1 Definition of Reduction

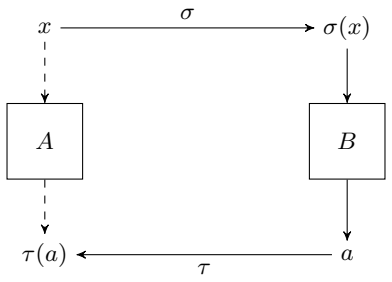
Given a problem A and a problem B, a *reduction* from A to B consists of

- an efficient algorithm σ to transform an input instance of problem A into an input instance of problem B, and
- an efficient algorithm τ to transform the solution of B back to a solution of A,

such that

- if x is an instance of problem A, and if a is a correct solution of problem B on input $\sigma(x)$, then $\tau(a)$ is a correct solution of A on input x.

Intuitively, a reduction allows us to view problem A as a special case of B. We can think of the function σ as encoding instances x of A as instances $\sigma(x)$ of B in such a way that a solution to the B problem on input $\sigma(x)$ gives a solution to the A problem on input x. If we had a fast algorithm for B, this would give a fast algorithm for A by composing it with σ and τ: on input x, we run σ to get $\sigma(x)$, then run the algorithm for B on $\sigma(x)$ to get an output a, then run τ on a to get our final answer $\tau(a)$.



When such a pair of algorithms σ, τ exist and are computable in polynomial time, we write $A \leq_{\mathrm{p}} B$ and say that A *reduces to* B *in polynomial time.*

The picture simplifies a little in the case of *decision problems.* A *decision problem* is a computational problem whose output is a one-bit answer "yes" or "no". If A and B are decision problems, the function τ is typically just the identity function mapping "yes" to "yes" and "no" to "no". In this case, we can simplify the definition: A *reduction* from A to B consists of an efficient algorithm σ to transform an input instance of problem A into an input instance of problem B such that

- if x is a "yes" instance of problem A, then $\sigma(x)$ is a "yes" instance of problem B, and
- if y is a "no" instance of problem A, then $\sigma(y)$ is a "no" instance of problem B.

Equivalently,

- The instance x of problem A is a "yes" instance if and only if $\sigma(x)$ is a "yes" instance of problem B.

2 Specifying Reductions

In practice, giving a reduction $A \leq_{\mathrm{p}} B$ is a four-step process.

1. Specify σ and τ.
2. Show that σ correctly translates any given instance x of A to a well-formed instance $\sigma(x)$ of B.
3. Show that τ translates a correct answer a to problem B for input $\sigma(x)$ to a correct answer $\tau(a)$ to problem A for input x.
4. Analyze the complexity of σ and τ. You want to show small polynomial time. The composition of σ and τ with a polynomial-time algorithm for B will then yield a polynomial-time algorithm for A.

If A and B are decision problems and τ is the identity function, the process is a little simpler:

1. Specify σ.
2. Show that σ correctly translates any given instance x of A to a well-formed instance $\sigma(x)$ of B.
3. Show that x is a "yes" instance of problem A if and only if $\sigma(x)$ is a "yes" instance of problem B.
4. Analyze the complexity of σ. You want to show small polynomial time. The composition of σ with a polynomial-time algorithm for B will then yield a polynomial-time algorithm for A.

A couple caveats:

- Although the reduction is only in one direction (from A to B), this does not say you only need to show one direction of the "if and only if" in step 3. You must show both directions. Equivalently, you must show that if x is a "yes" instance of problem A, then $\sigma(x)$ is a "yes" instance of problem B, and if x is a "no" instance of problem A, then $\sigma(x)$ is a "no" instance of problem B.
- In step 4, the function σ may increase the size of x, and this must be accounted for in the complexity analysis. For example, x may be of size n, but $\sigma(x)$ may be of size n^2. Thus if B had an $O(n^3)$ algorithm, and we applied it to $\sigma(x)$, then the total complexity would be $O((n^2)^3) = O(n^6)$. Luckily, the composition of two polynomial functions is always still a polynomial function.

