Chapter 6

NP Completeness

6.1 Introduction

This chapter studies the boundary between feasible and infeasible computations, that is between problems that can be solved with a reasonable amount of resources, time being the most critical resource, and those problems that require an inordinate amount of time to solve.

In order to be precise, we need to specify what feasibility means. We define feasibility to be Deterministic Polynomial Time, $\mathcal{P}$ for short. This is by contrast with another class, Non-Deterministic Polynomial Time, $\mathcal{NP}$ for short. As we will see, $\mathcal{P} \subseteq \mathcal{NP}$, and further it is an essentially universal belief that $\mathcal{P} \subsetneq \mathcal{NP}$; the hardest problems in $\mathcal{NP} \setminus \mathcal{P}$, and there are many important problems among these, are believed to be infeasible. Before discussing this further we need to define these classes.

We begin by defining running time in terms of the number of steps a program performs. By a step we intend a basic operation such as an addition, a comparison, a branch (due to a goto, in a while loop, in an if-statement, to perform a procedure call, etc.). We do not allow more complex steps, such as $B \leftarrow 0$ where B is an $n \times n$ array, to count as a single step. Finally, we also limit the word size—the contents of a single memory location—to $O(\log n)$ bits, where n is the input size. This implies that repeated squaring of a number would not be a legitimate series of steps, for it would soon result in arithmetic overflow.

More specifically, we define the running time with respect to the programming language defined in Chapter 4, modified as in Chapter 5 to include write instructions. This allows us to have terminating programs which decide a language (an output of 1 corresponds to recognition of an input, and an output of 0 to rejection). It also allows programs that compute functions.

Definition 6.1.1. *A program or algorithm has worst case running time $T(n)$, running time $T(n)$ for short, if for all inputs of size n it completes its computation in at most $T(n)$ steps. (Strictly, in addition, on some input of size n, the full $T(n)$ steps are performed.)*

6.1.1 The Class $\mathcal{P}$

Definition 6.1.2. *An algorithm runs in polynomial time if its running time $T(n)$ is bounded by a polynomial function of n.*

Definition 6.1.3. *A problem or language is in $\mathcal{P}$ (polynomial time) if there is an algorithm solving the problem or deciding the language that runs in polynomial time.*

We view polynomial time as corresponding to the class of problems having efficient algorithms. Of course, if the best algorithm for a problem ran in time n^{100} this would not be efficient in any practical sense. However, in practice, the polynomial time algorithms we find tend to have modest running times such as $O(n), O(n \log n), O(n^2), O(n^3)$.

One reason for defining $\mathcal{P}$ as the class of efficient algorithms is that there is no principled way of partitioning among polynomial time algorithms. Is $\Theta(n^3)$ OK but $\Theta(n^{3.5})$ too slow? Then what about $\Theta(n^{3.1})$, etc.? A second reason is that the class $\mathcal{P}$ is closed under composition: if the output of one polynomial time algorithm is the input to a second polynomial time algorithm, then the overall combined algorithm also runs in time polynomial in the size of the original input.

Another interesting way of viewing polynomial time algorithms is that a moderate increase in resources suffices to double the size of the problems that can be solved. For example, given that algorithm $\mathcal{A}$ can solve a problem of size n in one hour on a particular computer, if $\mathcal{A}$ runs in linear time, then given a new computer that runs twice as fast, in the one hour one could solve twice as large a problem with algorithm $\mathcal{A}$. Running twice as fast means that it performs twice as many operations in the given time. While if $\mathcal{A}$ runs in quadratic time ($\Theta(n^2)$), then give a new computer that runs four times as fast, in one hour one could solve a problem that is twice as large; again, if $\mathcal{A}$ runs in cubic time ($\Theta(n^3)$), then given a new computer that runs eight times as fast, in one hour one could solve twice as large a problem; and so forth.

This contrasts with the effect of an exponential running time ($\Theta(2^n)$, for instance). Here doubling the speed of the computer increases the size of the problem that can be solved in one hour from k to $k+1$, where k was the previously solvable size. As a result, exponential time algorithms are generally considered to be infeasible.

6.1.2 Polynomial Time Defined via Turing Machines

We could also define running time in general, and polynomial time in particular in terms of the number of steps performed by a Turing Machine computation. It will turn out that these two definitions (w.r.t. our programming language and w.r.t. Turing Machines) are equivalent in the sense that they define the same class of problems.

Definition 6.1.4. *A Turing Machine runs in polynomial time if its running time $T(n)$ on an input of length n, i.e. an input that occupies the first n cells of its tape, is bounded by a polynomial function of n.*

Definition 6.1.5 (Turing Machine version)**.** *A problem or language is in $\mathcal{P}$ (polynomial time) if there is a Turing Machine solving the problem or deciding the language that runs in polynomial time.*

We now prove the equivalence of the two definitions of $\mathcal{P}$.

Lemma 6.1.6. *Let Q be a program on w-bit words that runs in time T on an input x. Then there is a Turing Machine simulation of Q which runs in time $O(w^2 \cdot (T + |x|)^2 \cdot T)$.*

Proof. We will use the simulation given in Section 4.2, with two small but important modifications. We begin by analyzing a simulation of Q on the 6-tape Turing Machine M described there. We

note that in T steps, Q can access at most $3T$ distinct memory locations. Accordingly, we will store on one of M's tapes the contents of up to $3T$ of Q's memory locations (strictly speaking, the memory of the machine on which Q executes). We call this the *memory tape*. For each of these memory locations, M stores a pair in the form (location, value). These need not be contiguous locations (this is the first modification). Each pair will use $O(w)$ bits and hence $O(w)$ cells on its memory tape.

The second modification concerns M's input. As the input need not be written in contiguous locations of Q's memory, we will assume that M is provided the relevant memory locations as its input in the form of a series of pairs of the form (location, value). We will copy this input to the memory tape. Altogether, therefore, storing P's memory will use $O(w \cdot (T + |x|))$ cells.

Each step of the simulation requires up to three sweeps across M's memory tape, to retrieve and store the up to three variables that are used by the instruction being simulated. A further $O(w)$ steps are needed to perform the instruction. Three tapes are used for these three variables.

Tape 6 is used to hold Q's instruction counter.

