The decision problem

In computer science we deal with a huge variety of problems. The theory of computation seeks to minimize this complexity by instead studying exactly one problem, called the *decision problem*. The decision problem is sufficiently rich and complex to serve as a means for studying many important questions in the theory of computation.

Here are the definitions that play a role in the decision problem.

Definition An *alphabet* Σ is a set of characters.

Definition The *universe* Σ^* is the set of all strings x that can be formed from the characters in the alphabet Σ.

Definition A *language* L is a set of strings in Σ^*.

The *decision problem* for a language L is the problem of determining whether or not an arbitrary string x in Σ^* is in L.

Some examples

Languages can differ considerably in complexity. Here are some examples, ranging from simple languages to more complex languages. In the examples below

In all of the examples below $\Sigma = \{'0','1'\}$. The notation a^k stands for a sequence of k copies of the letter a.

$L_1 = \{ x \mid x \text{ contains an even number of 1s} \}$

$L_2 = \{ 0^n 1^n \mid n \geq 0 \}$

$L_3 = \{ 0^n 1^{n^2} \mid n \geq 0 \}$

$L_4 = \{ 1^p \mid p \text{ is a prime number} \}$

Deciders

In the theory of computation we imagine constructing algorithms that can solve the decision problem for various languages and we study the runtime efficiency of those algorithms. The runtime efficiency of an algorithm that decides a language is measured as a function of the length of the string the algorithm is decide.

Here is an example. A fairly obvious decision algorithm for the language

$L_2 = \{ 0^n 1^n \mid n \geq 0 \}$

would work something like this:

1. Make one pass through x to confirm that x is of the form $0^j 1^k$.
2. Make a second pass through x that counts the number of 0s and 1s in x.
3. Check that the number of 0s matches the number of 1s.

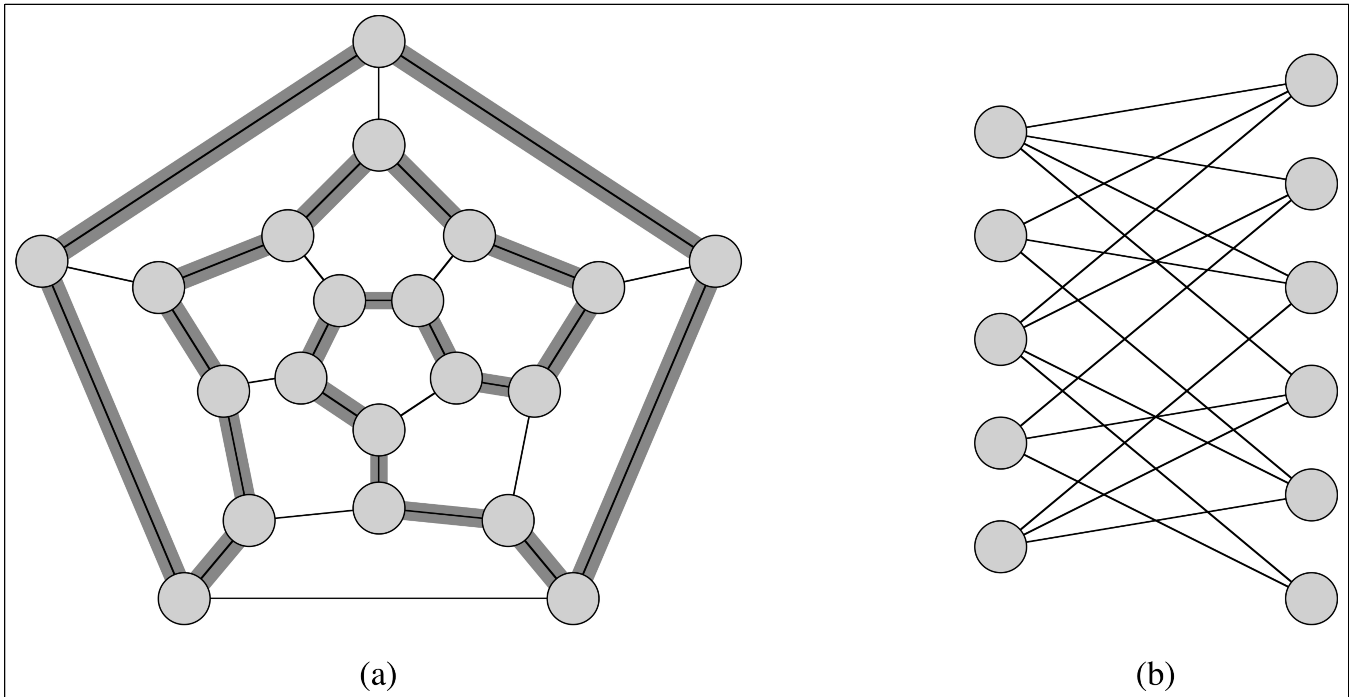
The runtime complexity of this algorithm is $O(|x|)$, which is linear in the length of the input string x.

