CS 624
Lecture 16: NP-Complete Problems

1 The classes P and NP

1.1 Problems and instances

Technically, a *problem* is a language. We'll be precise about this later, when we need to. But for now, we can simply say that a problem is pretty much what you think it is, and it has *instances*. For instance, there is the problem of SORTING. The instances of this problem are sets of elements that can be compared. The result of sorting an instance is the the same set, put in order. There are various algorithms (and we have seen some good ones) that can solve this problem, and they all run in time $O(n^2)$—and the good ones run in $O(n \log n)$, where n is the *size* of the instance. In this case, by the *size* of the instance, we mean the number of elements in the set to be sorted.

Another very famous and very important problem is the TRAVELING SALESMAN PROBLEM. Each instance of this problem is a connected weighted undirected graph. The problem is to find the minimum-weight simple cycle[1] that visits each vertex in the graph. This problem has been extensively studied, but there do not (as of today) appear to be any $O(n^2)$ algorithms that solve it. (Here, n is the size of the description of the graph. We could take $n = |V| + |E|$, for instance.) In fact, there are no known $O(n^3)$ algorithms for this problem, or even $O(n^k)$ algorithms for any fixed k. So it's a very difficult problem.

1.2 Polynomial-time algorithms

It's important to be clear about what we have been calling the *size* of a problem instance. Since the beginning of the course, we have talked about problems that are $O(n)$ or $O(n^2)$, or $O(n \log n)$, and so on. When we talk like this, n is some measure of the size of the problem. As above, if we are talking about sorting a set, then n will be the number of elements of the set. If we are talking

[1] A simple cycle in a graph is a path that ends where it starts and doesn't include any vertex twice (other than the start vertex, which is the same as the end vertex; and by convention we say that this vertex also occurs only once). There is one tiny weakness with this definition: If you have a graph consisting of two points (say a and b, connected by one edge), then according to this definition, the path $a \to b \to a$ is a simple cycle. Clearly this is not what is intended. There are a couple of ways of dealing with this:

1. Add the condition that no edge occurs twice on the path. If the path has more than 2 nodes, then this condition is automatically satisfied so long as the condition on nodes is satisfied. So the condition really only applies to one very small graph.
2. Add the condition that the path has at least 3 distinct nodes. That's what our textbook does.

I really don't think this question is worth much thought. None of the problems we will deal with have any interest when applied to the graph with two nodes.

about a graph $G = (V, E)$ as in the TRAVELING SALESMAN PROBLEM, then it is reasonable to let n be something like $|V| + |E|$. And so on.

In general, a problem instance a is *encoded* in some way, and n is just another name for $|a|$, which is the length of the encoding.

Some problems involve numbers—see for instance the SUBSET SUM problem below in Section 3.5 (page 20). For such problems, you might think that (just as the size of an instance of a problem involving graphs is the number of edges and vertices in the graph for that problem instance) the size of an instance of a problem involving integers is just how many integers there are in that instance. Alternatively, you might think that the size of an instance is the sum of all the absolute values of the numbers in that instance. It turns out that neither of these "sizes" is a good measure of the size of such a problem.

What is more useful is this: the size of the instance is computed by adding up the number of digits of each number (with maybe a small increase added for some spaces to separate the numbers). This is a good measure of the space it takes to specify the problem instance. It's also actually a reasonable measure of size. For instance, suppose there are two numbers n_1 (having d_1 digits) and n_2 (having d_2 digits. The time it takes to add those two numbers is $O\big(\max\{d_1, d_2\}\big)$, and the time it takes to multiply them is $O(d_1 d_2)$. One could, of course, use n_1 and n_2 instead of d_1 and d_2, but then we would have to introduce logarithms into the bound estimates—and the estimates really wouldn't indicate the realistic costs of the computations.

So from now on, we assume that we have a reasonable way of specifying the size of any instance of a problem. Once we have that, we can meaningfully define some crucially important **classes** of problems. Here is the first one:

Definition *The class P is the class of problems for which there is a number k and an algorithm which solves the problem and whose running time is $O(n^k)$ where n is the size of the instance of the problem.*

This is called the class of *polynomial-time* problems. All the problems we have seen in this course so far are in P, almost always with a very small exponent.

1.3 Polynomial reducibility

We are familiar with the idea of solving a problem by reducing it to another problem. This is what software engineers do all the time when they write subroutines, of course. We have also seen it in this course. Remember that we were able to prove that the average-case running time of building a binary search tree was $\Theta(n \log n)$ by showing that it was really the same algorithm as QUICKSORT.

In effect, we *reduced* the BUILD-BINARY-SEARCH-TREE algorithm to the QUICKSORT algorithm, and also reduced the QUICKSORT algorithm to the BUILD-BINARY-SEARCH-TREE algorithm—that is, there was a reduction in both directions, and it was so trivial (once you looked at it correctly) that we could see that the two algorithms really took the same amount of time (well, up to a constant multiple, say).

Of course this doesn't happen often. Most algorithms are not simple reinterpretations of other algorithms. But there are other, more general kinds of reductions, that are weaker, and only go in one direction, but are still quite useful. In fact, the notion of reduction is central to theoretical

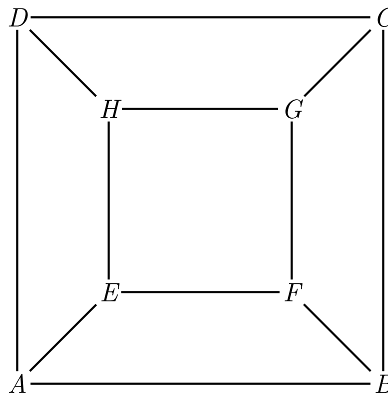
computer science, and there are many different kinds of reductions. The one we are concerned with here is called *polynomial reduction*.

There is a difference, however, between the kind of reductions we are going to consider here and the reductions we just talked about. Those reductions were reductions from one algorithm to another. The reductions that are important to us here are from one *problem* to another. That is, a reduction will tell us something to the effect that if we can solve problem A efficiently (by some algorithm), then we can use that solution to solve problem B efficiently.

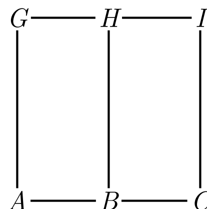
It turns out that to set this up well, it is best to consider a specific class of problems called *decision problems*. A decision problem is simply a problem for which the answer is "yes" or "no". Some problems are naturally decision problems. For instance, there is the HAMILTONIAN CYCLE problem: given an undirected graph $G = (V, E)$, is there a cycle that contains every vertex in V exactly once? We say that an *instance* of the Hamiltonian cycle problem is an undirected graph G, together with the question, "Does G have a Hamiltonian cycle?"

1.1 Exercise *For each of the following undirected graphs, either show that there is a Hamiltonian cycle, or prove that none can exist.*

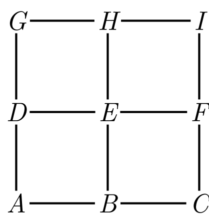
1. *The cube. Equivalently, the following graph:*



2. *The following graph:*



3. *The following graph:*



Hint: it's a pretty obvious point, but I suggest you prove the following lemma, which we'll also be using later:

1.2 Lemma *If G is a graph that has a Hamiltonian cycle C, then every vertex of G is an endpoint of exactly 2 edges in C.*

A decision problem thus consists of three parts: the name of the problem, a description of a general instance of that problem, and an associated question. In the problem we just considered, we can write this as follows:

Problem name: HAMILTONIAN CYCLE

Instance: A graph G, which may be directed or undirected. (This can really be thought of as splitting into two distinct problems: DIRECTED HAMILTONIAN CYCLE, and UNDIRECTED HAMILTONIAN CYCLE.)

Question: Is there a Hamiltonian cycle in G; that is, a cycle visiting each vertex exactly once?

Other problems, such as optimization problems, are not naturally decision problems. But they can be recast as decision problems. For instance, we say that an *independent set* in an undirected graph $G = (V, E)$ is a subset V_1 of the vertices V such that no two vertices in V_1 are joined by an edge in E. The "maximal independent set" problem is, given a graph G, to find the largest independent set in G. This is not a decision problem. However, we could change it into one: "Given a positive integer k, is there an independent set V_1 in the graph of size k? We say that the pair (G, k) is an *instance* of the independent set problem. (Implicitly of course, it comes with the question, "Does G have an independent set of size k?") That is, we define the problem like this:

Problem name: INDEPENDENT SET

Instance: An undirected graph G and a positive integer k.

Question: Is there an independent set in G of size k? That is, are there k vertices in G no two of which are joined by an edge?

So now we are ready to define what we mean by a polynomial-time reduction.

Suppose we have two decision problems (like HAMILTONIAN CYCLE and INDEPENDENT SET, for example). Let us call these decision problems A and B.

Definition *If A and B are two decision problems and there is a function f having the following properties:*

- $f : A \to B$. *That is* f *maps instances of* A *into instances of* B.
- f *is implemented by an algorithm that runs in polynomial time. That is, if* a *is an instance of the problem* A*, then the time to compute* $f(a)$ *is* $O(|a|^k)$ *for some* k*, where* $|a|$ *is the size of the instance* a.
- *For each* $a \in A$*,* a *has the same answer ("yes" or "no") as* $f(a)$.

then we say that A *is* polynomial-time reducible *to* B*, and we write*

$$A \leq_P B$$

Remarks *1. Note that we are* not *saying that either* A *or* B *are in* P*—we are not asserting that either has a polynomial-time algorithm that solves it. We* are *saying that there is a polynomial-time* reduction *from instances of* A *to instances of* B.

2. Also note that it is not necessary that every instance of B *"come from " an instance of* A *in this manner. It's just necessary that every instance of* A *"maps to" or "corresponds to" an instance of* B *in a way that can be computed in time bounded by a polynomial function of the size of the instance of* A.

There are some important properties we can draw from this:

- If $A \leq_P B$, then not only is $f(a)$ computable from a in polynomial time, but in fact $|f(a)|$ is a polynomially bounded function of $|a|$. That is $|f(a)| = O(|a|^k)$ for some k. This is because for some k, f only runs for $O(|a|^k)$ time and therefore cannot output an encoding for $f(a)$ that is longer than that.
- If $A \leq_P B$, and if B is a problem in P, then A is also. The reason is that we can take any instance a of A, transform it in polynomial time by f to a problem $f(a)$ in B whose size is no more than some fixed power of $|a|$, and then solve that problem by a polynomially bounded algorithm (since B is in P).

  To be more precise, suppose $|f(a)| = O(|a|^p)$, and suppose the algorithm for solving problem instances of B is $O(|b|^q)$. Then the cost of solving $f(a)$ is $O\Big(\big(|a|^p\big)^q\Big) = O(|a|^{pq})$. And since $f(a)$ has the same answer ("yes" or "no") as a does, solving $f(a)$ solves a. Therefore the polynomially bounded algorithm for B yields a polynomially bounded algorithm for A.

Well, that's very nice, but it turns out that the really useful way to use polynomial reducibility is in the other direction: if $A \leq_P B$ and A is in some sense hard to solve, then B must be also. For instance, if we knew that there was no polynomial-time algorithm for A, then we would also know that there could be no polynomial-time algorithm for B—we have just seen that such an algorithm for B would immediately yield one for A as well[2].