Thus M simulates one step of Q's computation using at most $O(w \cdot (T + |x|))$ of its steps, and therefore it simulates Q's T-step computation using $O(w \cdot (T + |x|) \cdot T)$ of its steps, i.e. in this time.

We now simulate this 6-tape Turing Machine M using the 1-tape Machine $\widetilde{M}$ described in the proof of Lemma 4.1.3. Recall that simulating one step of M requires one sweep forward and back across $\widetilde{M}$'s tape, and takes a number of steps equal to the length of tape traversed by $\widetilde{M}$. Also recall that if M uses C cells then so does $\widetilde{M}$. Next, note that M uses $O(w \cdot (T + |x|))$ cells. Thus $\widetilde{M}$ simulates one of M's steps in $O(w \cdot (T + |x|))$ steps. Overall, $\widetilde{M}$ simulates Q's computation using $O(w^2 \cdot (T + |x|)^2 \cdot T)$ steps. □

Theorem 6.1.7. *If Q is a program that runs in polynomial time, then the Turing Machine simulating Q described in Lemma 6.1.6 also runs in polynomial time.*

Proof. We make the standard assumption that a polynomial time algorithm uses a memory of polynomial size, and that it uses words of size $c \log n$ for some constant $c \geq 1$. Let n be the size of the input. Then, in Lemma 6.1.6 set $w = c \log n$ and $T = p(n)$, where $p(n)$ is the polynomial bounding the running time of Q. Then by Lemma 6.1.6, the simulating Turing Machine runs in time $O(\log^2 n \cdot (n + p(n))^2 \cdot p(n)) = O(n \cdot (n + p(n))^3)$, which is also a polynomial bound. □

It is easy to show the complementary result, namely given a Turing Machine M that runs in polynomial time, there is a program P that simulates M and also runs in polynomial time.

6.1.3 The Class $\mathcal{NP}$

This class of languages is characterized by being "verifiable" in polynomial time. We begin with some examples.

Example 6.1.8. Hamiltonian Cycle.
Input: A directed graph $G = (V, E)$.

Question: Does G have a Hamiltonian Cycle, that is a cycle that goes through each vertex in V exactly once?

Output: "Yes" or "no" as appropriate.

The "Yes" (or "Recognize") answer is polynomial time verifiable in the following sense. Given a sequence of $n = |V|$ vertices which is claimed to form a Hamiltonian Cycle this is readily checked in linear time (it suffices to check that each vertex appears exactly once in the list and that if the list is the sequence $v_1, v_2, \cdots, v_n$, then $(v_i, v_{(i+1) \mod n}) \in E$ for each $i, 1 \leq i \leq n$.)

If this sequence of vertices forms a Hamiltonian Cycle, as claimed, it is called a *certificate*.

By definition, a Hamiltonian graph has a certificate, namely the list of vertices forming (one of) its Hamiltonian Cycle(s). On the other hand, if the graph is not Hamiltonian, i.e. it does not have a Hamiltonian Cycle, then no attempted certificate will check out, for no proposed sequence of vertices will form a Hamiltonian Cycle. However, it might be that you are given a proposed certificate for a Hamiltonian graph which fails because it is not a Hamiltonian Cycle. This failure does not show anything regarding whether the graph is Hamiltonian; it only shows that the sequence of vertices did not form a Hamiltonian Cycle for this particular graph. You cannot conclude whether or not the graph is Hamiltonian from this.

Example 6.1.9. Clique.
Input: (G, k) where G is an undirected graph and k an integer.

Question: Does G have a clique of size k? A clique is a subset of vertices such that every pair of vertices in the subset is joined by an edge.

Output: "Yes" or "no" as appropriate.

Again the "Yes" (or "Recognize") answer is polynomial time verifiable, given the following additional information, or *certificate*: a list of k vertices which are claimed to form a clique. To verify this it suffices to check that the list comprises k distinct vertices and that each pair of vertices in this list are joined by an edge. This is readily done in $O(k^2 n)$ time, and indeed in $O(n^3)$ time (for if $k > n = |V|$, then the graph does not have a k-clique).

Again, if the graph does not have a k-clique, then any set of k vertices will fail to be fully connected; there will be a pair of vertices in the given k-vertex set with no edge between them. So any proposed certificate will fail to demonstrate that the graph has a k-clique.

However, just because you are given a set of k-vertices that does not happen to form a k-clique, you cannot then conclude that the graph has no k-clique.

Example 6.1.10. Satisfiability.
Input: F, a CNF Boolean formula.

F is in CNF form if $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause $C_i, 1 \leq i \leq m$, is an 'or' of literals:

$$C_i = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ij_i},$$

where each l_{ik} is a Boolean variable or its complement (negation).
e.g. $F_1 = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_3 \vee x_2)$; $F_2 = x$; $F_3 = x \wedge \overline{x}$.

Question: Is F satisfiable? That is, is there an assignment of truth values to F's variables that causes F to evaluate to TRUE?
e.g. For F_1, setting $x_1 =$ TRUE, $x_2 =$ FALSE, $x_3 =$ TRUE, causes F_1 to evaluate to TRUE; For F_2, setting $x =$ TRUE causes F_2 to evaluate to TRUE; for F_3, no setting of the variables will cause F_3 to evaluate to TRUE.

Here, the additional information, or certificate, is the assignment of truth values to the formula's variables that causes the formula to evaluate to TRUE. Notice that if the formula never evaluates to TRUE, then any proposed truth assignment for the Boolean variables will cause the formula to evaluate to FALSE. In other words, any proposed certificate fails.

Again, just because you are given a truth assignment that cause the formula to evaluate to FALSE, you cannot then conclude that the formula is not satisfiable. All you know is that this particular collection of truth assignments to the Boolean variables did not work (i.e. they caused the Formula to evaluate to FALSE in this case).

Definition 6.1.11. *An assignment σ for the variables of a Boolean formula F is said to be* satisfying *if it causes the formula to evaluate to* TRUE.

Definition 6.1.12. *A language $L \in \mathcal{NP}$ if there are polynomials p, q, and an algorithm V called the verifier, such that:*

- *If $x \in L$ there is a* certificate *$C(x)$, of length at most $p(|x|)$, such that $V(x, C(x))$ outputs "Yes".*
- *While if $x \notin L$, for every string y of length at most $p(|x|)$, $V(x, y)$ outputs "No".*

In addition, $V(x, y)$ runs in time $q(|x| + |y|) = O(q(n + p(n)))$, where $|x| = n$ and $q(\cdot)$ is a polynomial.