3 Examples

3.1 A Reduction to Max Flow

Here is a solution of K&T Ch. 7 Ex. 27, a carpooling problem, in which we reduce the problem to max flow. In this example, problem A is the carpooling problem as stated in the text and problem B is the max flow problem in directed graphs. The solution to both parts (a) and (b) will follow from our reduction.

An instance of the carpooling problem consists of a set of participants $P = \{p_1, \ldots, p_k\}$ and a nonempty subset $S_j \subseteq P$ of carpoolers for each day j, $1 \leq j \leq d$. We would like to designate a driver $p_{i_j} \in S_j$ for each day j so that each participant p_i drives at most $\lceil \Delta_i \rceil$ times over the d days, where

$$\Delta_i = \sum_{\substack{1 \leq j \leq d \\ p_i \in S_j}} \frac{1}{|S_j|}. \tag{1}$$

We will prove that this is always possible and give an efficient algorithm to choose the drivers.

Step 1. Specify σ and τ.

To specify σ, we would like to construct a flow graph from a given instance of the carpooling problem. Consider the flow graph with:

- a source node s,
- a sink node t,
- nodes $p_1, \ldots, p_k$ corresponding to the k participants,
- nodes $s_1, \ldots, s_d$ corresponding to the d days,
- edges (s, p_i) of capacity $\lceil \Delta_i \rceil$ for $i \in \{1, \ldots, k\}$,
- edges (p_i, s_j) of infinite (or very large finite) capacity if $p_i \in S_j$, that is, if p_j is carpooling on day i,
- edges (s_j, t) of capacity 1.

We will show below that an integral max flow f in this flow graph has flow value exactly d. Thus there is one unit of flow through each node s_j, so there is exactly one i such that $f(p_i, s_j) = 1$. Choose person p_i as the driver for day j. This is our map τ that constructs a solution for the carpooling problem from a solution to the flow problem.

Step 2. Show that σ correctly translates any given instance x of A to a well-formed instance $\sigma(x)$ of B.

There is not much to say here. We have defined the nodes, edges, capacities, source, and sink, thus we have specified a valid flow graph.

Step 3. Show that τ translates a correct answer a to problem B for input $\sigma(x)$ to a correct answer $\tau(a)$ to problem A for input x.

We must prove that an integral max flow in our flow graph gives a valid driving schedule in which each driver drives no more than $\lceil \Delta_i \rceil$ times in d days.

The first thing to notice is that there exists a valid *nonintegral* flow in this graph of value d. The flow is obtained by sending Δ_i units of flow along the edge (s, p_i) for each i, then splitting that flow and sending $1/|S_j|$ units along the edge (p_i, s_j), and finally sending one unit of flow along the edge (s_j, t) for each j. This flow satisfies the conservation law at nodes p_i by (1) and at nodes s_i by the fact that

$$\sum_{\substack{1 \le i \le k \\ p_i \in S_j}} \frac{1}{|S_j|} = \frac{|S_j|}{|S_j|} = 1.$$

This is a max flow, because there is a cut of capacity d, namely the edges into t.

Because the max flow value is d, and because all the capacities are integers, there is also an *integral* max flow f of value d. By conservation of flow at the nodes s_j, there is exactly one edge (p_i, s_j) with positive flow, and the flow on that edge must be 1 since f is integral. We let p_i drive on day j when that happens. Then p_i drives on no more than $\lceil \Delta_i \rceil$ days by conservation of flow at the node p_i, since the capacity of the edge (s, p_i) is $\lceil \Delta_i \rceil$.

Step 4. Analyze the complexity of σ and τ.