Converting problems to languages

The language decision problem is sufficiently flexible to serve as a means for studying many problems in computer science. Here is a typical example.

A standard problem in graph theory is the Hamiltonian cycle problem. In this problem you are given a directed graph G and you are asked to determine whether or not the graph has a *Hamiltonian cycle*, which is a cycle in the graph that visits each vertex in the graph exactly once.

The picture below shows an example of a graph that does have such a cycle and one that does not.



The Hamiltonian cycle problem can be rephrased as a language decision problem involving this language:

HAM-CYCLE = {<*G*> | *G* encodes a graph and that graph contains a Hamiltonian cycle. }

The class P

Languages can vary considerably in their complexity, so the theory of computation divides languages into broad classes based on their complexity. The first of these classes is the class P, the class of languages decidable in polynomial time.

Definition A language L is in P if there exists a decision algorithm that can decide whether or not an input string x is in L in time $O(|x|^c)$ for some positive integer c.

The class NP

The problem with the definition of the class P is that it hinges on having a polynomial time algorithm to decide the language. What if you don't know how to build a polynomial time decider? This is a real issue, because many problems in computer science, including the Hamiltonian path problem, don't currently have efficient decision algorithms.

Many problems, like the Hamiltonian cycle problem, have fairly obvious inefficient decision algorithms. For example, here is a pretty crudy decider for *HAM-CYCLE*.

On input <G>:

1. Check that <G> encodes a graph.
2. Number the vertices in G from 1 to $|V|$.
3. Generate every possible permutation p of the list $(1,2,...,|V|)$.
4. For each permutation p check to see if there is a path in G that passes through the listed vertices in the order given.
5. If you find a permutation that works, return true.
6. If you find no permutation that works, return false.

Since this decider does a brute force search through a space of permutations it will take time that is

exponential in the size of the input graph G.

What do we do in cases where we can't construct a polynomial time decider for a language? Do we give up and throw up our hands and say that the language is not in P? How do know that next week some brilliant computer scientist somewhere won't come up with a polynomial time algorithm to solve the problem?

After studying a range of problems in computer science for many decades, computer scientists have come to suspect that certain problems like the Hamiltonian path problem will never have polynomial time solutions. The only problem with this at this point in time is that no one knows how to prove that certain problems will never be decidable in polynomial time.

Computer scientists have responded to this conundrum by trying to make a more detailed study of problem types. As part of this study they have introduced classes of problems that are "just outside of P" in an effort to understand what separates languages in P from languages that are not in P.

The most important of these new classes is the class NP. The N in NP stands for nondeterministic. Languages in NP are decidable in polynomial time provided we are allowed to use a limited amount of nondeterministic behavior in the decider. For example, here is a nondeterministic, polynomial time decider for *HAM-CYCLE*:

On input <G>:

1. Check that <G> encodes a graph.
2. Number the vertices in G from 1 to $|V|$.
3. Guess a permutation p of the list $(1,2,...,|V|)$.
4. Check to see if there is a path in G that passes through the listed vertices in the permutation p.
5. If there is such a path, return true.
6. Return false.

Nondeterministic behavior is a strange thing to make use of, because no real computer scientist would develop an algorithm that includes a step that says "now guess an answer".

Fortunately, there is a way to work around the use of nondeterminism in an algorithm. The method is to essentially make the guess external to the algorithm. This leads to the use of the concept of a *certificate*. A certificate is some external piece of data that the algorithm can use to help it solve the decision problem. This leads to the following alternative definition for the class NP:

Definition A language L is in the class NP if there exists an algorithm that takes a string x and a certificate string y as its input and in polynomial time uses both x and the certificate to determine whether or not x is in L.

Here is a concrete example. The language *HAM-CYCLE* is in the class NP. The certificate we use is the list of vertices in the Hamiltonian cycle. Given an input string $x = <G>$, we use the certificate to verify that the list of vertices forms a valid Hamiltonian cycle in G. This verification is relatively straightforward and can be carried out in time $O(|V|)$, which is linear in the size of the inputs.

