

Definition. ***Subset-sum problem*** *Given a finite set S of positive integers and an integer* ***target*** $t > 0$*. Ask whether there exists a subset $S' \subseteq S$ whose elements sum to t.*

$$\text{Subset-Sum} = \{\langle S, t\rangle : \textit{ there exists a subset } S' \subseteq S \textit{ such that } t = \sum_{s \in S'} s\}$$

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

1. *Prove* Subset-Sum $\in NP$

   Proof. Given an instance $\langle S, t\rangle$. Let S' be the certificate. A verification algorithm simply sums up elements in S' and check if it equates to t, which can be done in polynomial-time □

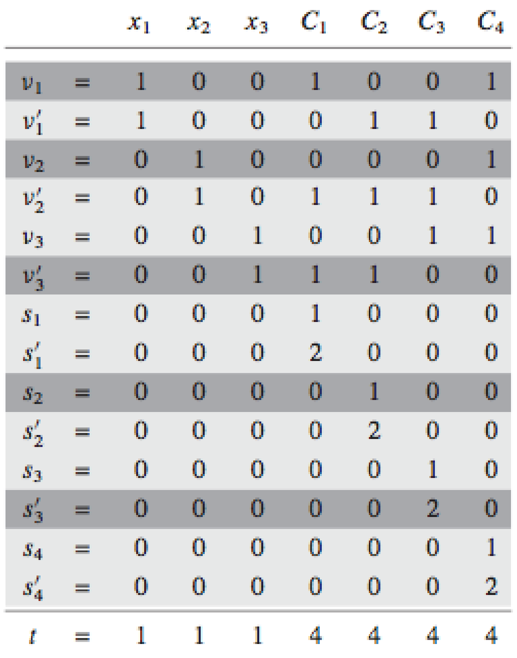
2. *Prove* 3-CNF-SAT $\leq_p$ Subset-Sum

   Proof. Given a 3-CNF formula ϕ over $x_1, x_2, \cdots, x_n$ with clauses $C_1, \cdots, C_k$, each containing exactly 3 distinct literals. The reduction algorithm constructs an instance $S, t\rangle$ such that ϕ is satisfiable if and only if there exists a subset of S whose sum is t. The reduction construct $2n+2k$ base 10 digits, each $n+k$ digits long, least significante k digits are labelled by clauses and most significant n digits are labled by variables

   (a) target t has 1 in each digit labeled by a variable and a 4 in each digit labled by a clause

   (b) For each x_i, set S contains two integers v_i and v'_i, with a 1 in the digit labeled by x_i and 0s in other variable digits. If $x_i \in C_j$, then digit labeled by C_j contains a 1. otherwise 0

   (c) For each C_j, set S contains 2 integers s_j and s'_j Each has 0 in all other digits other than the one labeled by C_j. For s_j there is a 1 in C_j digit, and for s'_j, there is a 2 in this digit. (slack variables)



		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

Figure 34.19 The reduction of 3-CNF-SAT to SUBSET-SUM. 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 $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$. The set S produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. The target t is 1114444. The subset $S' \subseteq S$ is lightly shaded, and it contains v_1', v_2', and v_3, corresponding to the satisfying assignment. It also contains slack variables s_1, s_1', s_2', s_3, s_4, and s_4' to achieve the target value of 4 in the digits labeled by C_1 through C_4.

The reduction can be done in polynomial time

(a) Suppose ϕ has a satisfying assignmen, For each i, if $x_i = 1$, then include v_i in S'. Otherwise, include v_i'. So we include either v_i or v_i' for each i but not both. We see that the variable digits sum up to 1s, which match n most significant digits of t. Note since all cluase is satisfied, there must be a literal in each clause with value 1. Therefore, each digit labeled by a clause (least significant bits) has at least one 1 contributing to its sum by a v_i or v_i' value in S'. Since there can be 1,2,3 literals that are 1s, each clause digit has a sum of 1,2,or 3. We can chieve the target of 4 in each digit labeled by clause C_j by including the appropriate nonempty subset of slack variables $\{s_j, s_j'\}$ in S'. So we have matched the target in all digits of the sum, the values of S' sum to t. So S' is a subset sum of S given target t

(b) Suppose there is a subset $S' \subseteq S$ that sums to t. S' must contain exactly one of

v_i and v_i' for each $i = 1, \cdots, n$ for otherwise the sum would not be 1. If $v_i \in S'$, set x_i to 1, otherwise $v_i' \in S'$, set x_i to 0. Claim that every clause C_j is satisfied by this assignment. Note to achieve a sum of 4 in digits labeled by C_j, there must be at least one of v_i or v_i' that has a 1 in the digit labeled by C_j, since inclusion of all slack s_j and s_j' sum to to 3 at most. If v_i has a 1 in C_j's position, then x_i appear in clause C_j, clause C_j is satisfied. Otherwise S' includes a v_i' that has a 1 in that position, then literal $\neg x_i$ appears in C_j. Since we set x_i to 0 by the assignment scheme described earlier, clause C_j is also satisfied. Thus all clauses C_j is satisifed, this completes the proof

$\square$