Comment. If $x \notin L$, there is no certificate for x; in other words, any string claiming to be a certificate is readily exposed as a non-certificate.

However, a non-certificate does not determine whether $x \in L$ or $x \notin L$.

6.1.4 Discussion

It is not known whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \subsetneq \mathcal{NP}$ ($\mathcal{P} \subseteq \mathcal{NP}$ as every problem Q in $\mathcal{P}$ has a polynomial time verifier: the verifier uses the empty string as the certificate and runs the polynomial time recognizer for Q).

However, due to the lack of success in finding efficient algorithms for problems in $\mathcal{NP}$, and in particular for the so-called $\mathcal{NP}$-Complete problems, or $\mathcal{NPC}$ problems for short, it is essentially universally believed that $\mathcal{P} \neq \mathcal{NP}$. $\mathcal{NPC}$ problems are the hardest problems in the class $\mathcal{NP}$ and they have the interesting feature that if a polynomial time algorithm is found for even one of these problems, then they will all have polynomial time algorithms. Given that many of these problems have significant practical importance, considerable effort has gone into attempting to find efficient algorithms for these problems. The lack of success of this effort has reinforced the belief that there are no efficient algorithms for these problems. In some sense, such an efficient algorithm would provide an efficient implementation of the "process" of inspired "guessing," which strikes many as implausible.

Factoring is an example of a problem in $\mathcal{NP}$ for which no efficient algorithm is known. By factoring, we mean the problem of reporting a non-trivial factor of a composite number. As it happens, there is a polynomial time algorithm to determine whether a number is composite or prime, but this algorithm does not reveal any factors in the case of a composite number. Indeed, the assumed hardness of the factoring problem underlies the security of the RSA public key scheme, which at present is a key element of the security of much of e-commerce and Internet communication.

It is also worth noting that factoring is not believed to be an $\mathcal{NPC}$ problem, that is it is not believed to be a "hardest" problem in the class $\mathcal{NP}$.

However, in the end, none of the above evidence demonstrates the hardness of $\mathcal{NPC}$ problems. Rather, it indicates why people believe these problems to be hard.

6.2 Reductions Between $\mathcal{NP}$ Problems

As with undecidable problems, the tool to show problems are $\mathcal{NPC}$ problems is a suitable type of reduction. The basic form is the following. Given a polynomial time decision or membership algorithm for language A we use it as a subroutine to give a polynomial time algorithm for problem B. Recall that a decision algorithm reports "Recognize" ("Yes") or "Reject" ("No") as appropriate. This is called a *Polynomial Time Reduction*.

Our goal is to show reductions among all $\mathcal{NPC}$ problems, for this then leads to the conclusion that if one of them can be solved by a polynomial time algorithm, then they all can be solved in polynomial time. We begin with several examples of such reductions.

6.2.1 Independent Set and Clique

Example 6.2.1. Independent Set.
Input: Undirected Graph G, integer k.

Question: Does G have an independent set of size k, that is a subset of k vertices with no edges between them?

Claim 6.2.2. *Given a polynomial time algorithm for Clique there is a polynomial time algorithm for Independent Set.*

Proof. Let $G = (V, E)$ be the input to the Independent Set problem. Consider the graph $\overline{G} = (V, \overline{E})$. We note that $S \subseteq V$ is an independent set in G if and only if it is a clique in $\overline{G}$. (For if a subset of k vertices have no edges among themselves in graph G, then in graph $\overline{G}$ all possible edges between them are present, i.e. the k vertices form a k-clique in $\overline{G}$. The converse is true also: a set of k vertices forming a k-clique in $\overline{G}$ also form an independent set in G. An example is shown in Figure 6.1.

Thus the Independent Set algorithm is simply to run the Clique algorithm on the pair $(\overline{G}, k)$, and report its result. □

Terminology We will use the name of a problem to indicate the set of objects satisfying the problem condition. Thus Indpt-Set $= \{(G, k) \mid G$ has an independent set of size $k\}$.

The algorithm $\mathcal{A}_{\text{IS}}$, for Independent Set, performs 3 steps.

1. On input (G, k), compute the input $(\overline{G}, k)$ for the Clique algorithm, $\mathcal{A}_{\text{clique}}$.
2. Run $\mathcal{A}_{\text{Clique}}(\overline{G}, k)$.
3. Use the answer from Step 2 to determine its own answer (the same answer is this case).

Demonstrating the algorithm is correct also entails showing:

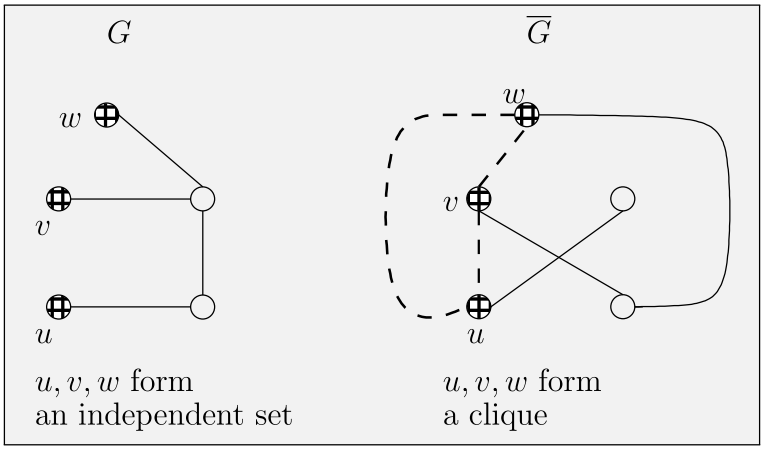

Figure 6.1: Example: An Independent Set in G forms a clique in $\overline{G}$.

4. $\mathcal{A}_{\text{IS}}$'s answer is correct. In this case that means showing that

$$(G, k) \in \text{Indpt-Set} \iff (\overline{G}, k) \in \text{Clique}.$$

   This was argued in the proof of Claim 6.2.2.

