

Lecture Note 9

The complexity classes P and NP

draft of September 29, 2026, by [N. Young](#)

© 2018–2026 [Neal E. Young](#). Text: CC BY-NC-SA 4.0; code: MIT; except where noted. See <https://github.com/nealeyoung/algs101/blob/main/LICENSE>.

Contents

9.1	Intractable computational problems	1
9.2	Reductions revisited	1
9.3	Reductions between some problems that are new to us	2
9.3.1	Three similar problems: Clique, Independent Set, and Vertex Cover	2
9.3.2	A very different (?) problem: 3-CNF-SAT	3
9.4	Review of key concepts	4
9.5	The class NP—problems that have poly-time verifiers	6
9.5.1	NP-hard problems; NP-complete problems	6
9.5.2	3-CNF-SAT reduces to Subset Sum	7
9.5.3	3-CNF-SAT reduces to Hamiltonian Path	10
9.6	External sources on P, NP, and NP-completeness	12

9.1 Intractable computational problems

Many important computational problems are *intractable*—not solvable by any reasonably fast algorithm. Since 1950 or so, computer scientists have developed a fairly deep understanding of intractable problems.

In designing algorithms, it is useful to be able to tell when a problem is intractable, or likely to be intractable. If your problem is intractable, there is no algorithm that is guaranteed to compute a correct solution to every input in reasonable time. In this case, you need some other approach. For example, perhaps you only care about solving a few particular instances of the problem. Then you might focus on solving just those instances. Or, you might look for fast algorithms that are guaranteed to return provably good *approximate* solutions. Or, you might revisit your application and try to find a different way to formally model what you need to do.

We focus here on a particular kind of intractability: whether or not the problem has any algorithm whose worst-case running time is bounded by *any polynomial*—that is, for some constant $c > 0$, on any input x , the algorithm takes time $O(|x|^c)$. The class of problems that have such algorithms is called *Polynomial Time*, and is often denoted P (or $\mathbb{P}$). Note that this is a class of *problems*, not algorithms. P is one of many *complexity classes* studied by computer scientists.

We also focus on another particular class of algorithmically important problems, so-called “NP-complete” problems. We have *strong evidence* that these problems are not in P (have no polynomial-time algorithms) so, are very likely to be intractable in that sense.

9.2 Reductions revisited

[Lecture Note 8](#) formally defined polynomial-time reductions, and gave reductions from Bipartite Matching to Max Flow, from Disjoint Paths to Max Flow, from Max Flow to Linear Programming, and from Shortest s - t Path to Linear Programming. These enable us to, for example, use any algorithm for Linear Programming to solve Shortest s - t Path, Max Flow, and Bipartite Matching.

Recall that when considering reductions we focus on *decision problems*—computational problems with ‘yes’/‘no’ answers. For any decision problem A , we define the *language*, $L(A)$ associated with A to be the set of inputs for which the answer is “yes”. For example, the language for the decision problem for Shortest s - t Path is the set

$$L = \{(G, s, t, W) \mid \text{there is an } s\text{-}t \text{ path in } G \text{ of weight at most } W\}.$$

To be formally precise, we have in mind here that each input is encoded in a standard way as a binary string. So, each language is, technically, a set of binary strings. But we can safely forget about this most of the time. It is important mainly in that it gives us a robust and general notion of input size—the size of an input is the number of bits in the encoding of the input.

If an algorithm for a decision problem A says ‘yes’ for a given input x , we say the algorithm *accepts* x . So, if the algorithm is correct, then, for all inputs x the algorithm accepts x if and only if $x \in L(A)$.

Recall from [Lecture Note 8](#) that a reduction from a decision problem A (such as Bipartite Matching) to another decision problem B (such as Max Flow) is, formally, a function r such that, given any input x for problem A , the function produces an equivalent instance $r(x)$ of problem B . *Equivalent* means that x is in $L(A)$ if and only if $r(x)$ is in $L(B)$. The reduction is a *polynomial-time* reduction if the function r can be computed in polynomial time. That is, there is an algorithm that, given input x , outputs $r(x)$ in time polynomial in $|x|$. Recall that the notation

$$A \leq_P B$$

means that there is a polynomial-time reduction from A to B . Informally, it means that A is “no harder” than B . (To be more precise, A is at most “polynomially harder” than B .) Recall the following lemmas from [Lecture Note 8](#):

Lemma 9.1. *Let A and B be decision problems such that A reduces in polynomial time to B . If B has a polynomial-time algorithm, then A has a polynomial-time algorithm. That is, if $A \leq_P B$ and $B \in P$, then $A \in P$.*

Lemma 9.2. *Let A , B , and C be decision problems. If A reduces in polynomial time to B , and B reduces in polynomial time to C , then A reduces in polynomial time to C .*

That is, if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

9.3 Reductions between some problems that are new to us

9.3.1 Three similar problems: Clique, Independent Set, and Vertex Cover

Given an undirected graph $G = (V, E)$, a *clique* is a subset $K \subseteq V$ of the vertices such that every pair of vertices in K has an edge in E . An *independent set* is a subset $I \subseteq V$ of the vertices such that no pair of vertices in I has an edge in E . A *vertex cover* is a subset $C \subseteq V$ of the vertices such that every edge in E has an endpoint in C . Here are the three corresponding decision problems.

