

8 NP-Hardness

So far, we have been presented a problem (e.g., minimum spanning tree, maximum flow), and our goal was to solve it by designing and analyzing an algorithm (e.g., Kruskal’s, Ford-Fulkerson). In this section, we will do something quite different. First, instead of considering optimization problems, we will consider decision problems. In a decision problem, the solution is either “yes” or “no.”

Any optimization problem can be turned into a decision problem by introducing an additional parameter. For example, the decision version of the shortest s - t path problem is “Does there exist a path from s to t in G of length at most k ?” For the Knapsack problem, it’s “Does there exist a subset of items with weight at most W and value at least k ?” For maximum flow, it’s “Does there exist a feasible flow whose value is at least k ?” and so on.

Secondly, instead of solving the decision problem, we will actually show that *no* polynomial-time algorithm can solve the problem.¹ In other words, instead of showing that a problem can be optimally solved in, say, $O(n^2)$ time, in this chapter, we’ll show that some problems cannot be solved even in $O(n^{100})$ time.

8.1 Basic Computational Complexity

Remark. For our purposes, it is not critical to fully understand the definitions of P and NP. Our focus will be on giving reductions from one decision problem to another. For more on this topic, I recommend *Introduction to the Theory of Computation* by Michael Sipser.

Informally speaking, a decision problem A is in **P** if it can be solved in polynomial time, and it is in **NP** if, for any “yes” instance of A , we can verify that a proposed solution is correct in polynomial time.² For example, consider the shortest s - t path problem again. The input is a graph G , vertices s and t , and a number k . This problem is in P because we can find a shortest s - t path using BFS. Now we show that it is in NP. A proposed solution is simply an s - t path P , and by counting the number of edges in P , we can verify that its length is at most k . This verification process takes linear time, which is polynomial, so the shortest s - t path problem is in NP.

In fact, any problem in P is in NP. Roughly speaking, this is because if a problem is in P, then we don’t need to verify any solution to be convinced that the answer is “yes” — in polynomial time, we can simply solve the problem ourselves from scratch. One intuitive way of interpreting this is that solving a problem is harder than verifying the solution to a problem (for example, think about Sudoku puzzles).

Polynomial-time reductions. Let A and B be decision problems. A *polynomial-time reduction* from A to B is a polynomial-time algorithm f that transforms every instance X of A into an instance $f(X)$ of B such that X is a “yes” instance of A if and only if $f(X)$ is a “yes” instance of B . We write $X \in A$ to denote that X is a “yes” instance for A ; thus, the

¹This statement is true only if we assume $P \neq NP$, which most computer scientists believe, but is perhaps the biggest open question in computer science.

²What do P and NP stand for? As you can guess, P stands for “polynomial,” because these problems can be solved in polynomial time. But NP does *not* stand for “not polynomial,” but rather, “nondeterministic polynomial time.”

reduction f must satisfy the following: $X \in A$ if and only if $f(X) \in B$. If such a reduction exists, we write $A \leq_P B$.

Intuitively, if $A \leq_P B$, then A is at most as hard as B . This is because any polynomial-time algorithm ALG_B for B yields a polynomial-time algorithm for A as follows:

1. On an instance X of A , apply the reduction to convert X into $f(X)$.
2. Run ALG_B on $f(X)$ and return whatever it returns (i.e., “yes” or “no”).³

The reduction itself only describes how to transform X into $f(X)$. But to prove that the reduction is correct, we must show $X \in A$ if and only if $f(X) \in B$, and this involves two proofs: the “forward” direction is “ $X \in A \implies f(X) \in B$,” and the “backward” direction is “ $f(X) \in B \implies X \in A$.” So finding a correct reduction can be tricky because the reduction must simultaneously work for both directions.

Proving NP-completeness. A problem A is **NP-hard** if every problem in NP is reducible to A , and A is **NP-complete** if it is both in NP and NP-hard.⁴ We think of NP-complete problems as the “hardest” problems in NP, because a polynomial-time algorithm for any of them would produce (as described above) a polynomial-time algorithm for every problem in NP. It is not obvious that NP-hard problems even exist, but in this chapter, we’ll see multiple examples.

To show that a problem B is NP-complete, we need to show that it is in NP and that it is NP-hard. Usually, showing that B is in NP is quite straightforward, so we won’t focus on this. (Again, for our purposes, it is not even important to precisely understand the definitions of P and NP.) To show that B is NP-hard, it suffices to prove that $A \leq_P B$ for some NP-hard problem A .