There is one other fact that we should note at this point:

---

[2] You have to be careful about the wording of this: If $A \leq_P B$, then

- A is *reducible* to B, and
- B is "harder" (or "at least as hard") as A.

The trouble is that the word "reducible" sounds like it ought to mean that B is in some sense "less" than A. But it really is "more", in the sense that B is "harder" than A. What we're really expressing is that A can be "reduced" to B in the sense that A can be solved by using B (and that getting B from A can be done in polynomial time).

1.3 Exercise *Prove carefully that if*

$$A \leq_P B$$
$$B \leq_P C$$

then

$$A \leq_P C$$

(This is not hard, and the proof is quite short. I just want you to be very careful and clear about it.)

1.4 The class NP

There is a very large class of problems for which no polynomial-time algorithm has been found, but which have a curious property: although they cannot (at least at the present time) be solved in polynomial time, they can be *checked* in polynomial time. For instance, no polynomial-time algorithm is known for the HAMILTONIAN CYCLE problem. Nevertheless, if someone gave you what they said was a Hamiltonian cycle[3] for a graph G, it would be easy for you to check that it was indeed a Hamiltonian cycle (or wasn't): you would just check

- that the list included all the vertices once and none twice, and
- that between each two consecutive vertices in the list (and between the first and the last) there was an edge in the graph.

This can obviously be done in linear time[4]

The class NP of problems is the the class of problems that can be checked in polynomial time.

Certainly P $\subseteq$ NP[5]. So at least some problems in NP are quite simple and easy to solve. But, as we will see, there are many that are not. And in fact, there are some problems that are as hard as any problems in NP.

Definition *A problem A is* NP-hard *iff for any problem B in NP,*

$$B \leq_P A$$

Definition *A problem A is* NP-complete *iff*

- *A is in NP, and*

---

[3] They would give it to you as a list of vertices.

[4] There is a subtle point here. It's not actually necessary that the information we give the algorithm is a solution. If the "checking" algorithm can prove, based on this information, that the instance G has a Hamiltonian cycle, that is all we really care about. For this reason, the actual technical name for this additional information is not a "solution", but a "certificate", or a "witness". (The two terms mean the same thing.) Usually a certificate is in fact a solution, but it doesn't have to be. This subtlety comes up in footnote 5 below.

[5] OK, maybe it's not so clear. Here's the reason: Remember that we are really concerned with decision problems. That is, we are not necessarily trying to produce an actual solution—all we have to do is say whether a solution exists. Now suppose that A is in P. For any instance a of A, if we are given a proposed solution (actually a certificate, as explained in the preceding footnote) s, we can "check" s by simply ignoring it and using our polynomial-time algorithm for A. This will return either "yes" or "no", which is all we really cared about anyway.

- *A is NP-hard.*

1.4 Exercise *Prove that all problems that are NP-complete are polynomially equivalent, in the sense that if A and B are NP-complete, then*

$$A \leq_P B$$
$$B \leq_P A$$

1.5 Exercise *Prove the following:*

In order to show that a problem A is NP-complete, it is enough to show that

- *A is in NP, and*
- *for some problem B that is NP-complete,*

$$B \leq_P A$$

(This is very simple: You use the fact that if C is any problem in NP, we must then have

$$C \leq_P B \leq_P A$$

which shows that A is in fact NP-complete.)

Without going further down this path, we are now going to give some examples of problems that are NP-complete, so you can get a feel for the kind of complexity that is typical of this class of problems.

2 Our first NP-complete problem

We are going to start with a problem that is known to be NP-complete. In fact, it is historically the first problem that was proved to be NP-complete. (It would be more accurate to say that it is one of the first such problems, because several people actually came to this at about the same time.) We won't prove that it actually is NP-complete right now—we will do this later. But we do mention the following: it is (as you will see) clearly in NP, and all the very best algorithms experts in the world have tried to find a polynomial-time algorithm for it, and they all failed. So at the very least, it is reasonable to assume that it is NP-complete. And as we will ultimately show, in fact it is.

The problem is known as SAT. (The text calls it CNF-SAT, which is probably more accurate.) It is a problem in mathematical logic, which sounds very abstract and far removed from practical concerns. But in fact it is closely related to problems in chip design, as the book hints at, and many other problems as well[6]. Here is the problem:

We have a set of Boolean variables. Let us call them $\{v_1, v_2, \ldots, v_n\}$. So each variable can take on either the value True or False. We make expressions (which we call *Boolean expressions*) using

[6] It is also general enough that—as we will see—many other problems can be easily expressed in terms of it. (More technically, and usefully, they can be *reduced* to it.) And there have been many special classes of instances of this problem that *can* be solved reasonably efficiently, and this is actually of great practical importance.

- these variables,
- and three operators:

$\vee$	this means "or", in the usual programming language sense
$\wedge$	this means "and", in the usual programming language sense
$\bar{v}$	this means "not v", in the usual programming language sense

- and parentheses, which we use in the usual way

It's not important to know this, but an expression such as $a \vee b$ is called a *disjunction*, and an expression of the form $a \wedge b$ is called a *conjunction*

We say that a *literal* is a very simple expression which is either v or $\bar{v}$ for some variable v.

Definition *A Boolean expression e is in* conjunctive normal form *if it is of the following form:*

$$e = c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

where each c_k is a clause*, which by the same definition is of the form*

$$c_k = (z_1^{(k)} \vee z_2^{(k)} \vee \ldots z_{n_k}^{(k)})$$

where each $z_j^{(k)}$ is a literal.

For instance, the expression

$$e = (v_1 \vee \bar{v}_2 \vee \bar{v}_3 \vee v_4) \wedge (v_1 \vee v_2 \vee \bar{v}_5) \wedge (v_3 \vee v_4 \vee v_5) \wedge (v_2 \vee v_4 \vee \bar{v}_5)$$

is an expression in conjunctive normal form. We say such an expression is *satisfiable* iff there is an assignment of True and False values to each of the variables v_j which makes the expression True. In this example, for instance, if we set $v_1 = v_4 = True$, then e will be True, regardless of the values of the other variables. So e is satisfiable.

> Warning: it is usually the case that the different literals in a clause come from different variables. But this is not guaranteed to be the case. For instance, $(v_1 \vee \bar{v}_1)$ is a legal clause.

The problem SAT is, given an expression in conjunctive normal form, to determine if it is satisfiable. That is, we write

> **Problem name:** SAT
>
> **Instance:** A Boolean expression e in conjunctive normal form.
>
> **Question:** Is e satisfiable?

This is a remarkably difficult problem. In fact, to my knowledge, there is no way to definitively solve it in general other than by exhaustive search, which certainly is a terrible way to proceed[7]. On the other hand, it is clearly in NP—if someone tells you a solution, you can check that solution in linear time.

We're not going to prove right now that this problem is actually NP-complete. But we are going to assume that this is the case, and we will derive some remarkable consequences from that fact. And eventually, we will actually prove that it is NP-complete.

2.1 Exercise *One of the really strange aspects of this field is that seemingly small changes to a problem can drastically alter its complexity. For instance, suppose we exchange the roles of "and" and "or": We say that an expression is in* disjunctive normal form *if it is of the form*

$$e = c_1 \vee c_2 \vee \cdots \vee c_m$$

where each c_k is a clause*, which by the same definition is of the form*

$$c_k = (z_1^{(k)} \wedge z_2^{(k)} \wedge \ldots z_{n_k}^{(k)})$$

where each $z_j^{(k)}$ is a literal.

Prove that the problem of satisfiability for expressions in disjunctive normal form is in P. (That is, prove that there is an algorithm which takes as input an expression e in disjunctive normal form and determines whether or not that expression is satisfiable in polynomial time. For simplicity, we can take $|e|$ to be the number of symbols in e.)

3 The basic toolkit of NP-complete problems

Now we are going to look at a number of problems, all of which we will be able to show are NP-complete. (We will show this based on the assumption which have made, and will later prove, that SAT is NP-complete.)

3.1 3-SAT

The first problem is 3-SAT. 3-SAT is a restricted form of SAT in which

1. all clauses have exactly 3 literals in them, and
2. those 3 literals are distinct.

For instance, here is an example of a 3-SAT expression:

$$(v_1 \vee \overline{v_1} \vee \overline{v_2}) \wedge (v_3 \vee v_2 \vee v_4) \wedge (\overline{v_1} \vee \overline{v_3} \vee \overline{v_4})$$

You might think that this simplifies the problem considerably, but in fact, it doesn't. We'll show that 3-SAT is just as hard as SAT, and is therefore NP-complete.

[7] As already mentioned, people have recently found algorithms that solve large numbers of instances (i.e., "special cases") of this problem reasonably efficiently, and this is significant. But the general problem still eludes us, and as we'll see below, this may be inevitable.

3.1 Theorem *3-SAT is NP-complete.*

PROOF.

1. **3-SAT is in NP.** This is straightforward: We can check an assignment to the variables of a 3-SAT expression by substituting them in each clause and verifying that each clause evaluates to True. This is certainly an $O(n)$ operation (where n is the number of literals in the whole expression).

2. **3-SAT is NP-hard.** We prove this by showing that SAT $\leq_P$ 3-SAT. We start with a SAT formula. This is just a formula in conjunctive normal form, but with no restriction on the number of variables in each clause. So it looks like this:

$$c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

where each c_k is a clause of the form

$$c_k = (z_1^{(k)} \vee z_2^{(k)} \vee \ldots z_{n_k}^{(k)})$$

where each $z_j^{(k)}$ is a literal—that is, it is either x or $\bar{x}$, where x is some variable.

We have to show how to turn this into an equivalent[8] 3-SAT expression—that is, an equivalent expression in which all the clauses have exactly 3 distinct literals, and such that the the algorithm that does this runs in polynomial time (in the size of the original expression). So as a consequence we also know that the size of the final expression will be polynomially bounded in terms of the size of the original expression.

Here's how we do it: We consider each clause separately. We replace each clause c by a set of clauses $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ (where n may be different for different original clauses c) in such a way that

- Each of the new clauses $\{\gamma_k : 1 \leq k \leq n\}$ will have exactly 3 distinct literals in it.
- The variables used in the new clauses $\{\gamma_k : 1 \leq k \leq n\}$ will be the variables in the original clause c together with possibly some new variables. But the new variables will occur only in the new clauses $\{\gamma_k : 1 \leq k \leq n\}$ that come from the original clause c, and not in any other new clauses that come from any other original clause in the expression.
- c will be True iff each clause in $\{\gamma_k : 1 \leq k \leq n\}$ is true
  - for *some* values given to the new variables, and
  - with the *same* values given to the variables of c_k

(This is a little bit tricky; I myself have been confused by it. The point is this: there must be at least one set of values given to the new variables that contributes to a satisfying assignment, and whether there is one set of such values or more than one set of such values, the values given to the original variables of c_k must be the same as those given in the original clause. So if the original clause c_k is satisfiable, then the same values of its variables work for each clause in $\{\gamma_k : 1 \leq k \leq n\}$ with some values for the additional variables; and in the other direction, if each clause in $\{\gamma_k : 1 \leq k \leq n\}$ is satisfiable with some assignment of truth values to all the variables, then the values of the original variables in that set of clauses must also satisfy the original clause c_k.)