NP-complete languages

The next step in the study of classes of languages it to try to find the outer boundary of the class NP. To find this, we need someway to measure the relatively difficulty of pairs of languages.

A mechanism that allows us to compare two languages and effectively say that one language is "harder" than another is the relation of *polynomial time reduction*.

Definition A language A is polynomial time reducible to a language B (written $A \leq_P B$) if there exists a polynomial time algorithm f that maps strings x in A to strings $f(x)$ in B. More specifically, we must have

$$f(x) \in B \text{ if an only if } x \in A$$

The relationship of polynomial time reducibility does in fact work as a measure of relative difficulty. The

main support for this is the following lemma:

Lemma If B is in the class P and $A \leq_P B$ then A is also in P.

The proof of this is as follows. Since B is in P, there must be an algorithm d that decides membership in B in polynomial time. Here then is a decider for A:

On input x:

1. Compute $f(x)$. (This takes time polynomial in the length of x.)
2. Run d on $f(x)$. (This takes time polynomial in the length of $f(x)$.)
3. Return what d returned.

A variant on this lemma is the observation that if A is not in P and $A \leq_P B$ then B can not possibly be in P, since we could use the construction above to make a polynomial time decider for A if B had a polynomial time decider. This observation would be a handy way to show that certain problems can not possibly have polynomial time solutions: all we would need is one instance of a language which is not in P and a way to connect that problem to others via polynomial time mappings. The only problem with this approach is that no one has found a way to show that there is some language in NP that can not possibly be in P. (This is the famous P $\neq$ NP problem.)

NP-complete problems

Since no one has found a way to show that there is a language in NP that is not in P, the theory of computation has had to fall back on an alternative strategy. In this stategy we seek out and study the "hardest" problems in NP. The following definition seeks to formalize this concept of a "hard" NP problem.

Definition A language L is said to be *NP-complete* if L is in NP and $M \leq_P L$ for all other languages M in NP.

Since the relation $\leq_P$ is a transitive relation, we also have the following

Lemma If a language L is NP-complete and $L \leq_P M$ for some other language M in NP, then M is also NP-complete.

Researchers have actually succeeded in finding many languages that are NP-complete. What this requires is finding a first problem that is NP-complete and then using the lemma to connect that initial problem to a chain of other problems. In the section below I will show several examples from the text that illustrate how to set up the mappings.

An initial NP-complete problem

There are several problems one can use as a starting point for constructing a chain of NP-complete problems. The hard part is showing that the first problem in the chain is NP-complete. This is somewhat beyond the scope of the discussion here: if you take the Theory of Computation course next year I will show a proof that a particular language is NP-complete.

For now, I will select one known NP-complete language as a starting point and then show how to map that initial problem to a chain of others.

The initial language I will use is known as 3-*CNF-SAT*. Strings in 3-*CNF-SAT* are logical formulas made up variables and logical operators. The formulas are limited to a particular structure: each formula is made up of the logical and of several clauses, and each clause is composed of exactly three terms combined with logical ors. The terms must be either variables or their negations. Here is an example of logical formula that meets these format rules:

$$(a \vee \neg b \vee d) \wedge (\neg a \vee \neg b \vee \neg c) \wedge (b \vee c \vee d) \wedge (\neg a \vee c \vee \neg d)$$

A formula is *satisfiable* if there is some assignment of truth values to the variables that causes the entire formula to evaluate to true. For example, the formula above is satisfiable by $a = T, b = T, c = F, d = F$.