For each, the input is $G = (V, E)$ and an integer k .

Clique— does G have a clique of size k or more?

Independent Set— does G have an independent set of size k or more?

Vertex Cover— does G have a vertex cover of size k or less?

Each of these three problems reduces in polynomial time to each of the others.

Reducing Clique to Independent Set. Here is a reduction f_1 of Clique to Independent Set. Given an instance $(G = (V, E), k)$ of Clique, define $f_1(G, k)$ to be the instance $(\overline{G}, k)$ of INDEPENDENT SET, where $\overline{G} = (V, \overline{E})$ has the same vertex set of G , and has exactly those edges that G does not have.

To verify correctness, observe that f_1 can be computed in time polynomial in the size of its input, and that, for all instances (G, k) of Clique, the answer to (G, k) is “yes” (G has a clique of size at least k) if and only if the answer to the instance $f_1(G, k)$ of Independent Set is “yes” ($\overline{G}$ has an independent set of size at least k). This is because a set S of vertices is a clique in G iff S is an independent set in $\overline{G}$.

Reducing Independent Set to Clique. The function f_1 also happens to be a poly-time reduction in the other direction: from INDEPENDENT SET to Clique.

Reducing Independent Set to Vertex Cover. Here is a reduction f_2 of Independent Set to Vertex Cover. Given an instance $(G = (V, E), k)$ of Independent Set, define $f_2(G, k)$ to be the instance (G, k') of Vertex Cover, where $k' = |V| - k$. To verify correctness, observe that f_2 can be computed in time polynomial in the size of its input, and that, for all instances (G, k) of Independent Set, the answer to (G, k) is “yes” (G has an independent set of size k) if and only if the answer to the instance $f_2(G, k)$ of Vertex Cover is “yes” (G has a vertex cover of size at most $k' = |V| - k$). This is because a set S of vertices is an independent set in G iff the complement $V \setminus S$ of S is a vertex cover in G .

Reducing Vertex Cover to Independent Set. The function f_2 also happens to be a poly-time reduction in the other direction: from VERTEX COVER to Independent Set.

Reducing Clique to Vertex Cover. And, by Lemma 9.2 (reductions compose), Clique reduces to Vertex Cover via the reduction $(G, k) \mapsto f_2(f_1(G, k))$.

Thus, we see that all three of these problems are equivalent, in the sense that if any one of them has a poly-time algorithm, then (via the reductions described above) each of them will have a poly-time algorithm.

It would be reasonable to conclude at this point that Clique, Independent Set, and Vertex Cover are, in some sense, just different names for the same problem, and that is why we can reduce each of these problem to the other. But the reason is deeper than that. Next we'll see reductions between problems that, on the surface at least, are not at all similar.

9.3.2 A very different (?) problem: 3-CNF-SAT

Here are two problems that seem very different from those three:

SAT —Given a Boolean formula ϕ , is there an assignment of values (true or false) to the variables of ϕ that *satisfies* the formula (makes it true).

3-CNF-SAT— Given a Boolean formula ϕ in 3-CNF form (an “and” of clauses, each of which is an “or” of at most three literals, each of which is a variable or its negation), is the formula *satisfiable* (that is, can one assign values to the variables to make the formula true).

For example, $\phi = (x \vee \bar{y} \vee z) \wedge (y \vee z) \wedge (\bar{y} \vee \bar{z})$ is satisfiable: take $x = T$, $y = T$, $z = F$.

We remark that SAT is a very “expressive” problem—computers are built from logic gates, so it is not too surprising that Boolean formulae can model computation. See Lemma 34.6 of the text [CLRS](#) and surrounding discussion for more details. Also, SAT reduces pretty easily to 3-CNF-SAT. So 3-CNF-SAT is just as expressive.

Lemma 9.3. $3\text{-CNF-SAT} \leq_p \text{Independent Set}$.

Proof. Here is the reduction. Let ϕ be any instance of 3-CNF-SAT. Construct the following instance $(G = (V, E), k)$ of Independent Set. For each variable x in ϕ , introduce a “variable gadget” consisting of two new “literal” vertices, labeled x and $\bar{x}$, with an edge between them.

(So far, if there are h variables in ϕ , then there will be 2^h independent sets of size h . Each independent set takes exactly one vertex from each variable gadget, and we have in mind that these 2^h independent sets correspond to the 2^h assignments true/false assignments to the variables of ϕ in the natural way.)

Next, for each clause in ϕ , introduce a “clause gadget” consisting of one new vertex for each literal in the clause, with edges between each pair of vertices within each clause gadget. For example, for the clause $x \vee \bar{y} \vee z$, the clause gadget would consist of three new vertices, labeled, respectively, x , $\bar{y}$, and z , connected by three edges.

(So far, if there are h variables and ℓ clauses in ϕ , each independent set of size $h + \ell$ must contain exactly one vertex from each gadget.)