There are four cases to consider:

[8] By "equivalent", we mean that the original SAT expression is satisfiable iff the 3-SAT expression is satisfiable.

$|c| = 1$. That is, $c = z_1$, where z_1 is some literal. We introduce two new variables y_1 and y_2, and we set

$$\gamma_1 = (z_1 \vee y_1 \vee y_2)$$
$$\gamma_2 = (z_1 \vee y_1 \vee \overline{y_2})$$
$$\gamma_3 = (z_1 \vee \overline{y_1} \vee y_2)$$
$$\gamma_4 = (z_1 \vee \overline{y_1} \vee \overline{y_2})$$

It should be easy to see that c is True iff $(\gamma_1 \wedge \gamma_2 \wedge \gamma_3 \wedge \gamma_4)$ is True with the same value of z_1 and with *any* values of y_1 and y_2.

$|c| = 2$. That is, $c = (z_1 \vee z_2)$. We introduce one new variable y_1, and we set

$$\gamma_1 = (z_1 \vee z_2 \vee y_1)$$
$$\gamma_2 = (z_1 \vee z_2 \vee \overline{y_1})$$

Again, we can see that c is True iff $(\gamma_1 \wedge \gamma_2)$ is true with the same values of z_1 and z_2 and *any* value of y_1.

$|c| = 3$. That is, $c_k = (z_1 \vee z_2 \vee z_3)$. In this case, there is nothing to do: we just set $\gamma_1 = c$.

$|c| \geq 4$. That is, we have

$$c = z_1 \vee z_2 \vee \dots z_n$$

where $n = |c| \geq 4$. We introduce new variables $y_1, \dots, y_{n-3}$ and set

$$\gamma_1 = (z_1 \vee z_2 \vee y_1)$$
$$\gamma_2 = (\overline{y_1} \vee z_3 \vee y_2)$$
$$\gamma_3 = (\overline{y_2} \vee z_4 \vee y_3)$$
$$\dots$$
$$\gamma_{n-2} = (\overline{y_{n-3}} \vee z_{n-1} \vee z_n)$$

In this case, not every assignment of truth values to the variables $\{y_1, \dots, y_{n-3}\}$ will work in general, but there will always be at least one assignment that does work. □

3.2 Exercise *Prove that this reduction does what it is supposed to do; that is, that it really does show that*

$$\text{SAT} \leq_P \text{3-SAT}$$

Note that to do this, you have to show that we have really satisfied all three conditions of the definition of a polynomial reduction (which is on page 4). The first condition is obvious, I think. The second is easy, but you do have to justify it. As for the third condition, there are four cases to consider, just as in the proof. The first three are easy, but you do have to consider them explicitly anyway.

3.2 Vertex cover

A *vertex cover* of an undirected graph $G = (V, E)$ is a subset of vertices $V_1 \subseteq V$ such that every edge $e \in E$ is incident on (at least) one element of V_1. So for instance, in Figure 1, the black vertices constitute a vertex cover[9].

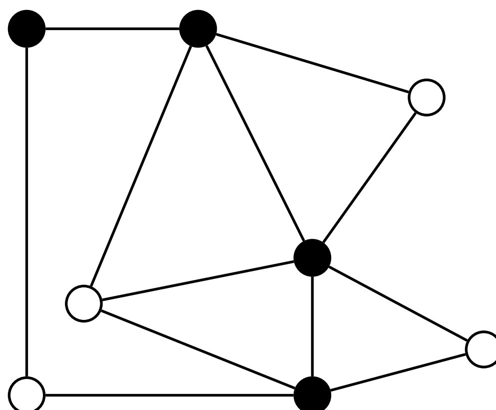


Figure 1: The black vertices form a vertex cover of the graph, since each edge is incident on at least one black vertex.

Given an undirected graph, we would like to find the smallest vertex cover. The corresponding decision problem is thus "Is there a vertex cover of size k?". That is,

Problem name: VERTEX COVER

Instance: An undirected graph G together with a positive integer k.

Question: Does G have a vertex cover of size k?

We abbreviate *vertex cover* by VC. An instance of VC is a pair (G, k) where G is a graph, and the question is "Is there a vertex cover of G of size k?"

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

PROOF.

1. **VC is in NP.** Clearly, given a set $V_1 \subseteq V$, we can check that the size of V_1 is k and that each edge $e \in E$ is incident on a vertex in V_1 in linear time. (Well, $O(E + V)$, say.)

2. **VC is NP-hard.** We prove this by showing that 3-SAT $\leq_P$ VC.

   Let us start with a 3-SAT instance with N variables and C clauses. We will construct a graph with $2N + 3C$ vertices such that

[9] The way to remember this is to say that "the vertices cover the edges."

- The construction can be done in "polynomial time".
- The original 3-SAT instance is satisfiable iff the graph we construct has a vertex cover with $N + 2C$ vertices.

If we can do this, we will be done.

We will use an example to show how the graph is constructed, but our construction will be described in perfectly general terms, so it applies to any 3-SAT instance.

The graph we will construct consists of three parts. The first part consists of pairs of vertices, one pair for each variable in the instance. Each pair is labeled with the variable and its negation, and the pair is connected by an edge, as in Figure 2. This part of the graph consists of the *truth-setting components*.

Figure 2: Stage 1 of the construction of the graph corresponding to the 3-SAT instance

$$(v_1 \vee \bar{v}_3 \vee \bar{v}_4) \wedge (\bar{v}_1 \vee v_2 \vee \bar{v}_4)$$

The second part of the graph consists of a triangle of nodes for each clause in the 3-SAT instance. The nodes of the triangle are labeled by the literals in the clause. Figure 3 shows how this is done in our running example. This second part of the graph consists of the *satisfaction-testing components*.

Notice that so far, we really have not used anything very specific about the 3-SAT instance. Any 3-SAT instance consisting of two clauses with four variables would yield a construction starting out exactly like this.

Finally, we add edges between the truth-setting components on top and the satisfaction-testing components on the bottom. These edges encode the literal values in the graph. We attach every node on the bottom to its node of the same name on the top. Figure 4 shows how this is done in our running example. We can refer to these edges as *cross edges*.

Now let us look at what a vertex cover of this graph must be like.

- First, since every one of the truth-setting edges (on the top) must be covered, a vertex cover must include at least one of every pair of truth-setting vertices on the top. Thus, the vertex cover must include at least N vertices of this type.
- In addition, any vertex cover must include at least 2 out of the three vertices of each satisfaction-testing triangle on the bottom, because the edges of those triangles can't be covered in any other way. Thus, the vertex cover must include at least $2C$ vertices of that type.

So any vertex cover of the graph must include at least $N + 2C$ vertices.

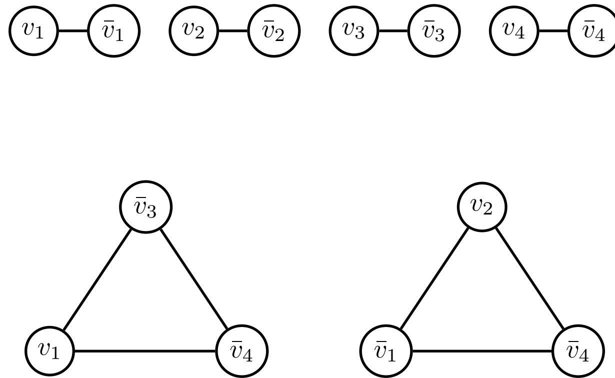


Figure 3: Stage 2 of the construction of the graph corresponding to the 3-SAT instance $(v_1 \vee \bar{v}_3 \vee \bar{v}_4) \wedge (\bar{v}_1 \vee v_2 \vee \bar{v}_4)$

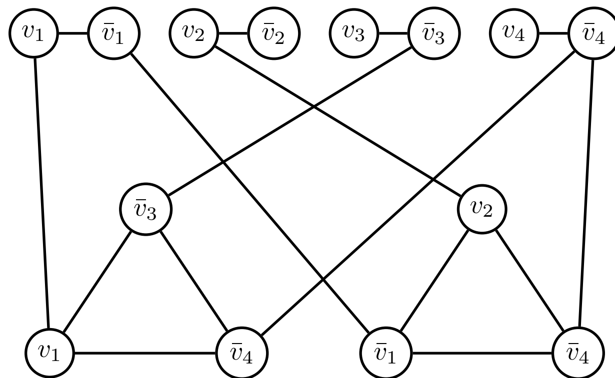


Figure 4: The graph corresponding to the 3-SAT instance $(v_1 \vee \bar{v}_3 \vee \bar{v}_4) \wedge (\bar{v}_1 \vee v_2 \vee \bar{v}_4)$

And if it does include N vertices "on the top" and $2C$ vertices "on the bottom", then all the top edges and all the triangle edges will be covered. The only issue, then, is whether the cross edges are covered.

So now we are ready to prove that "the solution to a is the same as the solution to $f(a)$". We will do this in two steps: "$a \implies f(a)$" and $f(a) \implies a$". We'll write each step as a lemma (contained entirely in the proof of the main theorem).

3.4 Lemma *If the original 3-SAT instance is satisfiable, then the derived graph has a vertex cover of size* $N + 2C$

Proof. We suppose the original 3-SAT instance is satisfiable. We construct our vertex cover as follows:

- For each pair of truth-setting vertices, color in the one that is True.
- Since each "triangle" must have at least one vertex corresponding to a literal that is True, the cross edge coming to that vertex will already be covered by what have just done. So pick the other two vertices for the vertex cover. That will ensure that not only is the triangle covered but that all the cross edges to that triangle are covered.

Thus, a satisfying set of truth values for the N variables in the original 3-SAT instance corresponds to a vertex cover of size $N + 2C$ of the derived graph □

And now we go in the other direction:

3.5 Lemma *If the derived graph has a vertex cover of size* $N + 2C$*, then the original 3-SAT instance is satisfiable.*

Proof. Now we suppose we have a vertex cover of the size $N + 2C$ of the derived graph. We already know that N of the "top" vertices are part of the cover and $2C$ of the "bottom" ones are also—two in every triangle. We let the N vertices of the cover on the top specify the truth values of each of the N variables.

Now this means that the vertex in each triangle that is not part of the cover must be true. The reason for this is that it is one endpoint of a cross edge, and since we have a vertex cover, the other endpoint of that cross edge must be part of the cover, and so that literal is True.

Thus, at least one literal in each clause is True, and so the original 3-SAT instance is satisfiable. □

Finally, we note that the construction of the derived graph is clearly a polynomial-time construction. And that concludes the proof of the theorem. □

3.3 CLIQUE and INDEPENDENT SET

A clique in an undirected graph is a subset of vertices such that each pair of the vertices is joined by an edge in the graph. (Equivalently, a clique is a complete subgraph.)

Given an undirected graph, we would like to find the largest clique. The corresponding decision problem is "Given a graph G and a number k, does G contain a clique of size k?". So an instance of CLIQUE is again a pair (G, k) where G is a graph, k is a number, and the associated question is as above.

Problem name: CLIQUE

Instance: An undirected graph G together with a positive integer k.

Question: Does G have a clique of size k?

An independent set in a graph is a set of vertices such that no two vertices in the set are joined by an edge. The INDEPENDENT SET problem is, given a graph, to find the largest independent set. The equivalent decision problem is "Does G contain an independent set of size k?". Again, an instance of INDEPENDENT SET is a pair (G, k) where G is an undirected graph, k is a positive integer, and the associated question is as above.