   Often, we will ague Step 4 right after Step 1 in order to explain why the constructed input makes sense.

5. Assuming that $\mathcal{A}_{\text{Clique}}$ runs in polynomial time, then showing that $\mathcal{A}_{\text{IS}}$ also runs in polynomial time.
   This is straightforward. For $(\overline{G}, k)$ can be computed in $O(n^2)$ time, so Step 1 runs in polynomial time. For Step 2, as $\mathcal{A}_{\text{Clique}}$ receives an input of size $O(n^2)$ and as $\mathcal{A}_{\text{Clique}}$ runs in polynomial time by assumption, Step 2 also runs in polynomial time. Finally, Step 3 takes $O(1)$ time, so the whole algorithm runs in time polynomial in the size of its input.

   As a rule, we will not spell out these arguments in such detail as they are straightforward.

The next claim is similar and is left to the reader.

Claim 6.2.3. *Given a polynomial time algorithm for Independent Set there is a polynomial time algorithm for Clique.*

6.2.2 Hamiltonian Path and Cycle

Example 6.2.4. Hamiltonian Path (HP).
Input: (G, s, t), where $G = (V, E)$ is a directed graph and $s, t \in V$.
Question: Does G have a Hamiltonian Path from s to t, that is a path that goes through each vertex exactly once?

Claim 6.2.5. *Given a polynomial time algorithm for Hamiltonian Cycle (HC) there is a polynomial time algorithm for Hamiltonian Path.*

Proof. Let (G, s, t) be the input to the Hamiltonian Path Problem. $\mathcal{A}_{\text{HP}}$, the algorithm for the Hamiltonian Path problem, proceeds as follows.

1. Build a graph H with the property that H has a Hamiltonian Cycle exactly if G has a Hamiltonian Path from s to t.
2. Run $\mathcal{A}_{\text{HC}}(H)$.
3. Report the answer given by $\mathcal{A}_{\text{HC}}(H)$.

H consists of G plus one new vertex, z say, together with new edges $(t, z), (z, s)$.

4. We argue that G has a Hamiltonian Path from s to t exactly if H has a Hamiltonian Cycle. For if H has a Hamiltonian Cycle it includes edges $(z, s), (t, z)$ as these are the only edges incident on z; we write $s = v_1$ and $t = v_n$. So the cycle has the form $z, v_1, v_2, \cdots, v_n$, and then $v_1, v_2, \cdots, v_n$ is the corresponding Hamiltonian Path in G. Conversely, if G has Hamiltonian Path $v'_1, v'_2, \cdots, v'_n$, where $s = v'_1$ and $t = v'_n$, then H has Hamiltonian Cycle $z, v'_1, v'_2, \cdots, v'_n$. An example of this construction is shown in Figure 6.2.
5. Clearly $\mathcal{A}_{\text{HP}}$ runs in polynomial time if $\mathcal{A}_{\text{HC}}$ runs in polynomial time.

□

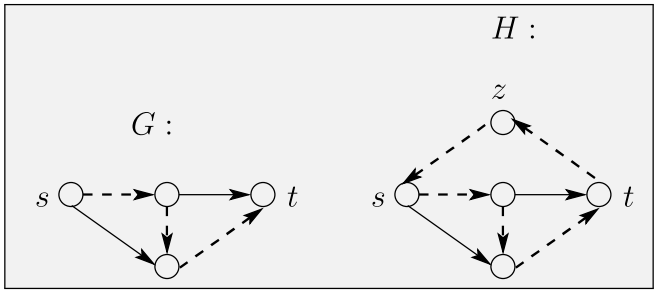

Figure 6.2: G has a Hamiltonian Path from s to t iff H has a Hamiltonian Cycle. A corresponding path and cycle are shown as dashed edges.

Example 6.2.6. Degree d Bounded Spanning Tree (DBST).
Input: (G, d), where G is an undirected graph G and d is an integer.
Question: Does G have a spanning tree T such that in T each vertex has degree at most d?

Example 6.2.7. Undirected Hamiltonian Path (UHC).
Input: (G, s, t), where $G = (V, E)$ is an undirected graph, and $s, t \in V$.
Question: Does G have a Hamiltonian Path from s to t, that is a path going through each vertex exactly once?

Claim 6.2.8. *Given a polynomial time algorithm for Degree d Bounded Spanning Tree there is a polynomial time algorithm for Undirected Hamiltonian Path.*

Proof. Let (G, s, t) be the input to the Hamiltonian Path Problem. $\mathcal{A}_{\text{UHP}}$, the algorithm for the Hamiltonian Path problem, proceeds as follows.

1. Build a graph H with the property that H has a degree 3 bounded spanning tree exactly if G has a Hamiltonian Path from s to t.
2. Run $\mathcal{A}_{\text{DBST}}(H, 3)$.
3. Report the answer given by $\mathcal{A}_{\text{DBST}}(H, 3)$.

H is the following graph: G plus vertices v' for each $v \in V$, plus vertices s'', t'', together with edges (v', v) for each $v \in V$ and edges $(s'', s), (t'', t)$. An example is shown in Figure 6.3.

4. We argue that G has a Hamiltonian Path from s to t exactly if H has a spanning tree with degree bound 3. Note that all the new vertices in H have degree 1 and consequently all the new edges must be in any spanning tree T of H. Removing these edges and the new vertices leaves a tree in which each vertex has degree at most 2, and in which s and t both have degree at most 1. As the resulting graph is still connected, this means that the remaining edges in T form a Hamiltonian Path in G from s to t.
5. Clearly $\mathcal{A}_{\text{HP}}$ runs in polynomial time if $\mathcal{A}_{\text{DBST}}$ runs in polynomial time.

