7 NP-Hard Problems

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. Firstly, instead of considering the optimization problem directly, we will consider its decision version. In the decision version, there is (usually) an additional parameter k. Instead of minimizing the cost of a solution, our goal is to simply determine if a solution with value at most k exists. (If our original goal was to find a solution of maximum value, then the question is whether or not a solution with value at least k exists.)

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 cost 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 giving an algorithm to solve a 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 show that many problems cannot be solved even in $O(n^{100})$ time.

7.1 Basic Complexity Theory

A decision problem is in the complexity class **P** if it can be decided in polynomial time. The problem is in **NP** if a solution can be verified in polynomial time. In other words, if the answer is "yes," then there should exist a "proof" that convinces us of this in polynomial time. We usually think of the "proof" as a "proposed solution."[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; let's show it's also in NP. A proposed solution is a path from s to t. To verify this solution, we simply need to check that it meets the definition of an s-t path, and its total 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 a proof in order 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). So if a problem can be solved in polynomial time, then its solution can be verified in polynomial time.

[1] This statement is true only if we assume P $\neq$ NP, which most computer scientists believe, but remains as perhaps the biggest open question in computer science. For a clear and thorough treatment of this topic, I recommend *Introduction to the Theory of Computation* by Michael Sipser.

[2] What 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." We will not go into the details; again, I recommend Sipser's textbook.

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 an instance X of A into an instance $f(X)$ of B such that X is a "yes" instance for A if and only if $f(X)$ is a "yes" instance for B. We write $X \in A$ to denote that X is a "yes" instance for A; thus, the 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 produces 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").

(There could be a faster algorithm for A, so A is at most as hard as B.)

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.[3]

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-complete (or NP-hard) problems even exist, but all of the remaining problems in this chapter are NP-complete.

The typical problem in complexity asks you to prove that a problem B is NP-complete. Usually, showing that B is in NP is straightforward, so we won't focus on this. To show that B is NP-hard, it suffices to prove that $A \leq_P B$ for some NP-hard problem A (we leave this proof as an exercise).

7.2 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:

- 3-Sat. Input: A set of clauses over n Boolean variables $x_1, \ldots, x_n \in \{0, 1\}$. Each clause is the disjunction ("OR") of 3 literals, where a literal is a variable or the negation of a

[3] One 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 beyond the scope of these notes, we must stick with Karp reductions: we can use ALG_B once, and we must output whatever it outputs.

[4] There are NP-hard problems outside NP, such as the Halting Problem. The other direction is addressed by a result known as Ladner's theorem, which states that if P $\neq$ NP, then there are problems in NP that are not in P but also not NP-complete. But again, nobody has proven P $\neq$ NP.

variable. (For example, $x_1 \vee \overline{x_2} \vee x_5$ 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?

- IndSet. Input: An undirected graph G and value 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?

Recall that we need to do three things: transform an instance of 3-Sat (i.e., a set of clauses) into an instance of IndSet (i.e., a graph G and a value of k), prove the "forward direction" (if there is a satisfying assignment for the 3-Sat instance, then G has an independent set of size at least k), and prove the "backward direction" (the converse).

Theorem 7.1. *The* 3-Sat *problem is reducible to* IndSet.

Proof. Suppose the 3-Sat instance has ℓ clauses $C_1, \ldots, C_\ell$. 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. So vertex v_i^j in G represents the j-th literal in clause C_i. 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 simply 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, so it must be a "negation" edge, which means u represents a variable x_i and v represents its negation $\overline{x_i}$ (or vice versa). By definition of S, the assignment sets both x_i and $\overline{x_i}$ 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 $\overline{x_i}$ is in S (as a vertex), then we can set x_i to either 0 or 1. If $x_i \in S$, then $\overline{x_i} \notin S$ because S is independent, so in this case, we can set $x_i = 1$. Similarly, if $\overline{x_i} \in S$, then we set $x_i = 0$ (so $\overline{x_i} = 1$). 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. □

7.3 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 value 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 value 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 7.2. *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 $k' = k$.

Now we 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 $u \notin S$, 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 S is a dominating set.

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 (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'. □

7.4 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 $x_1, \ldots, x_n$ 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 $\{1, \ldots, n\}$, 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 7.3. *The* SubsetSum *problem is reducible to* Schedule.

Proof. Given an instance of SubsetSum $(\{x_1, \ldots, x_n\}, T)$, 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 $[r_i, d_i] = [0, X + 1]$ and $\ell_i = x_i$. We also create a special $(n + 1)$-th job, with $[r_{n+1}, d_{n+1}] = [T, T + 1]$ and $\ell_{n+1} = 1$. This completes the reduction.

For the forward direction, suppose there exists $S \subseteq \{1, \ldots, n\}$ 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. □

7.5 More Hard Problems

The following problems are all NP-complete, and many 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 consistent across all sources, so for example, you may find a book that uses "HamPath" to refer to a small variant of the problem we describe below. But in these cases, the problems are reducible to one another, so the names of the problem don't really matter.

- Clique. Input: A graph G and value 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 some value k. Question: Does there exist a path (i.e., no repeated vertices) from s to t of length at least k?

- HamPath. 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 $S_1, \ldots, S_m \subseteq U$, and a value 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 and value k. A *k-coloring* of G is a function $f : V \to \{1, \ldots, k\}$; it is *proper* if, for every edge (u, v), $f(u) \neq f(v)$. Question: Does G contain a proper k-coloring?

7.6 Exercises

1. Let A, B, C be decision problems.
   (a) Prove that if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.
   (b) Suppose A is NP-hard and $A \leq_P B$. Prove that B is NP-hard.
2. Suppose P = NP. Prove that every problem in P is NP-complete.
3. Prove IndSet $\leq_P$ VertexCover and vice versa.
4. Prove VertexCover $\leq_P$ SetCover.
5. Consider the reduction from VertexCover to DomSet in Sec. 7.3. Explain what goes wrong if we set $G' = G$ and $k' = k$.
6. Prove HamPath $\leq_P$ HamCycle and vice versa.
7. Consider the HamPath problem, but suppose s and t are not specified. Show that the HamPath problem reduces to this version, and vice versa.
8. Let G be an undirected graph. Question: Does G contain 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 the previous exercise.)
9. Let $x_1, \ldots, x_n$ be positive integers. Question: Does there exist 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.
10. Let G be an undirected graph and k be a value. 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.
11. 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.
12. The Coloring problem is NP-hard even for $k = 3$. Consider the following variant: suppose the number of vertices in the input graph is a multiple of 3, and our goal is to find a proper 3-coloring that uses each color the same number of times. Prove that this problem is also NP-hard.
13. Suppose we are given an oracle that solves the HamPath (decision) problem in polynomial time. Use this oracle to design a polynomial-time algorithm that returns an s-t Hamiltonian Path (if one exists).
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).