CLIQUE is NP-complete

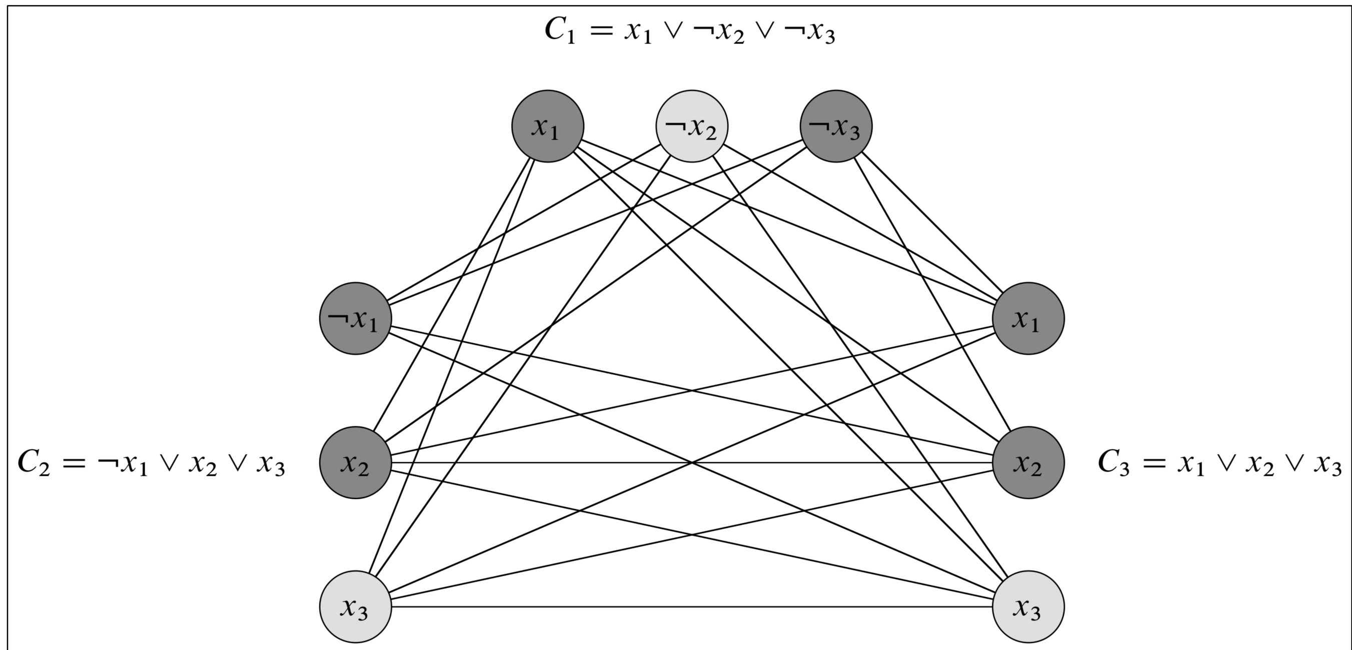
In an undirected graph G a *clique* of size k is a subset of k vertices where every vertex is connected to every

other vertex in the group. The language

$$CLIQUE = \{<G,k> \mid G \textbf{ contains a clique of size } k\}$$

is known to be NP-complete. *CLIQUE* is in NP because a certificate for a particular graph is the list of k vertices that form a clique. We can check in polynomial time that each vertex in the subset is connected to each one of the others. To show that *CLIQUE* is NP-complete, we show that $3\text{-}CNF\text{-}SAT \leq_P CLIQUE$.

The picture below shows how this mapping works. Given any formula in $3\text{-}CNF\text{-}SAT$ we construct a special graph. The graph is composed of clusters of vertices, with one cluster for clause in a formula. The vertices in clusters are connected to all of the vertices in other clusters that they are compatible with. To vertices are compatible if they use different variables or if they connect to terms with the same variables or two terms with negations of the same variables.



If we set k equal to the number of clauses, we see that the formula has a satisfying assignment if and only if the graph we constructed from the formula has a clique of size k. To make a satisfying assignment we must have at least one term in each clause that evaluates to true. If we select one true term in each clause the corresponding vertices in the graph must form a clique of size k because all of the vertices we have selected are compatible with each other and hence must have edges in the graph connecting them. Conversely, any clique of size k in the graph must use exactly one vertex from each group, since vertices in groups are not connected to each other. If we assign true values to terms whose vertices are in the clique we end up with a satisfying assignment for the original formula.