Finally, for every vertex v that occurs in each clause gadget, introduce an edge between that vertex and the “opposite” vertex in the variable gadget. For example, if v is labeled with the variable x , then introduce an edge between v and the vertex labeled $\bar{x}$ in the variable gadget for

x . Or, if v is labeled with $\bar{x}$, then introduce an edge between v and the vertex labeled x in the variable gadget for x .

This defines the graph G . Define the threshold k (the desired independent-set size) to be $h + \ell$ (the number of variables plus clauses, that is, the number of gadgets). This defines the reduction: $f(\phi) = (G, k)$, where G and k are defined from ϕ as described above.

Clearly the reduction can be computed in polynomial time. That is, given ϕ , one can construct G and k in time polynomial in $|\phi|$.

It remains to verify that the reduction f is correct. That is, we need to show

$$(\forall \phi) \quad \phi \text{ is satisfiable} \iff (G, k) \text{ has an independent set of size } k, \text{ where } (G, k) = f(\phi).$$

Above ϕ ranges over all instances of 3-CNF-SAT.

Consider an arbitrary instance ϕ of 3-CNF-SAT.

Let h be the number of variables, let ℓ be the number of clauses. Let $(G, k) = f(\phi)$, so that $k = h + \ell$.

We need to show that ϕ is satisfiable if and only if G has an independent set of size at least k .

Only if ($\implies$). Suppose that ϕ is satisfiable. We need to show that in this case G has an independent set I of size at least k . Fix any assignment A of the variables of ϕ that satisfies ϕ .

We will describe how to construct I from A . Make I contain one vertex from each gadget of G , as follows. For each variable x in ϕ , if A assigns x the value “true”, then make I contain the vertex labeled x from the variable gadget for x in G . Otherwise (A assigns x the value “false”), make I contain the vertex labeled $\bar{x}$ from the variable gadget. For each clause gadget in ϕ , make I contain the vertex from the clause gadget that corresponds to some literal in the clause that A makes true (there must be at least one such literal, since A satisfies the clause).

Clearly I has size $k = h + \ell$. Also, I is an independent set, because it takes only one vertex from each gadget, and (because of the way G is constructed) never takes a clause vertex labeled x and a variable vertex labeled $\bar{x}$, or vice versa.

If ($\impliedby$). Suppose that G has an independent set of size at least $k = h + \ell$. We need to show that in this case ϕ has some satisfying assignment A . Let I be any independent set of size k . Clearly I has exactly one vertex per gadget in G (it can’t have more than one in any gadget, as each gadget is a clique, so it has to have one in each gadget in order to have size k).

Define assignment A from I as follows. For each variable x , consider the variable gadget for x . If I takes the vertex labeled x , then make A assign x the value “true”. Otherwise (I takes the vertex labeled $\bar{x}$) make A assign x the value “false”. This defines the assignment. To see that it is a satisfying assignment, consider any clause of ϕ . I contains some vertex v from the clause gadget, where v is labeled with a literal in the clause—either a variable x or its negation $\bar{x}$. Consider the case when v is labeled with a variable x (the other case is similar). There is an edge in G between v and the vertex (call it v') labeled $\bar{x}$ in x ’s variable gadget. So v' cannot be in the independent set I . So the vertex labeled x in x ’s variable gadget has to be in the independent set I . So the assignment assigns x the value “true”. So the clause is satisfied. $\square$

9.4 Review of key concepts

The previous reduction is perhaps surprising. 3-CNF-SAT reduces to (apparently unrelated) graph problems: Independent Set, Vertex Cover, Clique. What’s going on? Let’s take stock. Here’s a review of the key concepts so far:

decision problem —A *decision problem* is a computational problem such that, for every input, the answer is either “yes” or “no”. In studying computational complexity, we work mostly with decision problems, just to keep things simple. Most computational problems can be cast as decision problems.

language —The *language* associated with a decision problem is the set of inputs for which the answer is “yes” (encoded in any standard binary encoding). We often identify the decision problem with its language.

An algorithm A for a decision problem is said to *accept* an input x if, on input x , A outputs “yes”. Otherwise A *rejects* x .

polynomial-time algorithm —An algorithm runs in *polynomial time* if there is some constant $c > 0$ such that the worst-case running time on any input of size n is $O(n^c)$. Note that this includes, for example algorithms that run in time $O(n \log n)$, because $O(n \log n)$ is $O(n^2)$.

the class P —Called *Polynomial Time*, this is the set of decision problems that have polynomial-time algorithms. This is one of many *complexity classes* studied by computer scientists. (A complexity class is a collection of computational problems, typically defined as containing all problems that have an algorithm that obeys a some given resource constraint.)

reduction —A *reduction* from a decision problem A to a decision problem B is a function f mapping each input x for A to some input $f(x)$ for B , such that the answer for x is “yes” if and only if the answer for $f(x)$ is yes. That is, $(\forall x) x \in L(A) \iff f(x) \in L(B)$. When there is a reduction from A to B , we say A *reduces to* B .

polynomial-time reduction —A reduction f is a *polynomial-time* reduction if there is a polynomial-time algorithm that, given any input x , outputs $f(x)$ in time polynomial in $|x|$. If there is such a reduction, we say A *reduces to* B *in polynomial time*, which we write as $A \leq_p B$.