□

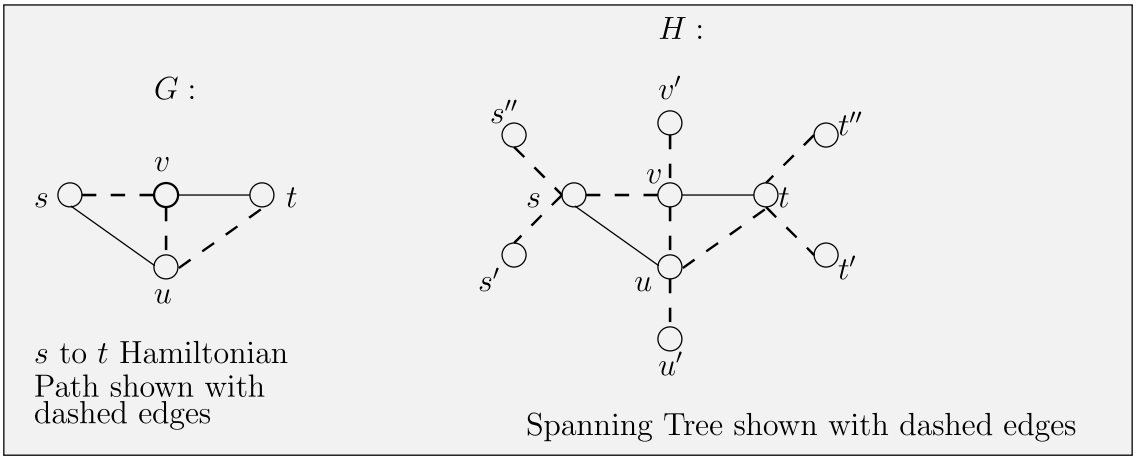

Figure 6.3: G has a Hamiltonian Path from s to t if and only if H has a Degree 3 Bounded Spanning Tree.

6.3 The Structure of Reductions

We observe that all these constructions have the following form: given a polynomial time algorithm $\mathcal{A}_B$ for membership in B we construct a polynomial time algorithm $\mathcal{A}_A$ for membership in A as follows.

Input to $\mathcal{A}_A$: I.

1. Construct $f_{A\leq B}(I)$ in polynomial time.
2. Run $\mathcal{A}_B(f_{A\leq B}(I))$.
3. Report the answer from Step 2.

If there is such a function $f_{A\leq B}$, we write $A \leq_P B$, or $A \leq B$ for short. It is read as: A can be reduced to B in polynomial time. We call the function $f_{A\leq B}$ the reduction.
In order for Step 3 to be correct we need that:

$$I \in A \iff f_{A\leq B}(I) \in B.$$

As $f_{A\leq B}(I)$ is computed in polynomial time it produces a polynomial sized output. Thus $\mathcal{A}_B(f_{A\leq B}(I))$ runs in time polynomial in $|I|$, as $\mathcal{A}_B$ runs in polynomial time. This uses the fact that if p and q are bounded degree polynomials, then so is $p(q(\cdot))$.

This is called a polynomial time reduction of problem A to problem B. It shows that if there is a polynomial time membership (decision) algorithm for B then there is also a polynomial time membership algorithm for A. The converse is true also:

> If there is no polynomial time membership algorithm for A then there is not one for B either.

6.4 More Examples of Reductions

6.4.1 Undirected Hamiltonian Path and Cycle

Example 6.4.1. *Undirected Hamiltonian Path* (UHP) and *Undirected Hamiltonian Cycle* (UHC) are the same problems as the corresponding problems for directed graphs, but the new problems are for undirected graphs.

Claim 6.4.2. *Given a polynomial time algorithm for* Undirected Hamiltonian Path (UHP) *there is a polynomial time algorithm for* Directed Hamiltonian Path.

Proof. Let (G, s, t) be the input to the Directed Hamiltonian Path algorithm, $\mathcal{A}_{\text{DHP}}$. The algorithm proceeds as follows.

1. Build the triple (H, p, q), where H is an undirected graph, and p and q are vertices in H with the property that
$$(G, s, t) \in \text{DHP} \iff (H, p, q) \in \text{UHP}.$$
2. Run $\mathcal{A}_{\text{UHP}}(H, p, q)$.
3. Report the answer given by $\mathcal{A}_{\text{UHP}}(H, p, q)$.

[itemsep=1pt] Now, we need to describe the construction and observe its correctness.

Let $G = (V, E)$ and $H = (F, W)$. Let $n = |V|$. (H, p, q) is constructed as follows. For each vertex $v \in V$, H has 3 vertices $v^{\text{in}}, v^{\text{mid}}, v^{\text{out}}$, plus the edges $(v^{\text{in}}, v^{\text{mid}}), (v^{\text{mid}}, v^{\text{out}})$. And for each edge $(v, w) \in E$, H receives edge $(v^{\text{out}}, w^{\text{in}})$. See Figure 6.4 for an illustration. Then $\mathcal{A}_{\text{DHP}}$ sets $p = s^{\text{in}}$ and $q = t^{\text{out}}$.

An example of the reduction is shown in Figure 6.5.

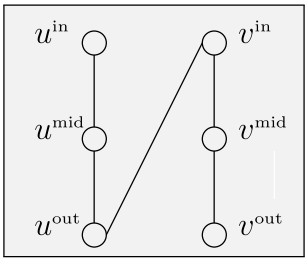

Figure 6.4: Edges in H corresponding to edge (u, v) in G

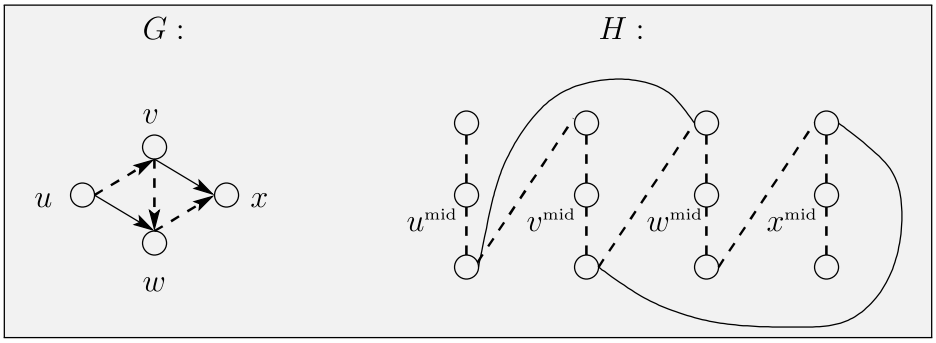

Figure 6.5: An example of the reduction from HP to UHP.