The graph G can be produced from the given instance of the carpool problem in linear time in the size of the representation of the carpool problem, assuming the sets S_j are given as lists. Let $m = \sum_j |S_j|$. Then $d \le m$ and (assuming each participant works at least one day) $k \le m$. We need only construct nodes s, t, $p_1, \ldots, p_k$, and $s_1, \ldots, s_d$, and the edges (s, p_i), (p_i, s_j), and (s_j, t) as above. It takes $O(d + k) \le O(m)$ to

construct the nodes and $O(d+k+m) = O(m)$ to construct the edges. The capacity of the edges (s, p_i) can be calculated in time $O(m)$ in all using the formula (1). All other capacities can be calculated in constant time for each edge. This gives a total cost of $O(m)$ to compute σ.

For τ, from an integral max flow on G, the driving assignments can be calculated in linear time; one need only go through the edges and find those of the form (p_i, s_j) with nonzero flow.

Composed with a max flow algorithm, these maps yield a polynomial-time algorithm for the carpool problem. The flow graph has $O(d+k)$ vertices and $O(m)$ edges, and the max flow value is d. So Ford–Fulkerson runs in time $O(md)$ on this graph. Edmonds–Karp is a special case of Ford–Fulkerson, so its running time is also $O(md)$.

3.2 Reductions Between Hard Problems

Even if we do not know any fast algorithms for problems A and B, we might still be able to efficiently reduce A to B. This would say that if a fast algorithm were found for B tomorrow, then we would automatically have a fast algorithm for A. Similarly, if it were proved tomorrow that no fast algorithm for A exists, then we would automatically know that no fast algorithm for B exists. (This is the case with NP-hardness and NP-completeness proofs. We will discuss NP-hardness and NP-completeness later.)

Here are two reductions between two hard problems, CNFSAT $\leq_{\mathrm{p}}$ CLIQUE and CLIQUE $\leq_{\mathrm{p}}$ CNFSAT. No polynomial-time algorithm is known for either problem. Nevertheless, they are efficiently reducible to each other, so they are equivalent (to within a small polynomial) in complexity, whatever that complexity might be. It is perhaps surprising that two problems so apparently different should be computationally equivalent. However, this turns out to be a widespread phenomenon.

Recall that the CNFSAT problem is: Given a Boolean formula in conjunctive normal form (CNF), say

$$\varphi = \bigwedge_{1 \leq i \leq k} \bigvee_{1 \leq j \leq k_i} \ell_{ij},$$

where each ℓ_{ij} is a literal (a variable or the negation of a variable), decide whether φ has a satisfying truth assignment. The subexpression $\bigvee_{1 \leq j \leq k_i} \ell_{ij}$ is called a *clause*.

The CLIQUE problem is: Given an undirected graph $G = (V, E)$ and a positive integer k, decide whether G contains a k-clique (complete subgraph on k nodes) as a subgraph.

Since both of these problems are decision problems, we can use the simplified procedure of §2.

3.2.1 CNFSAT $\leq_{\mathrm{p}}$ CLIQUE

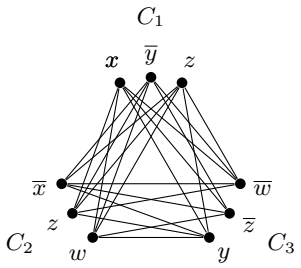
Given a Boolean formula φ in CNF, we want to construct an undirected graph $G = (V, E)$ and specify an integer k such that G has a k-clique iff φ is satisfiable. Note that to satisfy a formula in CNF, a truth assignment must assign the value **1** (true) to at least one literal in each clause, and different occurrences of the same literal in different clauses must receive the same truth value.

Step 1. Specify σ.

We take k to be the number of clauses in φ. We take the vertices of G to be the set of all the *occurrences* of literals in φ. We put an edge between two such occurrences if they are in different clauses and the two literals are not complementary. For example, the formula

$$\underbrace{(x \vee \overline{y} \vee z)}_{C_1} \quad \wedge \quad \underbrace{(\overline{x} \vee z \vee w)}_{C_2} \quad \wedge \quad \underbrace{(y \vee \overline{z} \vee \overline{w})}_{C_3}$$

would yield the graph



Step 2. Show that σ correctly translates any given instance x of A to a well-formed instance $\sigma(x)$ of B.

