REVIEW OF PART 1

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, $(\forall x)\ x \in A \iff x \in B$. We 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 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 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 $A \leq_P C$.

some decision problems

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

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

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

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

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 3.

- Clique $\leq_p$ Independent Set
- Independent Set $\leq_p$ Clique
- Independent Set $\leq_p$ Vertex Cover
- Vertex Cover $\leq_p$ Independent Set
- 3-CNF-SAT $\leq_p$ Independent Set

PART 2

NP is the class of problems that have poly-time verifiers. NP (non-deterministic polynomial time) is defined as the class of decision problems that have polynomial-time *verifiers*. NP is defined this way just 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. (However, *finding* a correct solution w for a given x may be 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 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)!

Any verifier V implicitly defines a language L_V, containing all inputs x that have a witness that makes the verifier accept:

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

The class NP is 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:

Composite — Given an integer x, does x have a non-trivial factor? The verifier $V(x, w)$ checks that w is an integer, and that w is a non-trivial factor of x. If so, the verifier says "yes". Otherwise, the verifier says "no."

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.

Hamiltonian Path — Given the instance (G, s, t) and witness w, the verifier $V((G, s, t), w)$ checks that w encodes a path P in G, and that P is an s-t path that visits each vertex exactly once.

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.

NP-hard problems, and NP-complete problems

Definition 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 NP-hard, and B has a polynomial-time algorithm, then every problem in NP has a polynomial-time algorithm (by reduction to B). We know for sure that either

1. no NP-complete problems 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, Hamiltonian Path) are all NP-complete (although we don't prove it here). So, you can try to reduce one of these problems to your problem.

Reducing 3-CNF-SAT to Subset Sum. For example, suppose you are studying the following problem:

Definition 2.

Subset Sum — *Given non-negative integers $x_1, x_2, \ldots, x_n$ 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 $n \log T$.)

You're unable 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 $x_1, x_2, \ldots, x_n$ 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

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

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 ~~$c_1 = v_1 \wedge \overline{v}_2 \wedge v_3$~~ $c_1 = v_1 \vee \overline{v}_2 \vee v_3$. 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	$\mathbf{c}_1$	
x_1	=	1	0	0	**1**	(v_1)
x_2	=	1	0	0	**0**	($\overline{v}_1$)
x_3	=	0	1	0	**0**	(v_2)
x_4	=	0	1	0	**1**	($\overline{v}_2$)
x_5	=	0	0	1	**1**	(v_3)
x_6	=	0	0	1	**0**	($\overline{v}_3$)
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 $v_1, \overline{v}_2, v_3$ 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	($\overline{v}_1$)
x_3	=	0	1	0	0	(v_2)
x_4	=	0	1	0	1	($\overline{v}_2$)
x_5	=	0	0	1	1	(v_3)
x_6	=	0	0	1	0	($\overline{v}_3$)
$\mathbf{x}_7$	=	0	0	0	**1**	(c_1)
$\mathbf{x}_8$	=	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 $c_2 = \overline{v}_1 \vee v_2$, add a digit and two filler numbers to get

		v_1	v_2	v_3	c_1	$\mathbf{c}_2$	
x_1	=	1	0	0	1	**0**	(v_1)
x_2	=	1	0	0	0	**1**	$(\overline{v}_1)$
x_3	=	0	1	0	0	**1**	(v_2)
x_4	=	0	1	0	1	**0**	$(\overline{v}_2)$
x_5	=	0	0	1	1	**0**	(v_3)
x_6	=	0	0	1	0	**0**	$(\overline{v}_3)$
x_7	=	0	0	0	1	**0**	(c_1)
x_8	=	0	0	0	1	**0**	(c_1)
$\mathbf{x}_9$	=	0	0	0	0	**1**	(c_2)
$\mathbf{x}_{10}$	=	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 4. *3-CNF-SAT* $\leq_p$ 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 $x(\overline{v}_i)$. Make the ith digits of $x(v_i)$ and $x(\overline{v}_i)$ 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 $\overline{v}_i$ occurs in c_j, then make the $(n + j)$'th digit of $x(\overline{v}_i)$ 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 ($\Longrightarrow$). 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 $x(\overline{v}_i)$.