Lemma 9.1. *If $A \leq_P B$ and $B \in P$, then $A \in P$.*

Lemma 9.2. *If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.*

Here are the decision problems we’ve just considered:

Clique— given graph G and integer k , does G have a clique of size k or more?

Independent Set— Given G and k , does G have an independent set of size k or more?

Vertex Cover— Given G and k , does G have a vertex cover of size k or less?

SAT —Given a Boolean formula ϕ , is there an assignment of values (true or false) to the variables of ϕ that *satisfies* the formula (makes it true).

3-CNF-SAT a.k.a. 3-SAT —SAT restricted to inputs ϕ that are in 3-CNF form (an “and” of clauses, where each clause is an “or” of at most three literals, where each literal is a variable or the negation of a variable).

Here are the reductions we’ve shown:

- $\text{CLIQUE} \leq_p \text{INDEPENDENT SET}$
- $\text{INDEPENDENT SET} \leq_p \text{CLIQUE}$

- INDEPENDENT SET $\leq_p$ VERTEX COVER
- VERTEX COVER $\leq_p$ INDEPENDENT SET
- 3-CNF-SAT $\leq_p$ INDEPENDENT SET (!!!)

9.5 The class NP—problems that have poly-time verifiers

These problems all have something in common: they belong to a class of problems called “NP” (for “non-deterministic polynomial time”). We’ll see that NP is defined as the class of decision problems that have polynomial-time *verifiers*. It is defined this way in order to capture a particular set of problems of interest. Intuitively, these are the problems for which, if you are given the correct solution w for a given input x , you can easily *verify* that the solution w is correct for x . (Even if *finding* a correct solution w for a given x is hard!)

Formally, a *verifier* is an algorithm V that is given two inputs: the problem instance x , and a *witness* w . The witness is also called a *certificate* or *proof* (of membership). The verifier V implicitly defines a language L_V , containing all inputs x for which there is a witness that makes the verifier accept:

$$L_V = \{x : (\exists w) V(x, w) \text{ outputs “yes”}\}.$$

The verifier is a *polynomial-time* verifier if $V(x, w)$ (the verifier run on input (x, w)) runs in time polynomial in the size of just x (regardless of the size of w)!

The class NP is defined as the set of decision problems whose languages have polynomial-time verifiers.

By definition, to show that a problem is in NP, you must show that it has a polynomial-time verifier. For example:

Clique —Given the instance $x = (G, k)$ and witness w , the verifier $V((G, k), w)$ checks that w encodes a subset S of the vertices, and that S is a clique in G of size k .

3-CNF-SAT —Given the instance ϕ and witness w , the verifier $V(\phi, w)$ checks that w encodes an assignment to the variables of ϕ , and that the assignment satisfies each clause in ϕ .

The class NP is defined in this way just because the definition captures many interesting problems, problems for which determining whether a solution exists may be hard, but verifying a solution (once you’ve found it) is easy. (Formally, “easy” means “in polynomial time”.)

Note that any problem in P is also in NP, because, given x and w , the verifier $V(x, w)$ can simply ignore the witness w and run the polynomial-time algorithm on x , and say “yes” to (x, w) if the algorithm says “yes” to x .

9.5.1 NP-hard problems; NP-complete problems

Definition 9.1. A decision problem B is NP-hard if, for every problem A in NP, A reduces in polynomial time to B . B is NP-complete if B is in NP and B is NP-hard.

If B is any NP-hard problem, and B has a polynomial-time algorithm, then *every* problem in NP has a polynomial-time algorithm (by reduction to B). So, we know for sure that either

1. no NP-hard problem has a polynomial-time algorithm ($P \neq NP$), or
2. every problem in NP has a polynomial-time algorithm ($P = NP$).

We don't know for sure which case holds—whether or not P equals NP is a main open problem in computer science. *After decades of study, nobody knows a polynomial-time algorithm for any NP-hard problem.* So we expect there are none. But we haven't been able to prove that!

Using NP-hardness. If you have trouble finding a polynomial-time algorithm for a problem, it might be because the problem is NP-hard. To show that a problem is NP-hard, you can reduce some other NP-hard problem to it.

The problems we've considered so far (Clique, Independent Set, Vertex Cover, 3-CNF-SAT) are all NP-complete (although we don't prove it here). So, you can try to reduce one of these problems to your problem.

To finish, we give some more examples of non-trivial reductions, from 3-CNF-SAT to Subset Sum, and from 3-CNF-SAT to a problem called Hamiltonian Path.

9.5.2 3-CNF-SAT reduces to Subset Sum

Recall Subset Sum:

Subset Sum — *Given non-negative integers $x_1, x_2, \dots, x_n$ and a target T , is there a subset S of indices such that $\sum_{i \in S} x_i = T$?*

(We've seen a dynamic-programming algorithm that runs in $O(nT)$ time. Note that that's not polynomial in the size of the input: the standard representation of the instance takes $O(n \log T)$ bits, so, when T is large (for example $T = 2^n$), time $O(nT)$ is not polynomial in the input size. To run in poly-time, the algorithm would have to run in time polynomial in $n \log T$.)

Let's say you've tried but failed to find a polynomial-time algorithm. So you decide to try to show the problem is NP-hard.