Again, not much to say. We have constructed an undirected graph G and a number k, which is a valid input to the CLIQUE problem.

Step 3. Show that x is a "yes" instance of problem A if and only if $\sigma(x)$ is a "yes" instance of problem B.

We must argue that the graph G has a k-clique iff φ is satisfiable. Intuitively, an edge between two occurrences of literals represents the ability to assign them both **1** without a local conflict; a k-clique thus represents the ability to assign **1** to at least one literal from each clause without global conflict. In the example above, $k = 3$ and there is a 3-clique (triangle) corresponding to every way to choose true literals to satisfy the formula.

Let us prove formally that G has a k-clique iff φ is satisfiable. First assume that φ is satisfiable. Let $v : \{x_1, \ldots, x_n\} \to \{\mathbf{0}, \mathbf{1}\}$ be a truth assignment satisfying φ. At least one literal in each clause must be assigned **1** under v. Choose one such literal from each clause. The vertices of G corresponding to these true literals are all connected to each other because no pair is complementary; they could not both be **1** otherwise. Thus they form a k-clique.

Conversely, suppose G has a k-clique. Since G is k-partite and the partition elements correspond to the clauses, the k-clique must have exactly one vertex in each clause. Assign **1** to the literals corresponding to the vertices in the clique. This can be done without conflict, since no pair of complementary literals appears in the clique. The complement of any literal assigned **1** must be assigned **0**. Assign truth values to the remaining unassigned variables arbitrarily. The resulting truth assignment assigns **1** to at least one literal in each clause, thus satisfies φ.

Step 4. Analyze the complexity of σ.

If $|\varphi| = n$, then G has $O(n)$ nodes and $O(n^2)$ edges. The nodes can be produced in linear time: they are just pairs (C, ℓ) where C is a clause of φ and ℓ is a literal appearing in C. The edges can be produced in time $O(n^2)$: an edge between (C, ℓ) and (C', ℓ') is included if $C \neq C'$ and ℓ and ℓ' are not complementary literals.

3.2.2 CLIQUE $\leq_{\mathrm{p}}$ CNFSAT

We give a reduction in the other direction as well. To reduce CLIQUE to CNFSAT, we must show how to construct from a given undirected graph $G = (V, E)$ and a number k a Boolean formula φ in CNF such that G has a k-clique if and only if φ is satisfiable.

Step 1. Specify σ.

Given $G = (V, E)$ and k, take as Boolean variables x_i^u for $u \in V$ and $1 \leq i \leq k$. Intuitively, x_i^u says, "u is the i^{th} element of the clique." The formula φ is the conjunction of three subformulas C, D and E, with the following intuitive meanings and formal definitions:

- C = "For every i, $1 \le i \le k$, there is at least one $u \in V$ such that u is the i^{th} element of the clique."

$$C = \bigwedge_{i=1}^{k} \left(\bigvee_{u \in V} x_i^u \right).$$

- D = "For every i, $1 \le i \le k$, no two distinct vertices are both the i^{th} element of the clique."

$$D = \bigwedge_{i=1}^{k} \bigwedge_{\substack{u,v \in V \\ u \neq v}} (\neg x_i^u \vee \neg x_i^v).$$

- E = "If u and v are in the clique, then (u, v) is an edge of G. Equivalently, if (u, v) is not an edge, then either u is not in the clique or v is not in the clique."

$$E = \bigwedge_{(u,v) \notin E} \bigwedge_{1 \le i,j \le k} (\neg x_i^u \vee \neg x_j^v).$$

Step 2. Show that σ correctly translates any given instance x of A to a well-formed instance $\sigma(x)$ of B.

The three constructed formulas C, D, and E are in conjunctive normal form, therefore their conjunction $\varphi = C \wedge D \wedge E$ is. Thus φ is a valid instance of the CNFSAT problem.