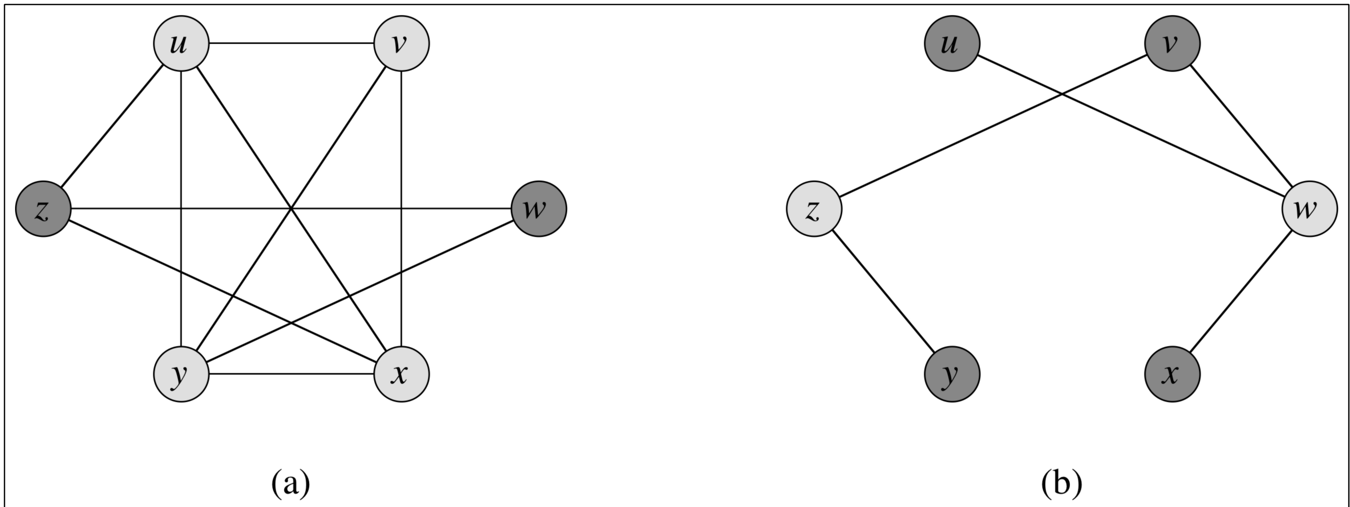
VERTEX-COVER

A vertex cover in a graph is a set of vertices that *cover* the edges: each edge in the graph touches at least one of the vertices in the cover. This leads to a language

$$VERTEX\text{-}COVER = \{<G,k> \mid G \textbf{ has a vertex cover of size } k\}$$

We can show that *VERTEX-COVER* is NP-complete via the reduction $CLIQUE \leq_P VERTEX\text{-}COVER$.

The picture below shows how the mapping works.