Problem name: INDEPENDENT SET

Instance: An undirected graph G together with a positive integer k.

Question: Does G have an independent set of size k?

It turns out that these two problems are easily shown to be equivalent to the vertex cover problem. To see this, we need one further definition:

Definition *The* complement G^c *of an undirected graph* G *is a graph whose vertices are the same as the vertices of* G*, and such that for any pair of vertices* u *and* v*,* (u, v) *is an edge of* G^c *iff it is not an edge of* G*.*

3.6 Exercise *Prove that the following are equivalent:*

1. V_1 *is a vertex cover of* G*.*
2. $V \setminus V_1$ *is an independent set in* G*.*

and, continuing, prove that the following are equivalent:

1. V_2 *is an independent set in* G*.*
2. V_2 *is a clique in* G^c*.*

Thus these three problems are equivalent (and clearly *polynomially equivalent*), and so since the vertex cover problem is NP-complete, so are the other two problems.

3.7 Exercise *Show that INDEPENDENT SET when restricted to trees is in P. Do this by constructing a dynamic programming solution and giving a nice bound for the running time.*

3.4 Integer linear programming

An instance of the INTEGER LINEAR PROGRAMMING problem consists of a set $\{v_1, v_2, \ldots, v_n\}$ of integer variables, a set of linear inequalities (with integer coefficients) over these variables, a

function $f(v_1, v_2, \ldots, v_n)$ to maximize, and an integer B. The function f is of the form

$$f(v_1, v_2, \ldots, v_n) = \sum_{i=1}^{n} c_i v_i$$

where each coefficient c_i is an integer. Thus, we can write

The yes-or-no question for the instance is, "Does there exist an assignment of integers to the variables $\{v_i\}$ such that all the inequalities are true and $f(v_1, v_2, \ldots, v_n) \geq B$?"

Problem name: INTEGER LINEAR PROGRAMMING

Instance: A set $\{v_1, v_2, \ldots, v_n\}$ of integer variables, a set of linear inequalities (with integer coefficients) over these variables, a function $f(v_1, v_2, \ldots, v_n)$ to maximize, and an integer B. The function f is of the form

$$f(v_1, v_2, \ldots, v_n) = \sum_{i=1}^{n} c_i v_i$$

where each coefficient c_i is an integer.

Question: Does there exist an assignment of integers to the variables $\{v_i\}$ such that all the inequalities are true and $f(v_1, v_2, \ldots, v_n) \geq B$?

For example, here is an instance:

variables: v_1, v_2

inequalities:

$$\begin{aligned} v_1 &\geq 1 \\ v_2 &\geq 0 \\ v_1 + v_2 &\leq 3 \end{aligned}$$

function: $f(v_1, v_2) = 2v_2$

bound: $B = 3$

A solution to this instance is

$$\begin{aligned} v_1 &= 1 \\ v_2 &= 2 \end{aligned}$$

On the other hand, here is an instance that has no solution:

variables: v_1, v_2

inequalities:

$$v_1 \geq 1$$
$$v_2 \geq 0$$
$$v_1 + v_2 \leq 3$$

function: $f(v_1, v_2) = 2v_2$

bound: $B = 5$

3.8 Theorem *Integer linear programming is NP-hard.*

Proof. We will show that SAT reduces to it:

$$\text{SAT} \leq_P \text{INTEGER LINEAR PROGRAMMING}$$

We start with some instance of SAT. This instance contains variables $\{v_1, v_2, \ldots, v_n\}$ and clauses. We will create a linear integer programming instance as follows:

There will be two variables for each variable v_i. We will name these variables V_i and $\bar{V}_i$. They will correspond to the literals v_i and $\bar{v}_i$. (But be careful—notice that in the integer programming instance they are separate variables, not one variable and its "negation". In fact, they are variables that take on integer values, not Boolean values.)

Next we introduce the inequalities. There are three classes of inequalities:

Class I. In order to make these variables model the original Boolean variables, we will introduce some inequalities as part of our integer programming instance:

$$0 \leq V_i \leq 1$$
$$0 \leq \bar{V}_i \leq 1$$

This really amounts to four inequalities, and it means that each V_i and $\bar{V}_i$ is either 0 (which we can think of as False) or 1 (which we can think of as True).

Class II. Further, we introduce two more inequalities:

$$1 \leq V_i + \bar{V}_i \leq 1$$

Of course this is just the equation $V_i + \bar{V}_i = 1$, but we needed to express it in terms of inequalities to make this be what we have said an integer linear programming instance is. This equation says exactly one of V_i and $\bar{V}_i$ is true.

Class III. Finally, we introduce inequalities that encode the clauses in the SAT problem. For each clause

$$z_1^{(k)} \vee z_2^{(k)} \vee \cdots \vee z_{n_k}^{(k)}$$

we create an inequality

$$W_1 + W_2 + \cdots + W_{n_k} \geq 1$$

where

$$W_j = \begin{cases} V_p & \text{if } z_j^{(k)} = v_p \\ \bar{V}_p & \text{if } z_j^{(k)} = \bar{v}_p \end{cases}$$

For instance, if we have a clause

$$v_1 \vee \bar{v}_{19} \vee \bar{v}_7 \vee \cdots \vee v_6$$

then we introduce the inequality

$$V_1 + \bar{V}_{19} + \bar{V}_7 + \cdots + V_6 \geq 1$$

Clearly (given the fact that each variable is constrained to be either 0 or 1), this inequality is satisfied iff at least one of the variables in it is 1, which corresponds to at least one of the literals in the clause being True.

As for the function f and the bound B, we don't need them at all. We can simply set $f(V_1, \bar{V}_1, \ldots, \bar{V}_n) = 0$ and $B = 0$.

Now having done this, we see immediately that

- The integer linear programming instance that we have constructed from the SAT instance has a solution iff the SAT instance is satisfiable.
- The construction of the integer programming instance from the SAT instance is a polynomial-time algorithm.

And that concludes the proof. □

There are a few things we need to say about this:

- We have not really shown that INTEGER LINEAR PROGRAMMING is NP-complete. We *have* shown that it is NP-hard, but it is not quite obvious that it is in NP, because the integers in the solution to the instance (not in the instance itself!) might in principle be so large that they couldn't even be written out in polynomial time.
- On the other hand, what we actually showed was that SAT could be reduced to a more restricted problem: 0-1 INTEGER LINEAR PROGRAMMING, in which each variable can take on only the values 0 and 1, and in which each coefficient is also 0 or 1. This problem is certainly in NP, and so it is NP-complete.

  This shows, in particular, that the reason that INTEGER LINEAR PROGRAMMING is hard has nothing to do with big coefficients or big ranges on variables—restricting all values to 0 and 1 still leaves the problem being NP-hard. So not only did this reduction prove that INTEGER LINEAR PROGRAMMING was NP-hard, it also gave us some insight into where the difficulty lies (and where it doesn't lie).
- And actually, although we did not prove this, it *has* been proved[10] that INTEGER LINEAR PROGRAMMING is in fact in NP, and so is NP-complete.

[10] See I. Borosh and L. B. Treybig, "Bounds on Positive Integral Solutions of Linear Diophantine Equations", Proceedings of the American Mathematical Society, Vol. 55, No. 2, March 1976 (299–304), and Christos H. Papadimitriou, "On the Complexity of Integer Programming", Journal of the Association for Computing Machinery, Vol. 28, No. 4, October 1981 (765–768).

3.5 SUBSET SUM

This problem is also called INTEGER PARTITION. An instance of the problem is a set S of integers and a "target" integer t. The question is, "Is there a subset of S whose sum is t?"

Problem name: SUBSET SUM

Instance: A set S of integers and a "target" integer t.

Question: Is there a subset of S whose sum is t?

For instance, if

$$S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$$

and $t = 3754$, then the answer is "yes", because

$$1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = t$$

3.9 Theorem *SUBSET SUM is NP-complete.*

PROOF.

1. **SUBSET SUM is in NP.** This is obvious: checking that a *particular* subset adds up to t can certainly be done in linear time[11].

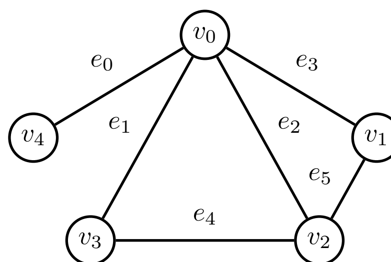
2. **SUBSET SUM is NP-hard.** We will prove this by reducing VERTEX COVER to SUBSET SUM:

   $$\text{VC} \leq_P \text{SUBSET SUM}$$

   Thus we need to start with a graph in which we are trying to find a vertex cover of size N, and turn this VC instance into an instance of SUBSET SUM.

   We take our graph, and we construct its *incidence matrix*:

	e_0	e_1	e_2	e_3	e_4	e_5
v_0	1	1	1	1	0	0
v_1	0	0	0	1	0	1
v_2	0	0	1	0	1	1
v_3	0	1	0	0	1	0
v_4	1	0	0	0	0	0



[11] And remember that by "linear time" in this context, we mean "linear in the number of digits of the numbers in the problem".

Note that in an incidence matrix there are exactly two 1's in each column. That will be a key point. We will call this matrix b, and we have for instance $b[2,1] = 0$ and $b[3,1] = 1$.

Now we are going to think of each row as a base-4 representation of an integer, only with the low-order digits on the *left* so that the row for v_2 corresponds to

$$4^2 + 4^4 + 4^5$$

In other words, the row corresponding to the vertex v_i corresponds to the number

$$\sum_{j=0}^{|E|-1} b[i,j] \cdot 4^j$$

Then we are going to extend the matrix by adding a new row for each edge, and we will put a 1 in the column that corresponds to that edge:

	e_0	e_1	e_2	e_3	e_4	e_5
v_0	1	1	1	1	0	0
v_1	0	0	0	1	0	1
v_2	0	0	1	0	1	1
v_3	0	1	0	0	1	0
v_4	1	0	0	0	0	0
e_0	1	0	0	0	0	0
e_1	0	1	0	0	0	0
e_2	0	0	1	0	0	0
e_3	0	0	0	1	0	0
e_4	0	0	0	0	1	0
e_5	0	0	0	0	0	1

Now each column has three 1's in it: two from vertex rows and one from an edge row. The top rows of this matrix are just the original matrix b.

Now corresponding to each vertex row, we construct the number (which is just the number we talked about above, but with a high-order term added).

$$V_i = 4^{|E|} + \sum_{j=0}^{|E|-1} b[i,j]4^j$$

We will call these the "vertex numbers".

And corresponding to each edge row, we construct the number (this time *without* the high-order term added)

$$E_k = 4^k$$

We will call these the "edge numbers".

Now the subset sum instance is this: the numbers in our set S are just the numbers V_i and E_k we just constructed. The target number is

$$t = N \cdot 4^{|E|} + 2 \cdot \sum_{j=0}^{|E|-1} 4^j$$

We will now show that the graph we started with has a vertex cover of size N iff the subset sum problem we have just constructed is solvable.