Step 3. Show that x is a "yes" instance of problem A if and only if $\sigma(x)$ is a "yes" instance of problem B.

Any satisfying assignment v for $C \wedge D$ picks out a set of k vertices, namely those u such that $v(x_i^u) = \mathbf{1}$ for some i, $1 \le i \le k$. If v also satisfies E, then those k vertices form a clique. Conversely, if $u_1, \ldots, u_k$ is a k-clique in G, set $v(x_i^{u_i}) = \mathbf{1}$, $1 \le i \le k$, and set $v(y) = \mathbf{0}$ for all other variables y; this truth assignment satisfies φ.

Step 4. Analyze the complexity of σ.

The formula C has k clauses and $n = |V|$ literals per clause. It takes time $O(kn)$ to write down C. The formula D has kn^2 clauses and 2 literals per clause. It takes time $O(kn^2)$ to write down D. The formula E has k^2n^2 clauses and 2 literals per clause. It takes time $O(k^2n^2)$ to write down D. Thus the total time to write down φ is $O(k^2n^2)$.

4 The Class NP

Let A be a decision problem. We say that A is in *deterministic polynomial time* (P) if there exist a polynomial-time computable one-place predicate ψ such that for all instances x of A,

$$x \text{ is a "yes" instance } \Leftrightarrow \psi(x).$$

As a rule of thumb, we generally consider a decision problem to be computationally tractable if it is in P.

We say that A is in *nondeterministic polynomial time* (NP) if there exist a polynomial-time computable two-place predicate $\psi(x, y)$ and $k > 0$ such that for all instances x of A,

$$x \text{ is a "yes" instance } \Leftrightarrow \exists w \; |w| \le |x|^k \wedge \psi(x, w).$$

Thus whether x is a "yes" instance is determined by the existence of a relatively small w that satisfies ψ. The element w is often called a *witness*. The process of deciding whether a given instance x of A is a

"yes" instance amounts to determining whether there exists a suitable witness w. We can think of w as a short proof of the fact that x is a "yes" instance, and of the predicate ψ as a polynomial-time verification procedure to verify that w is indeed a proof of this fact. It is important that the verification procedure ψ be computable in polynomial time and that the witness, if it exists, be polynomial in size as a function of the size of x (hence the k and the bound on the size of w).

The Boolean satisfiability problem is of this form. Here the instances of the problem are Boolean formulas and the witnesses are satisfying assignments. A formula is a "yes" instance iff it is satisfiable, that is, if there exists a satisfying assignment. The predicate ψ in this case would be a verification procedure that takes a given formula and a truth assignment and checks that that truth assignment satisfies the formula, which can be done in polynomial time. Thus SAT is in NP.

Another example is the CLIQUE problem. Here the instances of the problem are pairs (G, k), where $G = (V, E)$ is an undirected graph and $k \geq 0$ is the size of the desired clique. A witness would be a subset $K \subseteq V$ that forms a k-clique. An input to the clique problem (G, k) is a "yes" instance iff such a K exists. The predicate ψ in this case would be a verification procedure that takes G, k, and $K \subseteq V$, and checks that $|K| = k$ and all pairs of nodes of K are adjacent, which can be done in polynomial time. Thus CLIQUE is in NP.

Several other problems we have mentioned in class are also of this form: Independent Set, Vertex Cover, k-Colorability, Traveling Salesperson, Hamiltonian Circuit, Set Cover, Knapsack, and Bin Packing, to name a few.

4.1 Guess and Verify

Problems in NP are sometimes referred to as *guess and verify* problems. Given an input x, we would like to know whether there exists a witness w such that $\psi(x, w)$. Finding the witness is the hard part; once a witness has been found, the verification is easy using ψ. But imagine a procedure that magically guesses a witness w if it exists, then verifies that its guess was correct by running $\psi(x, w)$. If the guessing procedure is very lucky, it will guess the witness if there is one to be guessed. The "nondeterministic" in the name NP refers to this magical guessing. Of course, no such guessing mechanism really exists, and there are typically exponentially many potential witnesses to try. Nevertheless it is convenient to think of NP in those terms.