8.2 3-SAT to Independent Set

In this section, we give a polynomial-time reduction from the 3-SAT problem to the INDSET problem, defined below. Recall that these are decision problems, so for each instance of the problem, the answer is either “yes” or “no.”

- **3-SAT.** Input: A set of clauses over n Boolean variables $x_1, \dots, x_n \in \{0, 1\}$. Each clause is the disjunction (“OR”) of 3 literals, where a literal is a variable or the negation of a variable. (For example, $x_1 \vee \bar{x}_2 \vee x_5$ is one possible clause.) A clause is *satisfied* if at least one of its literals is set to 1. Question: Does there exist an assignment of the x_i variables to 0/1 (i.e., False/True) such that all clauses are satisfied?

³One might think that to show $A \leq_P B$, we can assume that we are given a polynomial-time algorithm for B , and we must use this to design a polynomial-time algorithm for A . In other words, in the algorithm for problem A above, why can we only use ALG_B once, and why are we forced to immediately output whatever it outputs? This is a fair question; in fact, this kind of reduction is known as a Cook reduction, while our definition is known as a Karp reduction. However, for reasons relevant to TCS but beyond the scope of these notes, we must stick with Karp reductions.

⁴There are NP-hard problems outside NP, such as the Halting Problem. For the other direction, Ladner’s theorem states that if $P \neq NP$, then there are problems in NP that are neither in P nor are they NP-hard. But again, nobody has proven $P \neq NP$.

- **INDSET.** Input: An undirected graph G and an integer k . An *independent set* is a subset S of vertices such that no two vertices in S share an edge. Question: Does G contain an independent set of size at least k ?

To reduce 3-SAT to INDSET, we need to describe a polynomial-time algorithm f that transforms any instance X of 3-SAT into an instance $f(X)$ of INDSET such that X is a “yes” instance of 3-SAT if and only if $f(X)$ is a “yes” instance of INDSET.

Theorem 8.1. *The 3-SAT problem is reducible to INDSET.*

Proof. Suppose the 3-SAT instance has ℓ clauses C_1, \dots, C_ℓ . For each C_i , we add a triangle of vertices v_i^1, v_i^2, v_i^3 to G for a total of 3ℓ vertices and 3ℓ edges so far. For any two vertices in G , if their correspond literals are negations of each other, then we add an edge between these two vertices. This completes the construction of G ; for k , we set its value equal to ℓ .

To prove the forward direction, suppose there is an assignment of each x_i to 0/1 such that all k clauses are satisfied. This means that in each clause, one literal is set to 1. (If multiple literals are 1, we pick one arbitrarily.) Let S be the corresponding set of vertices, so $|S| = \ell$; we claim that S is independent. For contradiction, suppose $u, v \in S$ and (u, v) is an edge of G . This edge is not a “triangle” edge since S has one vertex per triangle, so it must be a “negation” edge, which means u represents a variable x_i and v represents its negation \bar{x}_i (or vice versa). By definition of S , the assignment sets both x_i and \bar{x}_i to 1, contradicting the validity of the assignment.

Now we prove the backward direction. Let S be an independent set of size ℓ ; we must assign each variable x_i to 0/1. Recall that there is a one-to-one relation between vertices of G and literals in clauses. If neither x_i nor \bar{x}_i is in S (in vertex form), then we can set x_i to either 0 or 1. If $x_i \in S$, then $\bar{x}_i \notin S$ because S is independent, so in this case, we can set $x_i = 1$. Similarly, if $\bar{x}_i \in S$, then we set $x_i = 0$ (so $\bar{x}_i = 1$). Notice that we cannot have both x_i and \bar{x}_i in S due to the “negation” edges. We claim that this assignment satisfies every clause. Recall that $|S| = \ell$, so S contains one vertex from each of the ℓ triangles in G . In each triangle, the literal corresponding to the vertex in S is set to 1, so the clause corresponding to that triangle is satisfied. \square

8.3 Vertex Cover to Dominating Set

Our next reduction is from the Vertex Cover problem to the Dominating Set problem. On the surface, the two problems look similar (both seek to find a subset of vertices that “cover” elements of the graph); our reduction will follow this intuition.

- **VERTEXCOVER.** Input: An undirected graph G and an integer k . A *vertex cover* is a subset S of vertices such that every edge of G is incident to at least one vertex in S . Question: Does G contain a vertex cover of size at most k ?
- **DOMSET.** Input: An undirected graph G and an integer k . A *dominating set* is a subset S of vertices such that every vertex of G is either in S or is adjacent to at least one vertex in S . Question: Does G contain a dominating set of size at most k ?