The first step is to choose an NP-hard problem to reduce to your problem. Let's try 3-CNF-SAT.

Given a 3-CNF formula ϕ , your goal is to construct an equivalent instance of Subset Sum: a sequence $x_1, x_2, \dots, x_n$ of numbers and a target T such that some subsequence of the numbers sums to T if and only if ϕ is satisfiable.

To get started, try to invent a “variable gadget” to model the notion that a variable v_i of ϕ has to be assigned “true” and “false”. Here's one way: take the sequence of numbers $x_1 = 1, x_2 = 1$, and total $T = 1$. For this small instance, you have to take exactly one of the two numbers to reach total T . We can interpret taking x_1 as setting variable v_i to true, and taking x_2 as setting variable v_i to false.

Next, try to somehow combine multiple variable gadgets into a single instance. For this, use numbers with multiple digits, using one digit for each variable. For example, combining the gadgets for three variables v_1, v_2, v_3 gives

$$\begin{array}{rcl}
 x_1 & = & 100 \text{ (for } v_1) \\
 x_2 & = & 100 \text{ (for } \bar{v}_1) \\
 x_3 & = & 010 \text{ (for } v_2) \\
 x_4 & = & 010 \text{ (for } \bar{v}_2) \\
 x_5 & = & 001 \text{ (for } v_3) \\
 x_6 & = & 001 \text{ (for } \bar{v}_3) \\
 \hline
 T & = & 111
 \end{array}$$

Above, the numbers are written in decimal. Because each column contains at most two 1's, when we add a subset of the numbers, there will be no carry. Hence, to reach a total of T , you have to

take x_1 or x_2 , then x_3 or x_4 , then x_5 or x_6 : there are 8 subsets that sum to T , each representing one of the 8 assignments to the three variables v_1, v_2, v_3 .

Next, how can we model that each clause in ϕ must be satisfied? Consider clause $c_1 = v_1 \vee \bar{v}_2 \vee v_3$. We can ensure that one literal in this clause will be true by adding a digit for c_1 as follows:

	v_1	v_2	v_3	c_1	
x_1	=	1	0	0	1 (v_1)
x_2	=	1	0	0	0 ($\bar{v}_1$)
x_3	=	0	1	0	0 (v_2)
x_4	=	0	1	0	1 ($\bar{v}_2$)
x_5	=	0	0	1	1 (v_3)
x_6	=	0	0	1	0 ($\bar{v}_3$)
T	=	1	1	1	1

Whereas there were 8 subsets that summed to T before, now only those that choose exactly one of x_1, x_4, x_5 will work. These subsets correspond to assignments that make exactly one of the three literals $v_1, \bar{v}_2, v_3$ true.

This doesn't quite capture the notion of satisfying a clause, because a clause is also satisfied if two or three of its literals are true. To fix this, we add two "filler" numbers for c_1 , and increase the digit in T from 1 to 3:

	v_1	v_2	v_3	c_1	
x_1	=	1	0	0	1 (v_1)
x_2	=	1	0	0	0 ($\bar{v}_1$)
x_3	=	0	1	0	0 (v_2)
x_4	=	0	1	0	1 ($\bar{v}_2$)
x_5	=	0	0	1	1 (v_3)
x_6	=	0	0	1	0 ($\bar{v}_3$)
x_7	=	0	0	0	1 (c_1)
x_8	=	0	0	0	1 (c_1)
T	=	1	1	1	3

We've now captured exactly those assignments that satisfy clause c_1 . For each of the remaining clauses, we add a digit (and a pair of filler numbers). For example, for the second clause $c_2 = \bar{v}_1 \vee v_2$, add a digit and two filler numbers to get

	v_1	v_2	v_3	c_1	c_2	
x_1	=	1	0	0	1	0 (v_1)
x_2	=	1	0	0	0	1 ($\bar{v}_1$)
x_3	=	0	1	0	0	1 (v_2)
x_4	=	0	1	0	1	0 ($\bar{v}_2$)
x_5	=	0	0	1	1	0 (v_3)
x_6	=	0	0	1	0	0 ($\bar{v}_3$)
x_7	=	0	0	0	1	0 (c_1)
x_8	=	0	0	0	1	0 (c_1)
x_9	=	0	0	0	0	1 (c_2)
x_{10}	=	0	0	0	0	1 (c_2)
T	=	1	1	1	3	3

With the ideas in place, we can now prove that 3-CNF-SAT reduces to Subset Sum.

Lemma 9.4. $3\text{-CNF-SAT} \leq_p \text{Subset Sum}$.

Proof. First we define the reduction, then we prove it correct.

The reduction. Consider any instance ϕ of 3-CNF-SAT. Let n be the number of variables in ϕ . Let m be the number of clauses. From ϕ , construct a Subset Sum instance $I = (x, T)$ as follows. Every number in the sequence will have $n + m$ decimal digits, each either 0 or 1. The target T will have $n + m$ decimal digits: n ones followed by m threes.