4. We argue that G has a Hamiltonian Path from s to t exactly if H has a Hamiltonian Path from p to q.

   First, suppose that G has a Hamiltonian Path from s to t. Suppose that the path is $u_1, u_2, \cdots, u_n$, where $s = u_1, t = u_n$, and $u_1, u_2, \cdots, u_n$ is a permutation of the vertices in V. Then the following path is a Hamiltonian Path from p to q in H.

   $$u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \cdots, u_n^{\text{in}}, u_n^{\text{mid}}, u_n^{\text{out}}.$$

   Next, suppose that H has a Hamiltonian Path P from p to q. For P to go through vertex v^{mid}, P must include edges $(v^{\text{in}}, v^{\text{mid}})$ and $(v^{\text{mid}}, v^{\text{out}})$. Thus the path has the form

   $$v_1^{\text{in}}, v_1^{\text{mid}}, v_1^{\text{out}}, v_2^{\text{in}}, v_2^{\text{mid}}, v_2^{\text{out}}, \cdots, v_n^{\text{in}}, v_n^{\text{mid}}, v_n^{\text{out}},$$

   where $|V| = n$, $s = v_1$, $t = v_n$, and $v_1, v_2, \cdots, v_n$ is a permutation of the vertices in V. Then $v_1, v_2, \cdots, v_n$ is a Hamiltonian Path from s to t in G.

   This shows the claim.

5. Clearly $\mathcal{A}_{\text{DHP}}$ runs in polynomial time if $\mathcal{A}_{\text{UHP}}$ runs in polynomial time.

□

6.4.2 Vertex Cover, Independent Set, and 3-SAT

Example 6.4.3. Vertex Cover (VC).
Input: (G, k), where $G = (V, E)$ is an undirected graph and k is an integer.
Question: Does G have a vertex cover of size k, that is a subset $U \subseteq V$ of size at most k with the property that every edge in E has at least one endpoint in U?

Claim 6.4.4. *Given a polynomial time algorithm for* Independent Set (IS) *there is a polynomial time algorithm for* Vertex Cover.

Proof. This is immediate from the following observation. If I is an independent set of G then $V - I$ is a vertex cover, and conversely. See Figure 6.6 for an example.

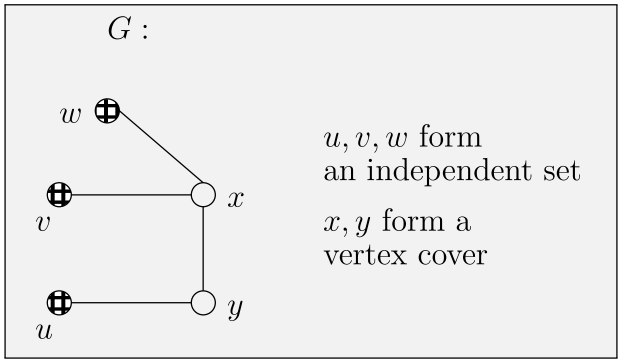

Figure 6.6: Example: The complement of an Independent Set is a Vertex Cover.

For if I is an independent set, then as no edge joins pairs of vertices in I, any edge has at most one endpoint in I, and hence at least one endpoint in $V - I$. Conversely, if $V - I$ is a vertex cover, then any edge has at least one endpoint in $V - I$, and so there is no edge with two endpoints in I, meaning that I is an independent set.

We can conclude that $(G, k) \in \text{VC} \iff (G, |V| - k) \in \text{IS}$.

We leave the details of creating the algorithm for Vertex Cover to the reader. □

Example 6.4.5. 3-SAT.
Input: F, a CNF Boolean Formula in which each clause has at most 3 variables.
Question: Does F have a satisfying assignment (of Boolean values to its variables)?

Claim 6.4.6. *Given a polynomial time algorithm for* Independent Set (IS) *there is a polynomial time algorithm for* 3-SAT.

Proof. Let F be the input to the 3-SAT algorithm, $\mathcal{A}_{\text{3-SAT}}$. The algorithm proceeds as follows.

1. $\mathcal{A}_{\text{3-SAT}}$ computes (G, k), where G is a graph and k is an integer, with the property that

$$F \in \text{3-SAT} \iff (G, k) \in \text{IS}.$$

2. It runs $\mathcal{A}_{\text{IS}}(G, k)$.
3. It reports the answer given by $\mathcal{A}_{\text{IS}}(G, k)$.

Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_\ell$, and suppose that F has variables $x_1, x_2, \cdots, x_n$. G receives the following vertices. For each clause $C_j = a \vee b \vee c$ it has vertices u_a^j, u_b^j, u_c^j connected in a triangle (if the clause is $C_j = a \vee b$ it has vertices u_a^j, u_b^j joined by an edge, and if $C_j = a$ it has the vertex u_a^j alone). In addition, for every pair j, k of distinct clauses, for each variable x appearing in C_j as x and in C_k as $\overline{x}$, the pair of vertices u_x^j and $u_{\overline{x}}^k$ are joined together. See Figure 6.7 for an example. Finally, $\mathcal{A}_{\text{3-SAT}}$ sets $k = \ell$.

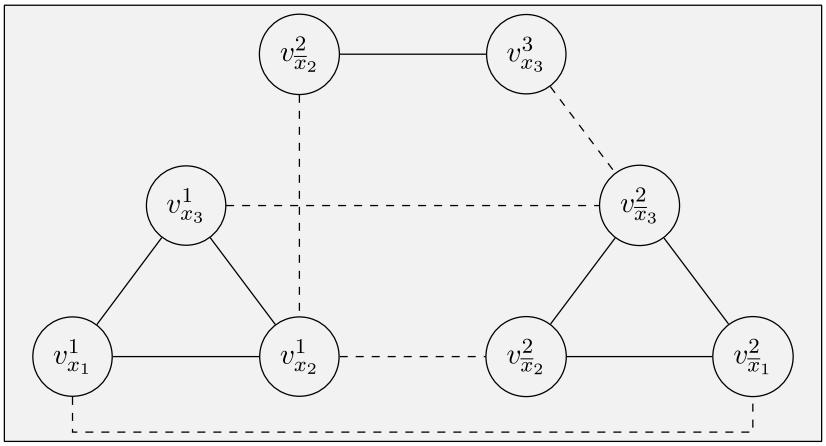

Figure 6.7: The graph for the Independent Set construction for $F = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3)$. The edges joining nodes in different triangles are shown as dashed lines for greater visibility.