Now let us notice the following facts:

- If we add up any subset of numbers in S (even if we add up *all* the numbers in S, there will be no "carries" from one column to the next in the base-4 addition. The reason for this is that each column can contain at most three 1's, and it would take four 1's to produce a carry.
- It follows from this that for a sum of numbers in S to equal t it must contain exactly N vertex numbers, since that is how many vertex numbers we will need to get the high term $N \cdot 4^{|E|}$ in t.

So we are now ready to show that the VC instance is solvable iff the SUBSET SUM instance is solvable. As before, we will write this as two lemmas, completely contained in the body of this proof.

3.10 Lemma *If the VC instance is solvable, then the derived SUBSET SUM instance is solvable.*

PROOF. If we have a vertex cover of the graph with N vertices, and if we take the sum of the corresponding vertex numbers, we will certainly have a high-order term of $N \cdot 4^{|E|}$. As for the other terms, since each edge in the graph is "covered", we will have at least a contribution of $1 \cdot 4^j$ for each edge e_j. If we only have $1 \cdot 4^j$ and not $2 \cdot 4^j$, then we can add the edge number E_j. In this way we arrive at a solution to the SUBSET SUM problem. □

3.11 Lemma *If the derived SUBSET SUM instance is solvable, then the VC instance is solvable.*

PROOF. Take the vertex numbers in the solution of the SUBSET SUM instance. We know that there are exactly N of them. Further, we know that the rest of the numbers in the solution (which are the edge numbers) can only contribute at most a 1 in each remaining column. Therefore the vertex numbers have to contribute either 1 or 2 in each column. This means that each edge is covered by (i.e., incident on) either 1 or 2 vertices in the subset of vertices that corresponds to the vertex numbers in the solution to the derived SUBSET SUM instance. And that means that those vertices constitute a vertex cover of size N. □

Finally, as before, it is clear that this transformation of VC into SUBSET SUM is a polynomial-time algorithm. And that concludes the proof. □

3.12 Exercise

1. *Show that there is a dynamic programming solution to SUBSET SUM that solves the problem in time $O(nt)$.*
2. *Why does this not show that SUBSET SUM is in P?*

3.6 HAMILTONIAN CYCLE

A *Hamiltonian cycle* in a graph G is a simple cycle that visits each vertex. There are two variants of this problem, depending on whether we take the graph to be directed or undirected. In either case, no one has found a polynomial-time solution, but on the other hand, each problem is clearly checkable in polynomial time. So both problems are in NP. And in fact, as we will see, both problems are NP-complete.

The construction for this is pretty tricky, I think. And personally, I find the proof in our text really difficult to read. Part of the reason, I think, is that it is trying to prove too much at once. It is proving that UNDIRECTED HAMILTONIAN CYCLE is NP-complete. It turns out to be easier to prove this in two steps:

- Prove that DIRECTED HAMILTONIAN CYCLE is NP-complete by reducing some known NP-complete problem to it.
- Then prove that UNDIRECTED HAMILTONIAN CYCLE is NP-complete by reducing DIRECTED HAMILTONIAN CYCLE to it.

Historically, this seems to be the way this was originally proved, and it seems to me that the construction given in the text is in some way an amalgamation of the two constructions for those two separate proofs. And I think it's just easier to see each one separately. So that's what I'll do here.

3.13 Theorem *DIRECTED HAMILTONIAN CYCLE is NP-complete.*

PROOF.

1. **DIRECTED HAMILTONIAN CYCLE is in NP.** Clearly it's polynomial-time checkable.
2. **DIRECTED HAMILTONIAN CYCLE is NP-hard.** We will prove this by reducing 3-SAT to it[12]. That is, we will show that

$$\text{3-SAT} \leq_P \text{DIRECTED HAMILTONIAN CYCLE}$$

So we start with a 3-SAT instance that has n variables $\{v_1, v_2, \ldots, v_n\}$ and k clauses $\{c_1, c_2, \ldots, c_k\}$, where each clause is of the form $z_1 \vee z_2 \vee z_3$, each z_j being a literal. We will show how to take this 3-SAT instance and produce from it a graph $G = (V, E)$ such that

- The construction can be performed in polynomial time as a function of $n + k$.
- G has a Hamiltonian cycle iff the 3-SAT instance is satisfiable.

Here we go.

First, we may assume that each clause in our 3-SAT instance involves 3 distinct variables. For if a clause is of the form

$$v_1 \vee \bar{v}_1 \vee v_2$$

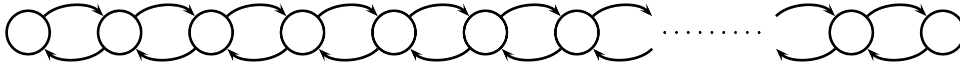
[12] Of course it is just a matter of preference *which* NP-complete problem we choose to reduce to DIRECTED HAMILTONIAN CYCLE. The original proof used VERTEX COVER. I'm following Kleinberg and Tardos (Algorithm Design) and Sipser (Introduction to the Theory of Computation, 2nd edition) here.

then it is automatically true, and we can just eliminate it from the instance. On the other hand, a clause such as

$$v_1 \vee v_1 \vee v_2$$

is really just $v_1 \vee v_2$, and we have already seen how to turn this in to a pair of clauses (with a new variable), each clause containing 3 literals. So let us assume that this has been done and that each clause of our 3-SAT instance contains literals corresponding to 3 different variables. (Of course, as usual, one variable can occur in many *different* clauses.)

For each variable v_i we create a set of vertices in G and hook them together in a "doubly linked list", like this:



The exact number of nodes you can think of as "as many as we need". It will turn out to be $3(k+1)$.

We take the list corresponding to each node and connect it to some auxiliary nodes to form a oval-like structure, and we then hook up these oval structures vertically, as in Figure 5. We also add k other nodes, each one corresponding to one of the clauses in the 3-SAT expression. In Figure 5 we see that to form a Hamiltonian cycle, each row will either be traversed left-to-right or right-to-left, and the choice for each row is independent of the choice for every other row. We will say that a traversal of row i left-to-right encodes the value True for the variable v_i, and a traversal of row i right-to-left encodes the value False for the variable v_i.

So there are 2^n possible Hamiltonian cycles of the graph in Figure 5, and these different cycles correspond exactly to the 2^n different ways of assigning either True or False to the n different variables $\{v_1, v_2, \ldots, v_n\}$.

Now what we are going to do is hook up the nodes $\{c_1, c_2, \ldots, c_k\}$ to the rest of the graph in such a way that the clause information is encoded.

We do this as follows: we divide each row (corresponding to each variable v_i as follows:

- An initial node (i.e., the left-most one).
- A "separator node".
- k sets of 3 nodes each. The jth set corresponds to the clause c_j. Actually, the first two nodes in the set correspond to c_j and the third node in each set is another "separator node". We will call the first two nodes in each set the "c_j group in row i".
- A final node (i.e., the right-most one).

Now each clause c_j contains three literals ($c_j = z_1^{(j)} \vee z_2^{(j)} \vee z_3^{(j)}$). For each of those literals, we add two edges involving c_j, as follows: Take one of the literals z. z corresponds to some variable v_i—that is, $z = v_i$ or $z = \bar{v}_i$. The two edges we insert will connect c_j with the two nodes in the c_j group in row i, as follows:

- If $z = v_i$, we insert an edge from

$$\text{the left node in the } c_j \text{ group} \to c_j$$

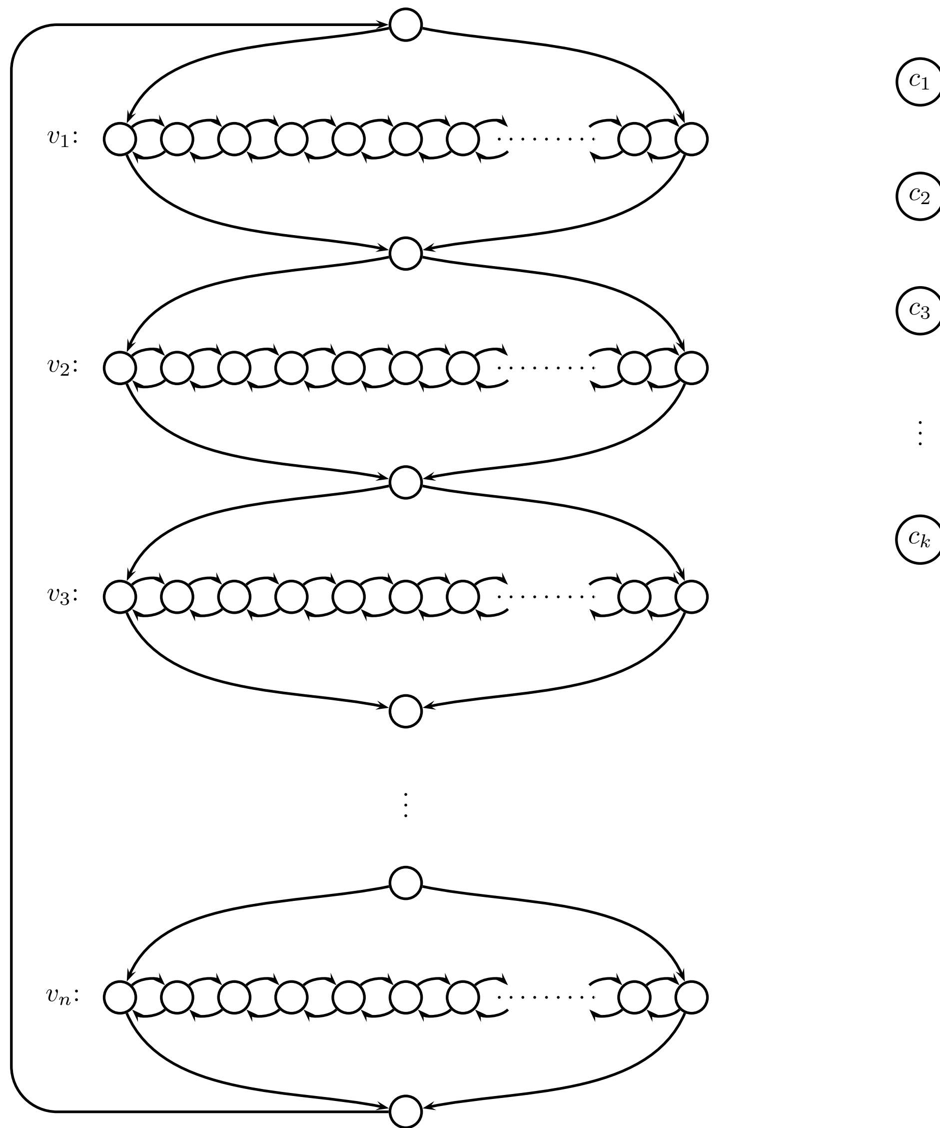


Figure 5: Outline of the directed graph produced by a 3-SAT instance. The nodes $\{c_1, c_2, \ldots, c_k\}$ are connected to the rest of the graph as well; we show that later.

and we insert an edge from

$$c_j \rightarrow \text{the right node in the } c_j \text{ group}$$

See Figure 6. The reason for doing this is that if v_i has the value True, then (since row i will be traversed left-to-right), we can use these two edges to make a side trip to c_j, thus including c_j in the cycle.