For each variable v_i , introduce two numbers into the instance I ; denote these numbers $x(v_i)$ and $x(\bar{v}_i)$. Make the i th digits of $x(v_i)$ and $x(\bar{v}_i)$ be 1. For each clause c_j , if literal v_i occurs in c_j , then make the $(n + j)$ 'th digit of $x(v_i)$ be 1; if literal $\bar{v}_i$ occurs in c_j , then make the $(n + j)$ 'th digit of $x(\bar{v}_i)$ be 1. Make the remaining digits 0.

For each clause c_j , introduce two copies of one “filler” number into the instance I ; denote the number $x(c_j)$. Make the $(n + j)$ 'th digit of this number be 1. Make the remaining digits 0.

This completes the description of the Subset Sum instance $I = (x, T)$. It's easy to see that the reduction takes polynomial time (given ϕ , the collection of numbers and T can be constructed in polynomial time).

Proof of correctness. To finish we prove that the reduction is correct. That is, given any 3-CNF-SAT instance ϕ , the reduction outputs a Subset-Sum instance $(x, T) = f(\phi)$ such that some subsequence of the numbers in x sums to T if and only if ϕ is satisfiable:

$$(\forall \phi) \phi \in \text{3-CNF-SAT} \iff f(\phi) \in \text{Subset Sum}.$$

Consider any 3-CNF-SAT instance ϕ . Let $I = (x, T)$ be the Subset Sum instance produced by the reduction.

Only if ($\implies$). Assume ϕ is satisfiable. We need to show that in this case some subsequence S of x sums to T . Let A be the assignment that satisfies ϕ . Construct a subsequence S from A as follows. For each variable v_i , if A sets v_i to be “true”, then make S contain $x(v_i)$, and otherwise make S contain $x(\bar{v}_i)$.

By inspection of the reduction, the i th digit of the sum of numbers in S will be 1.

For each clause c_j , A gives c_j either one, two, or three distinct true literals. if A gives c_j one true literal, make S contain both copies of $x(c_j)$. If A gives c_j two true literals, make S contain one copy of $x(c_j)$. If A gives c_j three true literals, make S contain neither copy of $x(c_j)$. This guarantees that the $(n + j)$ th digit of the sum of numbers in S is 3. Hence, the sum of the numbers in S equals T .

By this part of the proof, ϕ is satisfiable only if there is a subsequence of x summing to T .

If ($\impliedby$). Suppose some subsequence S of the numbers sums to T . We need to show that ϕ is satisfiable. From S , construct an assignment A to the variables of ϕ as follows.

Each of the numbers in I has $n + m$ decimal digits (each 0 or 1), and, for each i , at most five numbers in I have non-zero i th digit. Hence, in adding the numbers in S , no carry occurs. The first n digits of T are 1's, and the last m digits are 3's, so, (i) for each $i \leq n$, the subsequence S must have exactly one number whose i th digit is 1, and (ii) for $n < i \leq n + m$, the subsequence S must have exactly three numbers whose i th digit is 1.

From (i) and the construction of each $x(v_i)$ and $x(\bar{v}_i)$, it follows that S must have exactly one of the two numbers $x(v_i)$ or $x(\bar{v}_i)$. Define assignment A by assigning v_i to be “true” if S contains $x(v_i)$, and “false” otherwise.

For each clause c_j , there are only two “filler” numbers (copies of $x(c_j)$) in I that have non-zero $(n + j)$ 'th digit. By (ii), there are three numbers in S that have non-zero $(n + j)$ th digit, so there must be some number $x(v_i)$ or $x(\bar{v}_i)$ in S that has non-zero $(n + j)$ 'th digit. In the case that some

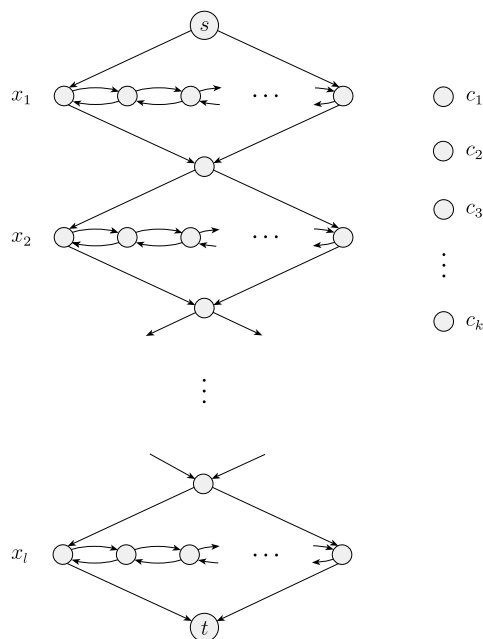


FIGURE 7.49
The high-level structure of G

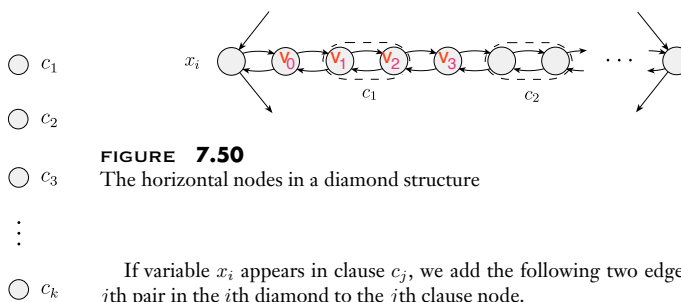


FIGURE 7.50
The horizontal nodes in a diamond structure

If variable x_i appears in clause c_j , we add the following two edges from the j th pair in the i th diamond to the j th clause node.