4. We argue that F has a satisfying assignment exactly if G has an independent set I of size ℓ.

   Suppose that F has a satisfying assignment σ. For each clause C_j, we identify one literal l_{ij} that has value TRUE in assignment σ. We add the corresponding vertex $v_{l_{ij}}^j$ to I. This puts ℓ vertices in I. We observe that there are no edges among the vertices added to I for the following two reasons: first, only one vertex per triangle is added. Second, for each variable x, if it is set to TRUE, then only vertices corresponding to the literal x are added to I, and there are no edges between these vertices; similarly, if x is set to FALSE, the added vertices correspond to the literal $\overline{x}$ and again there are no edges among these vertices. Consequently, the vertices in I form an independent set of ℓ vertices.

   Conversely, suppose that G has an independent set I of size ℓ. We describe a satisfying assignment σ for F. For each clause $C_j = a \vee b \vee c$, I can include at most one of u_a^j, u_b^j, u_c^j as these three vertices are all connected (and similarly for a clause of two or one literals). This provides at most ℓ vertices. As no other vertices are available, I must include one vertex for each clause.

   We set the variable truth values to match the vertices in I. If a vertex v_x is in I, then x is set to TRUE, and if a vertex $v_{\overline{x}}$ is in I, then x is set to FALSE. Note that both cannot occur as

there would then be an edge between the relevant vertices. If any variables x remain unset, they can be set to TRUE. Note that in each clause at least one literal has been set to TRUE and therefore the formula evaluates to TRUE.

This shows the claim.

5. Clearly $\mathcal{A}_{\text{3-SAT}}$ runs in polynomial time if $\mathcal{A}_{\text{IS}}$ runs in polynomial time.

□

Next, we show a reduction from IS to SAT. This introduces the technique of expressing NP problems in terms of Boolean formulas.

Claim 6.4.7. *Given a polynomial time algorithm for* SAT *there is a polynomial time algorithm for* Independent Set (IS).

Proof. Let (G, k) be the input to IS where $G = (V, E)$. Suppose $V = \{v_1, v_2, \ldots, v_n\}$. We will construct a CNF Boolean formula F such that $F \in$ SAT if and only if $(G, k) \in$ IS.

F will have kn variables x_i^j, for $1 \leq i \leq n$ and $1 \leq j \leq k$. We think of $x_i^j =$ TRUE as indicating that v_i is the jth vextex in the independent set (the order of the vertices in an independent set can be chosen arbitrarily). More precisely, if I is a size k independent set for G, containing vertices $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$, then the variables $x_{h_j}^j$ will be set to TRUE, for $1 \leq j \leq k$, and all the other variables will be set to FALSE.

Now we specify the clauses needed to enforce this meaning for the variables x_i^j. First, we require that for each j, exactly one of the x_i^j be set to TRUE; this corresponds to exactly one vertex being chosen as the jth vertex in I. To this end, we use two collections of clauses.

$$F_{j1} = (x_1^j \vee x_2^j \vee \ldots \vee x_n^j).$$

If satisfied, this ensures at least one of the x_i^j is set to TRUE.

$$F_{j2} = \wedge_{1 \leq h < i \leq n} (\overline{x}_h^j \vee \overline{x}_i^j).$$

If satisfied, the clause for h and i ensures that at most one of x_h^j and x_i^j is set to TRUE. Together, if satisfied, these two collections of clauses ensure exactly one of the x_i^j is set to TRUE.

Second, we require that the chosen truth values correspond to the jth and ℓth vertices in I being distinct, for $1 \leq j < \ell \leq k$. This follows if, for each i, at most one of the variables $x_i^1, x_i^2, \ldots, x_i^k$ is set to TRUE. To this end, we use the clauses

$$F_3 = \wedge_{1 \leq i \leq n} \wedge_{1 \leq j < \ell \leq k} (\overline{x}_i^j \vee \overline{x}_i^\ell).$$

The clause $\overline{x}_i^j \vee \overline{x}_i^\ell$ ensures that at most one of x_i^j and x_i^ℓ is set to TRUE for $j \neq \ell$.

Third, we ensure that the chosen truth values correspond to there being no edges between the vertices in I. This follows if, for each edge (v_h, v_i), at most one of x_h^j and x_i^ℓ is set to TRUE, for $j \neq \ell$ and $1 \leq j, \ell \leq k$. To this end, we use the clauses

$$F_4 = \wedge_{(v_h, v_i) \in E} \wedge_{1 \leq j, \ell \leq k, j \neq \ell} (\overline{x}_h^j \vee \overline{x}_i^\ell).$$

The final formula F is given by

$$F = \wedge_{1 \leq j \leq k} F_{j1} \wedge_{1 \leq j \leq k} F_{j2} \wedge F_3 \wedge F_4.$$

Clearly, F can be built in time $O(n^3)$, where $n = |V|$.

Suppose G has a size k independent set containing vertices $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$. Then, as specified above, the variables $x^j_{h_j}$ aree set to TRUE, for $1 \leq j \leq k$, and all the other variables are set to FALSE. The just described construction ensures F evaluates to TRUE.

Conversely, suppose that F evaluates to TRUE. For each variable $x^j_i =$ TRUE, we add v_i to the independent set I. The construction of F ensures that exactly k variables are set to TRUE, at most one for each value of i, and exactly one for each value of k. Thus k distinct vertices are added to I. Furthermore, F_4 ensures there are no edges among these vertices. Thus the set I is indeed a size k independent set.

□

6.4.3 3-SAT, SAT, and G-SAT

Example 6.4.8. G-SAT.
Input: F, a Boolean Formula.

Question: Does F have a satisfying assignment (of Boolean values to its variables), that is, an assignment that causes it to evaluate to TRUE?

Claim 6.4.9. *Given a polynomial time algorithm for* 3-Satisfiability (3-SAT) *there is a polynomial time algorithm for* G-SAT.

Proof. Let F be the input to the G-SAT algorithm, $\mathcal{A}_{\text{G-SAT}}$. The algorithm proceeds as follows.