- If $z = \bar{v}_i$, we do things "the other way": we insert an edge from

  $$\text{the right node in the } c_j \text{ group} \to c_j$$

  and we insert an edge from

  $$c_j \to \text{the left node in the } c_j \text{ group}$$

  See Figure 7. The reason for doing this is that if v_i has the value False (so $\bar{v}_i$ is True), then (since row i will be traversed right-to-left) we can use these two edges to make a side trip to c_j, thus including c_j in the cycle.

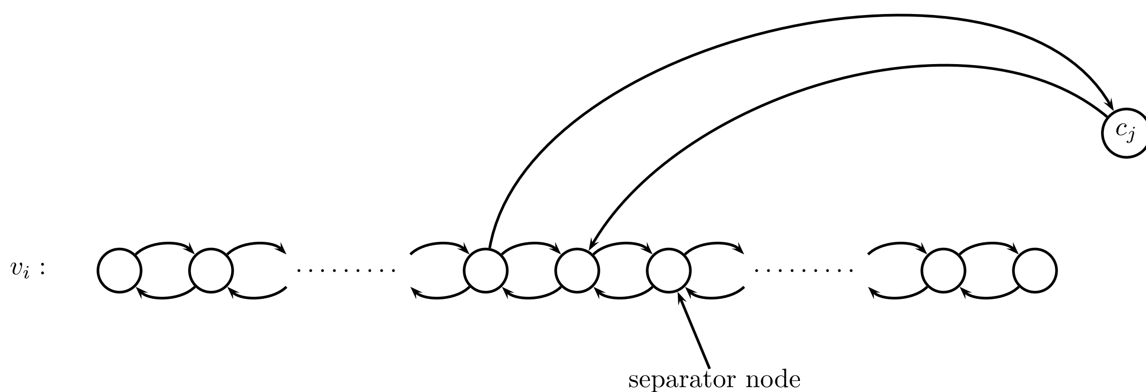


Figure 6: Two edges added when clause c_j contains v_i.

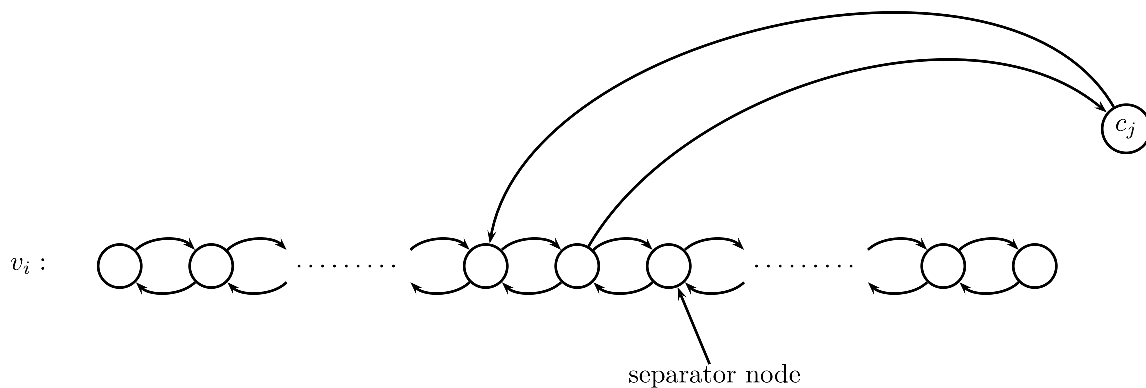


Figure 7: Two edges added when clause c_j contains $\bar{v}_i$.

Thus, each clause c_j has three pairs of edges that are inserted for it. And notice that these edges will never "step on each other": the edges from clause c_j will all attach to nodes in what

you can think of as "column j" of the main part of the graph. So the edges from two different clauses will never coincide. And further, since we have assumed that no clause contains the same variable twice, each of the three pairs of edges introduced for each clause goes to a different row, so they can't coincide either.

That is the construction of G that we promised. It is clear that if the original 3-SAT expression is satisfiable, then G has a Hamiltonian cycle: just traverse each edge in the appropriate way (i.e., left-to-right if v_i is True and right-to-left if v_i is False). For each clause c_j at least one of the literals in c_j will be True. For the row corresponding to the variable v_i in that literal, a side trip can be made to c_j since the two edges to and from c_j in that row were set up that way. Thus each c_j can be included in the cycle, and so we really do have a Hamiltonian cycle for G.

It remains to show the opposite implication: Suppose G has a Hamiltonian cycle. We must show that the original 3-SAT instance is satisfiable. Now this is immediately true if we know that each c_j is reached by a path to and from the same row. For then the variable in that row corresponds to a True literal in c_j, and so each c_j is satisfied. So all we have to prove is that if G has a Hamiltonian cycle, then each c_j is reached by a path to and from the same row.

Suppose it were not. Suppose we had something like the situation in Figure 8. Let us suppose that in Figure 8 that a_1 is a node in some row that is reached from the left, and that the edge from a_1 to c_j is not followed by an edge (in the path) from c_j to a_2.

We know that either a_2 or a_3 must be a separator nodes. Let us consider these two possibilities separately:

Case I: a_2 is a separator node. A separator node must be attached to the nodes on either side of it. But a_2 cannot be attached to a_1 by an edge in the Hamiltonian cycle, since a_1 already has two Hamiltonian cycle edges attached to it. So this is impossible.

Case II: a_3 is a separator node. In this case a_1 and a_2 must both correspond to the same clause (in this case, c_j). So a_2 must be attached either to c_j or to a_1 by an edge in the Hamiltonian cycle, and again neither one is possible, since both those nodes already have two Hamiltonian cycle edges attached to them.

Thus the situation we want to avoid is in fact impossible provided that a_1 is approached from the left. If it were approached from the right, then the "mirror image" of this argument would show that this case was also impossible.

Therefore we have in fact shown that a Hamiltonian cycle of G corresponds to an assignment of truth values to the variables $\{v_1, v_2, \ldots, v_n\}$ that satisfies the original 3-SAT instance.

Finally, we note that the construction of G was certainly a polynomial-time construction. And that concludes the proof. □

That was a pretty complex construction. Fortunately, the next one is actually pretty straightforward.

3.14 Theorem *UNDIRECTED HAMILTONIAN CYCLE is NP-complete.*

PROOF.

1. **UNDIRECTED HAMILTONIAN CYCLE is in NP.** Clearly it's polynomial-time checkable.

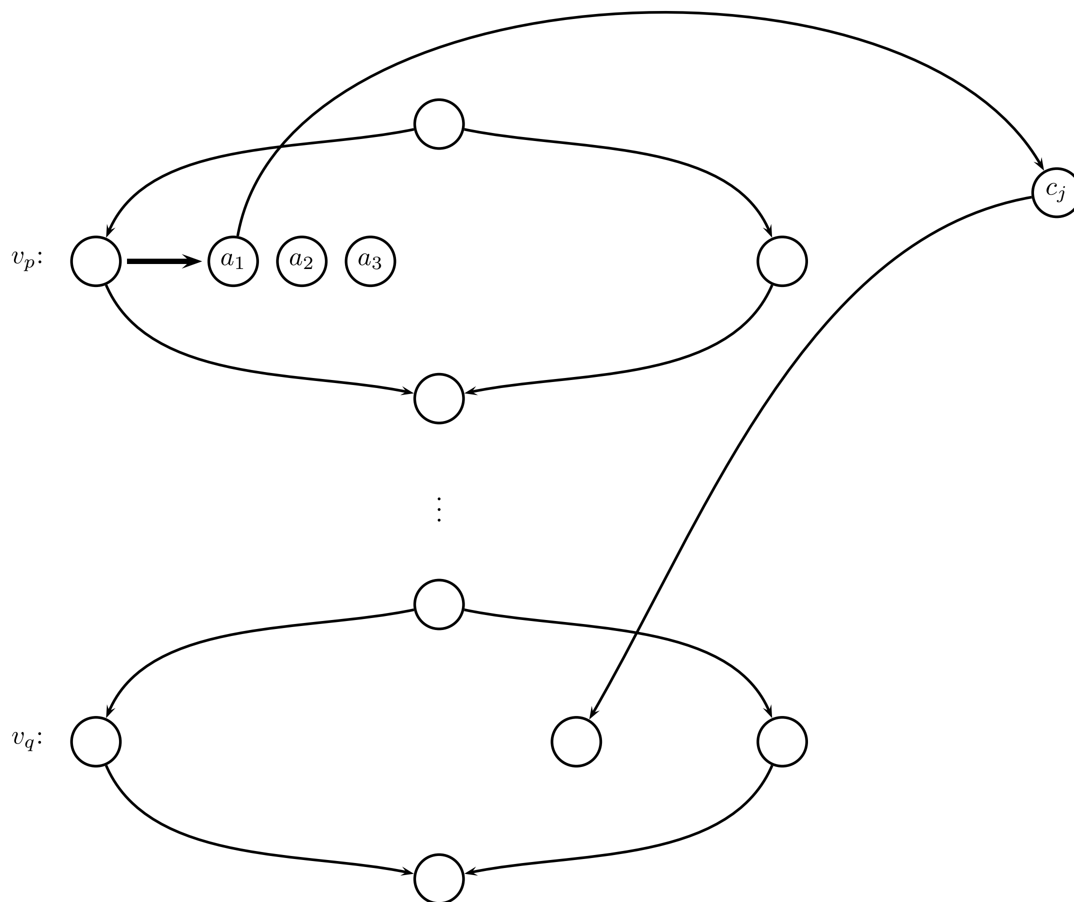


Figure 8: This can't happen.

2. **UNDIRECTED HAMILTONIAN CYCLE is NP-hard.** We will prove this by reducing DIRECTED HAMILTONIAN CYCLE to it. That is, we will show that

   DIRECTED HAMILTONIAN CYCLE $\leq_P$ UNDIRECTED HAMILTONIAN CYCLE

   So we start with an instance of DIRECTED HAMILTONIAN CYCLE—this is just a directed graph G—and we will construct from this an undirected graph H which has a Hamiltonian cycle iff G does.

   Here's how it works: Each vertex v in G corresponds to three vertices v^{in}, v^{mid}, and v^{out} in H. They are connected by two (undirected) edges: one between v^{in} and v^{mid}, and the other between v^{mid} and v^{out}. Those are all the vertices in H, and some of the edges.

   The rest of the edges in H mirror the edges in G: If (u, v) is a (directed) edge in G, we create an edge in H from u^{out} to v^{in}.

   And that's it. Now we have to prove that this works.

3.15 Lemma *If G has a Hamiltonian cycle, then H does.*

PROOF. This is simple. If $u_1 \to u_2 \to \cdots \to u_n \to u_1$ is a (directed) Hamiltonian cycle in G, then

$$u_1^{\text{in}} \leftrightarrow u_1^{\text{mid}} \leftrightarrow u_1^{\text{out}} \leftrightarrow u_2^{\text{in}} \leftrightarrow u_2^{\text{mid}} \leftrightarrow u_2^{\text{out}} \ldots \leftrightarrow u_n^{\text{in}} \leftrightarrow u_n^{\text{mid}} \leftrightarrow u_n^{\text{out}} \leftrightarrow u_1^{\text{in}}$$

is an undirected Hamiltonian cycle in H. □

3.16 Lemma *If H has a Hamiltonian cycle, then G does.*

PROOF. This is where the cleverness of the construction comes in.