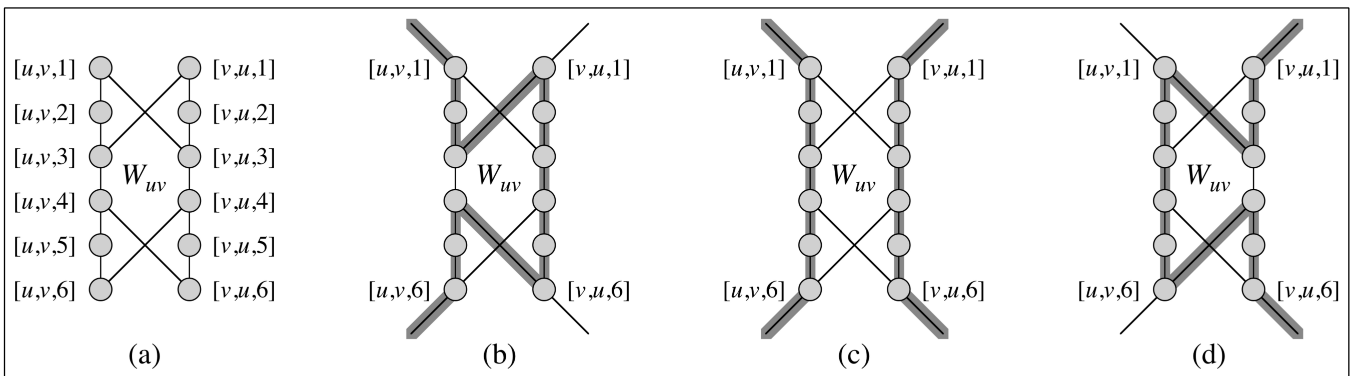


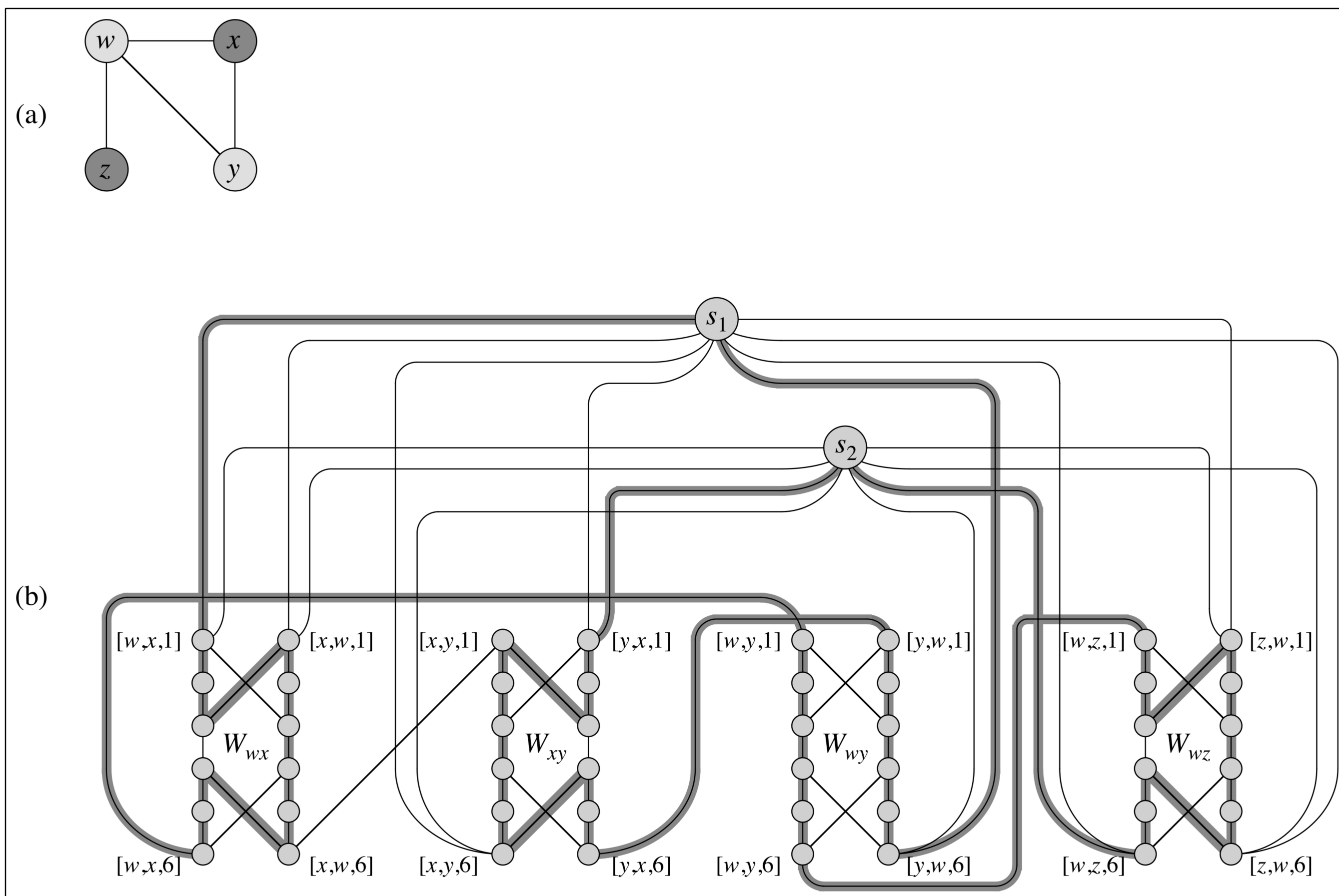
The graph on the left is a graph with a clique of a particular size. In the graph on the left there is a clique of size 4 made up of the light gray vertices. We form the graph on the right by using the same set of vertices but replacing the edges by the *complement* of the set of edges in the original graph. The new graph will have an edge between two vertices only if the original graph did not have an edge connecting those two vertices.

Suppose that the original graph has a clique of size k. We claim that the set of all vertices not in that clique in the complement graph form a vertex cover of size $|V|$ - k. Consider any edge in the new graph. This edge does not connect two vertices in the clique, since those vertices are all fully connected and edges that connect them would have been removed when we formed the complement. Thus, at least one of the two vertices connected by that edge must be outside the clique, and that vertex can cover that edge. Conversely, suppose the new graph has a vertex cover of size $|V|$ - k. I claim that the set of vertices not in the cover must have formed a clique in the original graph, because the new graph contains no edges connecting those vertices (if it did, that edge would not be covered). When we go backward from the complement graph to the original, all of the vertices not in the cover will have edges restored between them and we will be back to having a clique.

HAM-CYCLE

Finally, we can show that *HAM-CYCLE* is NP-complete via a reduction from *VERTEX-COVER*. The reduction is very complex, and is covered in detail in chapter 34. Below are the two key pictures that illustrate how the construction works.




(a)
w
x
z
y
(b)
s1
s2
[w,x,1]
[x,w,1]
[x,y,1]
[y,x,1]
[w,y,1]
[y,w,1]
[w,z,1]
[z,w,1]
W_wx
W_xy
W_wy
W_wz
[w,x,6]
[x,w,6]
[x,y,6]
[y,x,6]
[w,y,6]
[y,w,6]
[w,z,6]
[z,w,6]