Theorem 8.2. *The VERTEXCOVER problem is reducible to DOMSET.*

Proof. Let (G, k) denote an instance of VERTEXCOVER; we shall construct an instance (G', k') of DOMSET. Initialize G' as G . Additionally, for each edge $e = (u, v)$ of G , we add a vertex x_e to G' , as well as the edges (u, x_e) and (v, x_e) . So we can imagine G' as being G with an additional copy of each edge, and there is a new vertex placed “on” each copy. We’ll call these new vertices “edge” vertices. Finally, we set $k' = k$.

Let’s prove the forward direction. Let S be a vertex cover of G such that $|S| = k = k'$; we claim that S is a dominating set in G' . Consider any vertex u in G' . If u is an “edge” vertex x_e , then the corresponding edge e is covered by some vertex in S , and u is adjacent to that vertex. Otherwise, if u is a “normal” vertex, then consider any edge (u, v) incident to u . Since S is a vertex cover that doesn’t contain u , it must contain v . Thus, u is adjacent to $v \in S$, so we can conclude S is a dominating set in G' .

Now we prove the backward direction. Let S' be a dominating set of G' such that $|S'| = k' = k$. Notice that the vertices in S' are not necessarily in G because G' contains the “edge” vertices. However, consider any “edge” vertex $x_e \in S'$, where $e = (u, v)$. Let’s remove x_e from S' and replace it with u (or v). (If both u and v were already in S' , then we can remove x_e from S' and S' remains a dominating set.) Then S' is still dominating because the three vertices dominated by x_e (i.e., x_e, u, v) are also dominated by u .

Thus, we can replace all “edge” vertices in S' with either of their endpoints and S' remains a dominating set. Now we can claim that S' is a vertex cover of G . Consider any edge $e = (u, v)$ in G . Since the vertex x_e in G' is dominated by S' , at least one of its neighbors $\{u, v\}$ must be in S' , so e is incident to at least one vertex in S' . \square

8.4 Subset Sum to Scheduling

Our final reduction is from a knapsack-like problem to a scheduling problem. Although the latter sounds complicated, in the reduction, the instance we construct is fairly simple.

- SUBSETSUM. Input: Positive integers x_1, \dots, x_n and a target value T . Question: Is there a subset of these integers that sums to exactly T ?
- SCHEDULE. Input: A set of jobs labeled $\{1, \dots, n\}$, where job i has release time $r_i \geq 0$, deadline $d_i > r_i$, and length $\ell_i \geq 1$. (All values are integers.) There is 1 machine; we must allocate every job i to a contiguous block of length ℓ_i in the interval $[r_i, d_i]$. Question: Is there an allocation that finishes all jobs by their deadline?

Theorem 8.3. *The SUBSETSUM problem is reducible to SCHEDULE.*

Proof. Given an instance of SUBSETSUM $(\{x_1, \dots, x_n\}, T)$, we must construct an instance of SCHEDULE. Let $X = \sum_{i=1}^n x_i$ denote the sum of the input integers. For each x_i , we create a job i with $(r_i, d_i, \ell_i) = (0, X + 1, x_i)$. We also create a special $(n + 1)$ -th job, with $(r_{n+1}, d_{n+1}, \ell_{n+1}) = (T, T + 1, 1)$. This completes the reduction.

For the forward direction, suppose there exists $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} x_i = T$. In any feasible schedule, we must assign job $n + 1$ to the interval $[T, T + 1]$. The jobs in S have total length T , so we can assign them (in any order) to finish exactly at time T . Finally, the amount of time between $T + 1$ and $X + 1$ is $X - T$, which is exactly the amount of time the remaining jobs require. Thus, we can order them arbitrarily in $[T + 1, X + 1]$.

Now we prove the backward direction. Again, any feasible schedule must assign job $n+1$ to $[T, T+1]$. This leaves X units of time to schedule the first n jobs, whose total length is X . So a feasible schedule cannot leave any idle time on the machine. In particular, the interval $[0, T]$ must be fully packed with jobs, so these jobs have total length T . Thus, the integers corresponding to these jobs must sum to exactly T . \square

8.5 More Hard Problems

The following problems are all NP-complete, and some of the exercises will refer to them. Unless otherwise stated, all of the input graphs are connected and undirected. Note that the nomenclature for these problems is not universally consistent. But generally speaking, the different versions of a problem are reducible to each another, so for the purposes of NP-hardness, the exact definition doesn't matter.