Note that since each "mid" node is connected by one edge to an "in" node and one edge to an "out" node, the only way that each "mid" node can be in a cycle is for all three ("in", "mid", "out") nodes to be in that cycle. And from that we can see that a Hamiltonian cycle of H must be of the form

$$u_1^{\text{in}} \leftrightarrow u_1^{\text{mid}} \leftrightarrow u_1^{\text{out}} \leftrightarrow u_2^{\text{in}} \leftrightarrow u_2^{\text{mid}} \leftrightarrow u_2^{\text{out}} \ldots \leftrightarrow u_n^{\text{in}} \leftrightarrow u_n^{\text{mid}} \leftrightarrow u_n^{\text{out}} \leftrightarrow u_1^{\text{in}}$$

But this corresponds exactly to the Hamiltonian cycle $u_1 \to u_2 \to \cdots \to u_n \to u_1$ in G. □

Finally, again it is clear that the construction of H from G is a polynomial-time construction, and the proof is complete. □

3.7 Some more simple reductions

3.7.1 The Traveling Salesman problem

We have a set of cities. Think of these cities as vertices in a graph. A salesman (this problem was considered before gender-neutral language was understood to be important) needs to visit each city as cheaply as possible. We will assume that the cost is the total distance of the trip. So we have a graph and between each two vertices there is an edge with an associated weight. We want to find the shortest path that visits each node. To make things simple, we may also assume that the path should be a simple cycle (and in fact, this is the most common way the problem is stated, and is probably the most useful version).

The associated decision problem is "Does this graph have a Hamiltonian cycle of weight $\le W$?"

We can reduce UNDIRECTED HAMILTONIAN CYCLE to this problem as follows: Let G be any undirected graph. This will be our instance of UNDIRECTED HAMILTONIAN CYCLE. We will construct a graph H with edge weights that will be an instance of the traveling salesman problem, as follows:

- The vertices of H are just the vertices of G.
- Every two vertices of H are connected by an edge. (That is, H is a complete graph.)
- The weight of an edge in H is 0 if that edge is also an edge in G, and is 1 otherwise.

Then the question "Does H have a Hamiltonian cycle of weight ≤ 0?" has a positive answer iff G has a Hamiltonian cycle. Thus the traveling salesman problem is NP-complete.

3.7.2 Subgraph isomorphism

The problem is this: given two graphs G and H, is H isomorphic to some subgraph of G? Again, the problem is clearly in NP. It's NP-hard because we can reduce CLIQUE to it. To ask the question "Does G have a clique of size k?" is to ask the question "Does G have a subgraph that is isomorphic to the complete graph on k vertices?" So CLIQUE $\leq_P$ SUBGRAPH ISOMORPHISM, and so SUBGRAPH ISOMORPHISM is NP-complete.

3.8 Changing the problem a little may change it a lot.

Suppose we consider the GRAPH ISOMORPHISM problem: given two graphs G and H, are they isomorphic? This seems like it should be pretty close to the SUBGRAPH ISOMORPHISM problem, but it doesn't seem to be. In fact, although GRAPH ISOMORPHISM is in NP, it is unknown whether it is NP-complete.

Here is another problem: Given an undirected graph, is there a path that includes every edge exactly once? (It may include a vertex more than once.) This seems similar to HAMILTONIAN CYCLE, but it's not close to it at all. Such a path—one that includes every edge once—is called an Eulerian path. The problem was first considered by Euler in 1735 in a famous paper titled "The Seven Bridges of Königsberg". Euler gave a very simple algorithm ($O(n)$, in fact) for deciding whether this problem had a solution. This paper is regarded as the starting point of the study of the topology of graphs.

4 SAT is NP-complete!

I'm following Aho, Hopcroft, and Ullman (The Design and Analysis of Computer Algorithms) in this last section of the notes.

4.1 Languages and Turing machines

I assume that you have studied something about Turing machines already. I'm not going to go into them in great depth. One important thing to realize is that there are a number of different variants, but they are all equivalent. For instance, a Turing machine can have 1 tape or k tapes. But any k-tape machine has an equivalent 1-tape machine, and in fact the 1-tape machine is at worst polynomially larger and polynomially slower than the k-tape machine. And similarly for the other variants. And in fact the same is true for the relationship between a 1-tape Turing machine and any realistic machine model or programming language. We're not going to prove any of this here, but we will rely on this fact to make our model of computation as simple as possible—that was, after all, the reason Turing defined Turing machines in the first place.

So here is the definition we will use:

Definition *A (1-tape) Turing machine consists of*

- *A finite set Q of* states*. Three of these states are special, and have special names:*
  - *There is an* initial state*, which we will denote by q_0.*

  - *There is an* accepting state*.*
  - *There is a* rejecting state*.*

- *A finite set* $T = \{\tau_0, \tau_1, \tau_2, \dots\}$ *of* tape symbols*. There is a subset* $I \subseteq T$ *of* input symbols*. And there is a special* blank symbol *in* $T \setminus I$*.*

- *A* move *or* transition *function*

$$\delta : Q \times T \to Q \times T \times \{L, R\}$$

- *A* tape*. This is simply a finite sequence of* cells*, which can be extended without limit in both directions. Each cell will contain a* tape symbol*.*

And the machine works like this:

At any point in the computation,

- the machine is in one of the states $q \in Q$,
- the tape has a certain (finite) sequence of tape symbols in its cells, and
- the machine is "pointing to" a particular cell.

These three items of information (the second one of which can be quite large of course, but in any event can be specified in a finite manner) constitute a *configuration*, or a *snapshot*, or an *instantaneous description* (ID) of the machine.

While it might be natural to represent the tape as a 1-dimensional array and to represent "where the machine is currently pointing" as an index into that array, the usual way authors prefer to do this is as follows: A configuration is represented as uq_iv where u and v are (possibly empty) strings of tape symbols, q_i is a state, and the convention is that

- The cells of the tape hold the string uv (i.e., u concatenated with v).
- The machine is pointing to the cell containing the first symbol in v if v is non-empty. If v is empty, the machine is pointing to the "next cell" after u, which contains the blank symbol.

The *initial configuration*, or *initial ID* of the machine is q_0w where w is some string formed from input symbols (i.e., elements of I).

The machine proceeds as follows: If an ID is given by uaq_ibv, where a and b are tape symbols, then $\delta(q_i, b)$ is computed (or, really, looked up, since $Q \times T$ is a finite set). It is of the form $\langle q_j, \tau, L\rangle$ or $\langle q_j, \tau, R\rangle$. L means "move to the left" and R means "move to the right". We consider these two cases separately:

- $\delta(q_i, b) = \langle q_j, \tau, R\rangle$. The machine changes b to τ, enters state q_j, and moves one cell to the right. So the new state is $ua\tau q_jv$.
- $\delta(q_i, b) = \langle q_j, \tau, L\rangle$. The machine changes b to τ, enters state q_j, and moves one cell to the left. So the new state is $uq_ja\tau v$.

This process is then repeated. One of three things can happen:

- Eventually the machine winds up in the accepting state. The machine then stops, and we say that it has accepted the initial input string w.
- Eventually the machine winds up in the rejecting state. The machine then stops, and we say that it has rejected the initial input string w.
- The machine goes on forever without ever entering either the accept or reject states.

So that's what a Turing machine (or at least, the variant of Turing machine that we will use) is.

We need another definition, which you also should be familiar with:

Definition *A* language *(more properly, a* formal language*) over the set T is simply a set of finite strings over T.*

Of course this definition is pretty empty. Real languages—even real computer languages—are normally described by a grammar, or at least by some structural property. But this definition is general enough to allow us to describe the kind of problems we have been considering.

For instance, suppose we are considering UNDIRECTED HAMILTONIAN CYCLE. We can let T be any set of symbols that we can use to write a description of an undirected graph in. And we can define UNDIRECTED HAMILTONIAN CYCLE to be the language consisting of those (finite) strings over T that correspond to undirected graphs having a Hamiltonian cycle.

Thus the problem we have been considering can be rephrased as follows: Given a set T and a language L over T, find an algorithm that takes as input a string over T and tells whether or not it is in L. Now by the Church-Turing hypothesis, we can replace "algorithm" with "Turing machine". If such a machine exists, we say that L is *Turing decidable* or simply *recursive*. Since all the problems we have been considering can be solved by exhaustive search, it is pretty evident that they are all decidable. The problem for us is one of efficiency.

Definition *A language L over a set T is in the class P iff there is a Turing machine that decides L in polynomial time.*

Here of course "in polynomial time" means with respect to the length of the input string.

Of course, when we encode a problem in this way, it might not be true that every string over T even corresponds to a problem of the type we are considering. But this is easily solved: just have the Turing machine walk over the input once and check that it really does represent a graph, or whatever we are representing. If it doesn't, then immediately reject it. If it does, then go on and decide whether it is in L or not.

4.2 Non-deterministic Turing machines and the class NP

Ordinary Turing machines, such as we have described above are *deterministic*. This simply means that every configuration determines the next one. So starting from an initial configuration, the entire sequence of configurations of the machine is completely determined, and there is no choice involved.

It turns out to be useful to consider a more general notion—one in which there *is* some choice, and the machine is therefore *non-deterministic*.

Definition *A* non-deterministic Turing machine *is exactly the same thing as a Turing machine with one exception:*

The transition function δ is no longer a function. Instead it returns a set of possible choices. That is,

$$\delta : Q \times T \to \mathcal{P}\big(Q \times T \times \{L, R\}\big)$$

where for any set X, $\mathcal{P}(X)$ is the power set *of X—that is, the set of all subsets of X. This power set is also often written as 2^X. (You can easily remember this because there are 2^n subsets of a set of size n.)*

The way a non-deterministic Turing machine works is that at each step, one of the choices that δ provides is used to construct the next configuration. You can think of this as follows: at each step in the computation the machine spawns a number of sub-processes. Each sub-process continues the computation, using a different one of the possible choices.

Thus in effect we have a tree of computations, rather than a single linear sequence of computations. If *any* path through this tree winds up in an accepting state, we say that the initial string is accepted by this machine.

Of course this model of computation is not practical as stated. It could, however, be thought of in terms of parallel computations (as above) or also could be implemented in terms of backtracking, which would in general be stupendously inefficient.

Definition *The class NP is the class of languages that are decided in polynomial time by non-deterministic Turing machines.*

Of course a deterministic Turing machine is just a special case of a non-deterministic Turing machine. So with this definition, we know a priori that P $\subseteq$ NP.

On the other hand, it's not quite obvious that this definition is equivalent to the one we have been using up to now, in terms of polynomial-time verifiability. It's actually not hard to show that these two definitions are equivalent, but I'm not going to do it here. See Sipser, for instance, for a quick and short proof of this. In any case, it should be clear that the problems that we have considered so far are in NP in this new definition.

4.3 The Cook-Levin theorem

4.1 Theorem (Cook-Levin) *SAT is NP-complete.*

Proof. We first have to show that SAT is in NP. This is clear: a non-deterministic Turing machine can decide SAT in polynomial time, because each path down the tree can correspond to checking satisfiability for a different choice of truth values for the variables. If the SAT instance is satisfiable, then one of the branches of the computation will report this (i.e., will accept the expression), and this will actually happen in linear time.