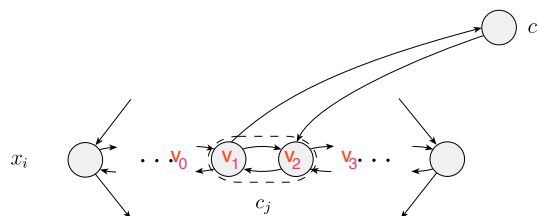


FIGURE 7.51
The additional edges when clause c_j contains x_i

Figure 9.1: Figures 7.49–7.51 of Sipser’s *Introduction to the Theory of Computation*, © Michael Sipser, reproduced here for commentary; they are not covered by these notes’ license.

such $x(v_i)$ is in S , it must be that v_i occurs in clause c_j , so the assignment (which sets v_i true, because $x(v_i) \in S$) satisfies the clause. Otherwise some such $x(\bar{v}_i)$ is in S , and $\bar{v}_i$ occurs in clause c_j , so the assignment (which sets v_i false, because $x(\bar{v}_i) \in S$) satisfies the clause.

By this part of the proof, if there is a subsequence of x summing to T , then ϕ is satisfiable.

By the parts above, for any 3-CNF formula ϕ , the reduction, given ϕ , outputs a subset-sum instance (x, T) such that ϕ is satisfiable if and only if x has a subsequence summing to T . That is, the reduction is correct. $\square$

9.5.3 3-CNF-SAT reduces to Hamiltonian Path

To conclude, we’ll sketch one more reduction: from 3-CNF-SAT to Hamiltonian Path:

Hamiltonian Path —Given directed graph $G = (V, E)$ and $s, t \in V$, is there an s - t path in G that visits each vertex exactly once.

Lemma 9.5. $3\text{-CNF-SAT} \leq_p \text{Hamiltonian Path}$.

Proof sketch. First we describe the reduction, then we prove it is correct.

The reduction. Given any instance ϕ of 3-CNF-SAT, the reduction $f(\phi)$ outputs (G, s, t) where digraph G , source s , and sink t are as shown in Figures 7.49–7.51. (We’ve borrowed these figures from the excellent text *Theory of Computation*, by Sipser.)

The global structure of G is shown in Sipser’s Fig. 7.49. Each variable x_i in ϕ has a variable gadget in G . Each clause in ϕ has a clause gadget. The variable gadget for x_i consists of a long, doubly linked list of vertices with edges from each endpoint of the list to an intermediate vertex

below. (See Sipser's Fig. 7.50.) The vertices of the list come in groups: for each clause c_j there is a pair of adjacent vertices, and between consecutive pairs, and at each end of the list, there is a *separator* vertex, whose only edges are those of the list. The gadgets for the variables $x_1, x_2, \dots, x_n$ are connected in sequence as shown in Fig. 7.49, ending with the sink t .

For each clause c_j , the clause gadget has a single new vertex labeled c_j , and edges as shown in Fig. 7.51. For each literal x or $\bar{x}$ in the clause, we select a unique adjacent pair of vertices (u, w) in the variable gadget for x (where u is to the left of w in the list). If the literal is a variable x , we add edges from u to the clause vertex c_j and from that vertex to w . Otherwise (the literal is a negation, $\bar{x}$), we add edges in the opposite direction: from w to the clause vertex and from the clause vertex to u . In either case, we call the path from u to w (or vice versa) formed by the two edges a *detour*.

This completes the reduction: given any 3-CNF-SAT formula ϕ , the reduction produces the Hamiltonian Path instance $f(\phi) = (G, s, t)$ consisting of the graph G , source s and sink t described above.

Clearly the reduction can be computed in time polynomial in $|\phi|$.

Proof of correctness. To finish, we need to prove that the reduction is correct. That is, we need to show

$$(\forall \phi) \phi \in \text{3-CNF-SAT} \iff f(\phi) \in \text{Hamiltonian Path}.$$

Consider any instance ϕ of 3-CNF-SAT. Let $(G, s, t) = f(\phi)$ as described above.

Only if ($\implies$). Suppose that ϕ is satisfiable. We need to show that, in this case, G has a Hamiltonian path P . Let A be the assignment that satisfies ϕ . Construct P from A as follows.

Start at s . If A assigns x_1 the value “true”, then traverse the variable gadget for x_1 from left to right. Along the way, for each clause c_j that contains the literal x_1 , we will encounter a pair of adjacent vertices u, w (in that order) such that u has an edge to the clause vertex for c_j , which in turn has an edge back to w . Take this “detour” (the two edges) to visit the clause vertex, returning to w and continuing to traverse from left to right.

Or, if A assigns x_1 the value “false”, then traverse the variable gadget for x_1 from *right to left*. Along the way, for each clause c_j that contains the literal $\bar{x}_1$, we will encounter a pair of adjacent vertices u, w (in that order) such that u has an edge to the clause vertex for c_j , which in turn has an edge back to w . Take this “detour” (the two edges) to visit the clause vertex, returning to w and continuing to traverse from right to left.