For example, we can think of a nondeterministic polynomial-time procedure for SAT as guessing a satisfying truth assignment, then verifying deterministically in polynomial time that the guessed assignment indeed satisfies the formula. A nondeterministic polynomial-time procedure for CLIQUE would guess a subset of the vertices of size k, then verify that the guess subset formed a clique.

5 NP-Hardness and NP-Completeness

Reductions are often used to show that a problem B is at least as hard as another problem A. When using reductions in this way, A is our known hard problem, and we want to show that B is at least as hard by showing that $A \leq_{\mathrm{p}} B$.

A decision problem B is said to be NP-*hard* if *every* decision problem in NP reduces to B; that is, for every decision problem $A \in$ NP, $A \leq_{\mathrm{p}} B$. Thus B is as hard as any problem in NP. A decision problem B is said to be NP-*complete* if it is NP-hard and in NP itself. Many decision problems we have discussed (Sat, Clique, Independent Set, Vertex Cover, k-Colorability, Traveling Salesperson, Hamiltonian Circuit, Set Cover, Knapsack, Bin Packing) are NP-complete.

Cook and Levin independently showed that SAT is NP-complete. We will take this result as given for now. Since the reducibility relation $\leq_{\mathrm{p}}$ is transitive, one can show that another problem B is NP-hard by reducing SAT, or any of the other known NP-complete problems, to B. To show that B is NP-complete, one must show in addition that B is in NP.

Thus, to show a problem B is NP-complete, one should do the following:

1. Show that B is in NP-hard by giving a polynomial-time reduction from one of the known NP-complete problems (SAT, CNFSAT, CLIQUE, etc.) to B.
2. Show that B is in NP by giving a nondeterministic polynomial-time guess-and-verify algorithm for it.

If you are only asked to show NP-hardness, you only need to do 1. But if you are asked to show NP-completeness, you need to do both 1 and 2.

For a good sample of most of the NP-complete problems we will talk about and general thoughts on reductions involving them, see K&T §8.10, A Partial Taxonomy of Hard Problems.

To show a problem B is NP-hard, first pick a known NP-complete problem H. We want to show that $H \leq_{\mathrm{p}} B$. From the notation, you should notice that you need to reduce H to B. If you were to reduce B to H then you would be showing that $B \leq_{\mathrm{p}} H$, which says that B is no harder than H, but does not help us show that B is hard.

The reduction takes the form described in §2 involving a polynomial-time transformation σ of instances of H to instances of B. Sometimes the transformation will be straightforward. Other times the reduction will require the construction of intricate gadgets.

To show that B is in NP, you should give a nondeterministic polynomial-time guess-and-verify algorithm for it. This entails specifying a suitable set of witnesses, making sure a witness (if it exists) is not too big, and giving a *deterministic* polynomial-time verifier that, given a candidate for a witness, checks that it is indeed a witness. This step is usually not difficult.

Some other general tips on constructing reductions:

- If a problem asks you to decide if there exists a set of *at least* k objects satisfying some property, try reducing from another problem that involves picking *at least* k objects, e.g. Independent Set or Clique.
- Similarly, if a problem asks you to decide if there exists a set of *at most* k objects satisfying some property, try reducing from another problem that involves picking *at most* k objects, e.g. Vertex Cover or Set Cover.
- When a problem does not easily fit into either of the general categories listed above, usually the best thing to try first is 3CNFSAT.
- Show that your reduction takes polynomial time. This is implicit in the notation $\leq_{\mathrm{p}}$.
- Show that x is a solution to the problem you are reducing from *if and only if* $\sigma(x)$ is a solution to the problem you are trying to show is NP-hard. You need to show the implication in both directions.