So we have to show that SAT is NP-hard. And with our current definition, this means the following: For each language L in NP (say the language is over the set of symbols T), and each finite string w over T, we need to algorithmically create a SAT expression (which, remember, means a Boolean expression in conjunctive normal form), such that

- The algorithm taking the string w to the SAT expression is a polynomial-time algorithm. That is, the time taken by the algorithm is bounded by a constant times some power of $|w|$.
- w is in L iff the SAT expression is satisfiable.

To do this, of course, we need not only to encode the string w in the SAT expression, but also some information about the language L. Now the only thing we really know about L is that it is in NP—that is, that there is a non-deterministic Turing machine M that decides membership in L. So that is the information we will use.

Thus, we start with a non-deterministic Turing machine M that decides a language L over T, described as above, together with its input string w. From M and w we will create a SAT instance by a polynomially bounded function, such that the SAT instance is satisfiable iff M accepts the initial string w. In this way we show that every NP problem is polynomially reducible to SAT, and that will complete the proof.

So let us begin.

We know that M decides L in polynomial time. So we actually have a polynomial p such that M decides each input string w in time $\leq p(|w|)$. Of course p depends on M, but it doesn't depend on anything else. Certainly it doesn't depend on w.

In fact, we are going to make a simplifying assumption: let us assume that the machine has been modified (it is easy to do this, and it doesn't change the problem in any "non-polynomial" way) so that every path of computations is exactly of length $p(|w|)$. In other words, if a computation finished early, it just continues without change until time $p(|w|)$. This simplifies some of the reasoning below.

We are now going to do what we probably should have done originally, and represent the string on tape at each step of the computation as an array. And since the computation for w takes time at most $p(|w|)$, we know that M can move a distance of at most $p(|w|)$ cells in either direction from its start position. Therefore we know that in this case we can index this array from $-p(|w|)$ to $p(|w|)$, where the machine initially points to position 0.

Also, for notational convenience, let us say that

- α is the index among the states of the accepting state. So q_α is the accepting state.
- β is the index among the tape symbols of the blank symbol. So τ_β is the blank symbol.
- The tape symbols in the initial string w are $w_0, w_1, \ldots, w_{|w|-1}$. For convenience, let us write

  $$w_i = \tau_{t(i)}$$

  That is, $t(i)$ is simply the index of the tape symbol w_i, where w_i is the ith symbol of the initial string w.

Now to construct our SAT expression, we need to specify what the variables in that expression are going to be. Each of these variables will say something about the state of our machine M at some point in the computation. To make things as simple as possible, we name our variables using three capital letters:

- C stands for *cell*.

- S stands for *state*.
- H stands for *head*. Most authors, instead of saying that the machine "points to" a certain cell, say that the machine has a "read/write head" (like that on a tape recorder) that is positioned at a certain cell. That's where the term *head* comes from.

Now here are the variables:

- $C_{i,j,t}$ is a variable whose value is True iff the ith cell on the tape contains the tape symbol τ_j at time t. Note that this is really a whole family of variables: there are $2p(|w|)+1$ possible values for i, $|T|$ possible values for j, and $p(|w|)$ possible values for t, so we have just defined

  $$\big(2p(|w|)+1\big)|T|p(|w|)$$

  variables.

  The particular value of this is completely unimportant. The only significance is that the number of these variables is bounded by a fixed polynomial in $|w|$.
- $S_{k,t}$ is a variable whose value is True iff M is in state q_k at time t. Here we have $|Q|p(|w|)$ variables.
- $H_{i,t}$ is a variable whose value is True iff at time t the machine is pointing at cell i. Here we have $\big(2p(|w|)+1\big)p(|w|)$ variables.

Now we will create our Boolean SAT expression. It will be convenient to use an auxiliary function—you can think of this as a macro: $U(x_1, x_2, \ldots, x_r)$ will be True iff exactly one of its arguments is True (and the rest are False). To construct an expression for U, note that

$$\begin{aligned} x_1 \vee x_2 \vee \cdots \vee x_n \quad &\text{is True} \iff \text{at least one of the variables is True} \\ \bigwedge_{1\le i<j\le n} (\bar{x}_i \vee \bar{x}_j) \quad &\text{is True} \iff \text{no two of the variables are True} \end{aligned}$$

Therefore, we can write

$$U(x_1, x_2, \ldots, x_n) = (x_1 \vee x_2 \vee \cdots \vee x_n) \wedge \bigwedge_{1\le i<j\le n} (\bar{x}_i \vee \bar{x}_j)$$

Note that this expression is in conjunctive normal form.

We are going to construct a SAT expression as follows: We will create expressions A, B, C, and so on to encode the following 7 statements:

A**:** The machine is pointing to exactly one cell at each time. (We will call this the "current cell".)

B**:** Each cell contains exactly one tape symbol.

C**:** The machine is in exactly one state at each time.

D**:** Exactly one tape cell (the current cell) is modified from one time to the next.

E**:** The first configuration is the initial configuration.

F**:** The state in the last configuration is the accepting state.

G**:** The change in state, current cell, and cell contents between successive times is allowed by the transition function of M.

Each of these expressions will be an expression in conjunctive normal form. Therefore the conjunction of them (i.e., "and"-ing them together) again yields an expression in conjunctive normal form. And that expression will be True iff the machine M accepts w.

Here are the expressions:

A**.** This asserts that the machine is pointing to exactly one cell at each time.

Let A_t assert that at time t the machine is pointing to exactly one cell. Then $A = A_0 \wedge A_1 \wedge \cdots \wedge A_{p(|w|)}$. And we have

$$A_t = U(H_{-p(|w|),t}, \ldots, H_{-1,t}, H_{0,t}, H_{1,t}, \ldots, H_{p(|w|),t})$$

As we have seen, each A_t is in conjunctive normal form, so A is as well.

B**.** This asserts that each cell contains exactly one tape symbol . Let $B_{i,t}$ assert that the ith cell contains exactly one symbol at time t. Then $B = \bigwedge_{i,t} B_{i,t}$, and

$$B_{i,t} = U(C_{i,1,t}, C_{i,2,t}, \ldots, C_{i,|T|,t})$$

Again, B is in conjunctive normal form.

C**.** This asserts that the machine is in exactly one state at each time.

We have

$$C = \bigwedge_{0 \le t \le p(|w|)} U(S_{1,t}, S_{2,t}, \ldots, S_{|Q|,t})$$

C is in conjunctive normal form.

D**.** This asserts that exactly one tape cell (the current cell) is modified from one time to the next. We have

$$D = \bigwedge_{\substack{0 \le t < p(|w|) \\ -p(|w|) \le i \le p(|w|) \\ 1 \le j \le |T|}} (\overline{C_{i,j,t}} \vee H_{i,t} \vee C_{i,j,t+1})$$

This can be read as follows:

- Either the ith cell does not contain the symbol τ_j at time t,
- or it does, in which case either
  - it also contains it at time $t+1$, or
  - the machine is pointing at the ith cell at time t.

D is in conjunctive normal form.

E. This asserts that the first configuration is the initial configuration.

$$E = S_{0,0} \wedge H_{0,0} \wedge \bigwedge_{i=-p(|w|)}^{-1} C_{i,\beta,0} \wedge \bigwedge_{i=0}^{|w|-1} C_{i,t(i),0} \wedge \bigwedge_{i=|w|}^{p(n)} C_{i,\beta,0}$$

(Note the use of the subscript $t(i)$ in the specification of the initial string.) E is in conjunctive normal form.

F. This asserts that the state in the last configuration is the accepting state q_α.

$$F = S_{\alpha,p(|w|)}$$

F is in conjunctive normal form.

G. This asserts that the change in state, current cell, and cell contents between successive times is allowed by the transition function δ of M.

This expression is more complex than the previous ones. Let us pretend to start out with that our Turing machine is actually deterministic, so that for each state q and tape symbol τ, there is a unique state q', tape symbol τ' and direction d' (i.e., either L or R) such that $\delta(q,\tau)$ is $\langle q', \tau', d\rangle$.

In such a case, G will be the conjunction (i.e., the "and") of a number of clause groups. Each clause group is composed of three clauses, like this:

$$\overline{H_{i,t}} \vee \overline{S_{k,t}} \vee \overline{C_{i,j,t}} \vee H_{i',t+1}$$
$$\overline{H_{i,t}} \vee \overline{S_{k,t}} \vee \overline{C_{i,j,t}} \vee S_{k',t+1}$$
$$\overline{H_{i,t}} \vee \overline{S_{k,t}} \vee \overline{C_{i,j,t}} \vee C_{i,j',t+1}$$

where there is one such clause group for each t from 0 to $p(|w|) - 1$ and for each set of values (i, j, k, i', j', k') such that $-p(|w|) \leq i \leq p(|w|)$ and such that when we compute $\delta(q_k, \tau_j)$, either

- $\delta(q_k, \tau_j) = \langle q_{k'}, \tau_{j'}, R\rangle$, in which case we set $i' = i + 1$, or
- $\delta(q_k, \tau_j) = \langle q_{k'}, \tau_{j'}, L\rangle$, in which case we set $i' = i - 1$.

The way to understand these clauses is as follows: the only way $\overline{H_{i,t}} \vee \overline{S_{k,t}} \vee \overline{C_{i,j,t}}$ can be false is if at time t,

- the machine is pointing at cell i,
- the machine is in state k,
- and the cell i contains τ_j.

In that case

- The first clause says that at time $t+1$, the machine is pointing to cell i' (which is either cell $i+1$ or cell $i-1$).

- The second clause says that at time $t+1$ the machine is in state k'.
- And the third clause says that at time $t+1$ cell i contains $\tau_{j'}$.

Clearly G is in conjunctive normal form. However, in general, our Turing machine is non-deterministic. This means that for each state q and tape symbol τ, $\delta(q, \tau)$ is a (finite!) set of triples of the form

$$\begin{gathered} \langle q', \tau', d' \rangle \\ \langle q'', \tau'', d'' \rangle \\ \langle q''', \tau''', d''' \rangle \\ \dots \end{gathered}$$

Suppose that each of these possibilities gives rise to an expression G', G'', G''', and so on. Then our real expression G is just

$$G = G' \vee G'' \vee G''' \vee \dots$$

The only problem with this is that G is no longer in conjunctive normal form.

However, this can be easily dealt with. Without going into details, it is clear that just by using the distributive property over and over again, this expression can be rewritten as an expression in conjunctive normal form. This is not entirely trivial: the size of the rewritten expression may be exponentially larger than the size of the original expression.

For our purposes here, that doesn't matter at all. The reason is that the size of each original expression $G' \vee G'' \vee \dots$ does not depend on $|w|$. It only really depends on the properties of the Turing machine itself. Therefore it doesn't really matter how big G is, since the size of G does not vary with $|w|$ either. It's just some (possibly very large) constant. The *number* of such expressions G that we need depends on $|w|$—there are $p(|w|)\big(2p(|w|)+1\big)$ such expressions—and this is a polynomial function of $|w|$, which will be multiplied by the constant size of G, so it is still a polynomial function of $|w|$. And that's all we need.

Further (and I'm not going to beat this into the dirt, although I've done a little of that already), it is easy to see that everything here in this construction is polynomially bounded. Therefore the expression

$$A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$$

is an expression in conjunctive normal form which is derivable in a polynomially bounded way from M and w, and which is satisfiable iff M accepts w.

And that completes the proof. $\square$