By inspection of the reduction, the ith 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 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 ($\Longleftarrow$). 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 three numbers in I have non-zero ith 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 ith digit is 1*, and (ii) *for $n < i \leq m$, the subsequence S must have exactly three numbers whose ith digit is 1.*

From (i) and the construction of each $x(v_i)$ and $x(\overline{v}_i)$, it follows that S must have exactly one of the two numbers $x(v_i)$ or $x(\overline{v}_i)$. Define assignment A by assigning x_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 $x(\overline{v}_i)$ in S that has non-zero $(n+j)$'th digit. In the case that some 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 $x(\overline{v_i})$ is in S, and $\overline{v_i}$ occurs in clause c_j, so the assignment (which sets v_i false, because $x(\overline{v}_i) \in S$) 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. □

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

Lemma 5. 3-CNF-SAT $\leq_p$ 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

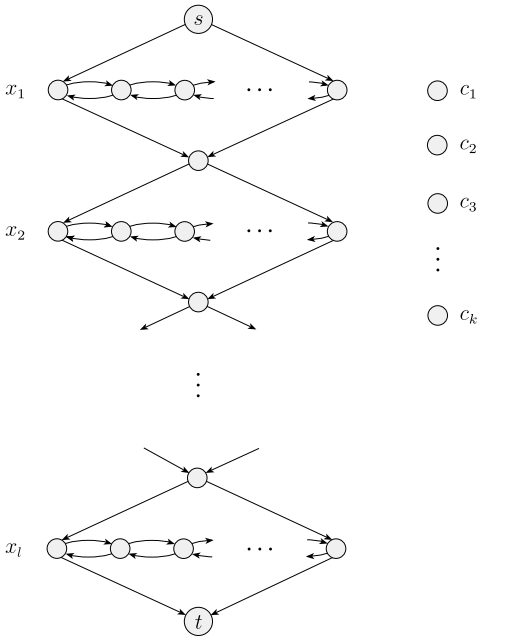

FIGURE **7.49**
The high-level structure of G

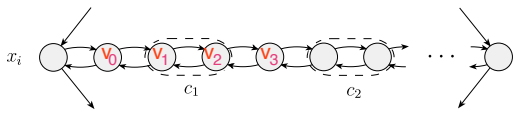

FIGURE **7.50**
The horizontal nodes in a diamond structure

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

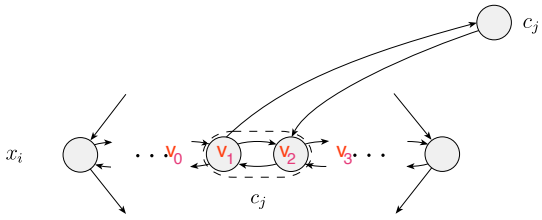

FIGURE **7.51**
The additional edges when clause c_j contains x_i

vertex below. (See Sipser's Fig. 7.50.) The gadgets for the variables $x_1, x_2, \ldots, x_n$ 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 $\overline{x}$ 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, $\overline{x}$), 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 ($\Longrightarrow$). 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 $\overline{x_1}$, 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.

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 ($\Longleftarrow$). 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 $x_1, x_2, \ldots, x_n$, 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 $\overline{x}_i$).

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 $\overline{x_i}$ 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 $x_1, x_2, \ldots, x_n$

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 is 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 $\overline{x}_i$ occurs in clause c_j. Consider the case that x_i occurs in clause c_j. (The case when $\overline{x}_i$ 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. □

External resources on P, NP, and NP-completeness

- CLRS Chapter 34. Dasgupta et al. Chapter 8. Kleinberg & Tardos Chapter 8.
- Erickson's Lecture 30:

  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/30-nphard.pdf
- MIT lecture videos
  - 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*