- **CLIQUE**. Input: A graph G and an integer k . A *clique* is a subset S of vertices such that there is an edge between every pair of vertices in S . Question: Does G contain a clique of size at least k ?
- **LONGESTPATH**. Input: A graph G , where each edge has some nonnegative length, two vertices s, t in G , and an integer k . Question: Does there exist a path (i.e., no repeated vertices) from s to t of length at least k ?
- **HAM-ST-PATH**. Input: A graph G and two vertices s, t in G . A *Hamiltonian path* is a path that visits every vertex exactly once. Question: Does G contain a Hamiltonian path from s to t ?
- **HAMCYCLE**. Input: A graph G . A *Hamiltonian cycle* is a cycle that visits every vertex exactly once. Question: Does G contain a Hamiltonian cycle?
- **TSP (Traveling Salesman Problem)**. Input: A complete graph G where every edge has a nonnegative length, and a value k . Question: Does G contain a Hamiltonian cycle of total length at most k ?
- **3D-MATCHING**. Input: Three disjoint sets U, V, W , all of size n , and a set of m triples $E \subseteq U \times V \times W$. Question: Does there exist a subset S of n triples such that each element of $U \cup V \cup W$ is contained in exactly one triple in S ?
- **SETCOVER**. Input: A set of n elements U , sets $S_1, \dots, S_m \subseteq U$, and an integer k . A *cover* is a collection of subsets whose union is U . Question: Does there exist a cover of size at most k ?
- **COLORING**. Input: A graph G an integer k . A *k -coloring* of G is a function $f : V \rightarrow \{1, \dots, k\}$; it is *proper* if, for every edge (u, v) , $f(u) \neq f(v)$. Question: Does G contain a proper k -coloring?

8.6 Exercises

- 8.1. Let A, B, C be decision problems. Prove the following facts, which form the basis of this whole chapter:
 - (a) Suppose A is NP-hard and $A \leq_P B$. Prove that B is NP-hard.
 - (b) Prove that if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.
- 8.2. Prove that $P = NP$ implies that every problem in P is NP-complete.
- 8.3. Prove $\text{INDSET} \leq_P \text{VERTEXCOVER}$ and vice versa.
- 8.4. Prove $\text{VERTEXCOVER} \leq_P \text{SETCOVER}$.
- 8.5. Suppose we want to reduce VERTEXCOVER to DOMSET . Explain why the reduction is not correct if we set $G' = G$ and $k' = k$.
- 8.6. Prove $\text{HAM-ST-PATH} \leq_P \text{HAMCYCLE}$ and vice versa.
- 8.7. Consider the HAM-ST-PATH problem, but suppose s and t are not specified. Show that the HAM-ST-PATH problem reduces to this version, and vice versa.
- 8.8. The COLORING problem is NP-hard even for $k = 3$. Consider the following variant: Assuming the number of vertices in the input graph G is a multiple of 3, the question is whether or not G contains a proper 3-coloring that uses each color the same number of times. Prove that this problem is also NP-hard.
- 8.9. Suppose we are given an oracle that solves the HAM-ST-PATH (decision) problem in polynomial time. Use this oracle to design a polynomial-time algorithm that returns a Hamiltonian s - t Path if one exists, or outputs “No Ham-st-Path” if none exists.
- 8.10. Consider the following problem: The input is an undirected graph G , and the question is whether or not G contains a spanning tree in which the degree of every vertex (in the tree) is at most 3. Prove that this problem is NP-hard. (Hint: You may want to use a result from a previous exercise.)
- 8.11. Consider the following problem: The input is a sequence of positive integers x_1, \dots, x_n , and the question is whether or not there exists a subset S of these integers such that the sum of integers in S is equal to the sum of integers not in S . Prove that this problem is NP-hard.
- 8.12. Let G be an undirected graph and k be an integer. A *strongly independent set* (SIS) S is an independent set such that the distance between any two vertices in S is at least 3. Question: Does G contain an SIS of size at least k ? Prove that this problem is NP-hard.
- 8.13. Consider the 3-SAT problem, but now the question is this: Are there *at least two* assignments of the x_i variables to 0/1 such that all clauses are satisfied? Prove that this problem is NP-hard.

- 8.14. Suppose we are given an oracle that solves the CLIQUE (decision) problem in polynomial time. Use this oracle to design a polynomial-time algorithm that finds a clique of maximum size. Do the same for VERTEXCOVER (i.e., find a minimum vertex cover).