Continue by, traversing the variable gadgets for x_2, x_3 , etc. similarly: (traverse the gadget for x_i left-to-right if A assigns x_i the value “true”, and otherwise right-to-left; detour to visit clause vertices that contain the corresponding literal, unless the clause vertex has already been visited once). Stop when we reach t . In this way, we can traverse from s to t , and visit each vertex exactly once.

By this part of the proof, ϕ is satisfiable only if G has a Hamiltonian path.

If ($\impliedby$). Suppose that G has a Hamiltonian path P from s to t . We need to show that, in this case, ϕ has some satisfying assignment A . Construct A from P as follows.

First, we argue that the path P has to be *well-formed*, meaning that it corresponds to some assignment. Specifically, P must traverse the gadgets for variables $x_1, x_2, \dots, x_n$, in order, and, for each i , must traverse the variable gadget for x_i either left-to-right (with detours to clause gadgets of clauses that contain literal x_i) or right-to-left (with detours to clause gadgets of clauses that contain literal $\bar{x}_i$).

We claim that *if P takes any edge on some detour, then it has to take the other edge as well.*

Next we prove the claim. Note Figures 7.50 and 7.51. Consider any detour (v_1, c_j, v_2) from the gadget for a variable x_i . Assume v_1 is just to the left of v_2 and clause c_j contains literal x_i (the case when v_1 is just to the right of v_2 and clause c_j contains literal $\bar{x}_i$ is similar). Let v_0, v_1, v_2, v_3 be the sub-path in the variable gadget around v_1, v_2 .

Vertex v_3 has only one neighbor other than v_2 , so P contains exactly one of the two edges (v_2, v_3) or (v_3, v_2) . In the case that P contains (v_3, v_2) , it also contains (v_2, v_1) (because no other edge leaves v_2). So (i) P contains either the edge (v_2, v_3) or the sub-path $P_3 = (v_3, v_2, v_1)$.

Similarly, vertex v_0 has only one neighbor other than v_1 , so P contains exactly one of the two edges (v_0, v_1) or (v_1, v_0) . In the case that P contains (v_1, v_0) , it also contains (v_2, v_1) (because no other edge enters v_1). So (ii) P contains either the edge (v_0, v_1) or the sub-path $P'_3 = (v_2, v_1, v_0)$.

First consider the case that P contains (v_2, v_3) . Then P can't contain P'_3 (which has edge (v_2, v_1)), so by (ii) must contain (v_0, v_1) along with (v_2, v_3) . If P also happens to contain (v_1, v_2) , then P can't contain either edge of the detour. Otherwise (P does not contain (v_1, v_2)), P has to contain the first edge of the detour because that is the only way left out of v_1 , and it has to contain the second edge of the detour (because that is the only way left into v_2). So, if P contains (v_2, v_3) , then the claim holds.

Next consider the remaining case, by (i): P contains $P_3 = (v_3, v_2, v_1)$. Then P can't contain (v_0, v_1) (because P contains (v_2, v_1)), so by (ii) must contain $P'_3 = (v_2, v_1, v_0)$ as well as P_3 , so must contain sub-path (v_3, v_2, v_1, v_0) , and cannot contain either edge of the detour. So the claim holds in this case as well. This completes the proof of the claim.

We now finish the proof that ϕ has a satisfying assignment A . Since, for each detour, P contains either both edges of the detour or neither, replacing each detour (v_1, c_j, v_2) from P by the edge (v_1, v_2) gives a path P' that visits all the nodes except the clause nodes. Considering the global structure of G (Fig. 7.49), this path P' must traverse the gadgets for the variables $x_1, x_2, \dots, x_n$ in sequence, and must traverse each gadget either left-to-right or right-to-left. Let A assign the value “true” to each variable x_i whose gadget P' traverses left-to-right. Let A assign “false” to each variable x_i whose gadget P' traverses right-to-left. This assignment assigns values to all the variables. To see that the assignment has to satisfy ϕ , consider any clause c_j . The original path P traversed this clause, and (by the claim) did so via some detour (v_1, c_j, v_2) from the gadget of a variable x_i such that x_i or $\bar{x}_i$ occurs in clause c_j . Consider the case that x_i occurs in clause c_j . (The case when $\bar{x}_i$ occurs in the clause is similar.) Then, by the construction of G , in the detour v_1 is just to the left of v_2 , which (considering P' and the claim) means that P' must traverse the gadget for x_i left-to-right, which by the way A is constructed that A assigns variable x_i value “true”, so A satisfies clause c_j .

By this part of the proof, if G has a Hamiltonian path then ϕ is satisfiable.

By the parts of the proof above, for all 3-CNF formulas ϕ , ϕ is satisfiable if and only if $f(\phi)$ has a Hamiltonian path. Therefore, the reduction is correct. $\square$

9.6 External sources on P , NP , and NP -completeness

- [DPV](#) Chapter 8. [JE](#) Chapter 12. [KT](#) Chapter 8 (and [KW slides](#)). [CLRS](#) Chapter 34. MIT video:
 - <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-23-computational-complexity/>
- Chapter 7.3 of [Sipser's Theory of Computation](#)