1. $\mathcal{A}_{\text{G-SAT}}$ computes E, where E is a 3-CNF formula, with the property that
$$F \in \text{G-SAT} \iff E \in \text{3-SAT}.$$
2. It runs $\mathcal{A}_{\text{3-SAT}}(E)$.
3. It reports the answer given by $\mathcal{A}_{\text{3-SAT}}(E)$.

$\mathcal{A}_{\text{G-SAT}}$ computes a CNF formula E such that E is satisfiable if and only if D is satisfiable. To help understand the process, let's view D as a binary expression tree. E receives a new variable for each internal node in the expression tree. Note that the leaves retain their literal labels (either x or $\overline{x}$). Let r be the variable at the root. The variables labeling the internal nodes are intended to take on the values the corresponding nodes would take on when evaluating the expression tree. An example is shown in Figure 6.8.

For a parent node with operator $\neg$ and corresponding variable z, with child having corresponding variable y, several clauses are added to E, clauses that evaluate to TRUE exactly if $z = \overline{y}$, which is the value z should take on. The following clauses suffice:

$$B = (\overline{z} \vee y) \wedge (z \vee \overline{y}).$$

For if B evaluates to TRUE (as is needed for E to evaluate to TRUE) then one of the following two situations applies:

- z is set to TRUE; then $\overline{z}$ is FALSE and y must be set to TRUE.

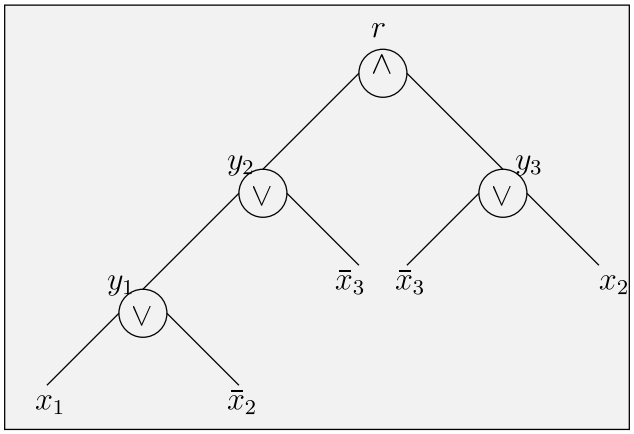

Figure 6.8: Expression Tree Representation of $D = [(x_1 \vee \overline{x}_2) \vee \overline{x}_3] \wedge (\overline{x}_3 \vee x_2)$.

- z is set to FALSE; then $\overline{y}$ must evaluate to TRUE, i.e. y is set to FALSE.

In sum, $z = \overline{x}$.

Similarly, for a parent node with operator $\vee$ and corresponding variable z, with children having corresponding variables x and y, the following clauses are added to E:

$$B = (\overline{z} \vee x \vee y) \wedge (z \vee \overline{x}) \wedge (z \vee \overline{y}).$$

For if B evaluates to TRUE then one of the following two situations applies:

- z is set to TRUE; then $\overline{z}$ is FALSE and $x \vee y$ must evaluate to TRUE.
- z is set to FALSE; then both $\overline{x}$ and $\overline{y}$ must be set to TRUE, and consequently $x \vee y$ evaluates to FALSE.

In sum, $z = x \vee y$.

Similarly, for a parent node with operator $\wedge$ and corresponding variable z, with children having corresponding variables x and y, the following clauses are put into E:

$$B = (\overline{z} \vee x) \wedge (\overline{z} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}).$$

For if B evaluates to TRUE then one of the following two situations applies:

- z is set to TRUE; then both x and y must be set to TRUE.
- z is set to FALSE; then $\overline{x} \vee \overline{y}$ must evaluate to TRUE, i.e. $x \wedge y$ evaluates to FALSE.

In sum, $z = x \wedge y$.

E consists of the 'and' of these collections of clauses, one collection per internal node in the expression tree, 'and'-ed with the clause (r).

4. Now we show that $D \in$ G-SAT $\iff$ $E \in$ SAT.

   First, suppose that E is satisfiable. Let τ be a satisfying assignment of Boolean values for E. Choose the variables in D to receive the Boolean values given by τ. By construction, if

E evaluates to TRUE, the Boolean values of the variables in E correspond to an evaluation of the expression tree for D. In particular, the Boolean value of r in E is the Boolean value of the root of the expression tree for D, or more simply of D itself. As E evaluates to TRUE by assumption, so must the clause (r), and consequently D evaluates to TRUE.

Now suppose that D is satisfiable. Let σ be a satisfying assignment of Boolean values for D. Then the following Boolean assignment causes E to evaluate to TRUE: Use σ as the truth assignment for these variables in E too. The remaining unassigned variables in E correspond to the internal nodes of the expression tree for D. These variables are given the Boolean values obtained in the evaluation of D's expression tree when the leaves have Boolean values given by σ.

As D evaluates to TRUE this means that the root of D's expression tree evaluates to TRUE, and hence so does clause (r) in E. The remaining clauses in E evaluate to TRUE exactly if the variable at each parent node v has Boolean value equal to the $\neg$, $\vee$, or $\wedge$, as appropriate, of the Boolean values of the variables for v's children; but this is exactly the Boolean values we have assigned these variables. Consequently, E evaluates to TRUE.

This shows the claim.

5. Clearly $\mathcal{A}_{\text{G-SAT}}$ runs in polynomial time if $\mathcal{A}_{\text{SAT}}$ runs in polynomial time.

□

6.5 NP-Completeness

We begin by showing that reductions compose.

Lemma 6.5.1. *If $J \leq_P K \leq_P L$ then $J \leq_P L$.*

Proof. As $J \leq_P K$ there is a polynomial time computable function f such that $x \in J \iff f(x) \in K$. And as $K \leq_P L$ there is a polynomial time computable function g such that $y \in K \iff g(y) \in L$. So $x \in J \iff g(f(x))) \in L$, and $g \circ f$ is also polynomial time computable. That is, $J \leq_P L$. □

Definition 6.5.2. *A language L is NP-Complete ($L \in \mathcal{NPC}$) if*

1. *$L \in \mathcal{NP}$, and*
2. *For every $J \in \mathcal{NP}$, $J \leq_P L$.*

Lemma 6.5.3. *If K is NP-Complete and we show that*

1. *$L \in \mathcal{NP}$, and*
2. *$K \leq_P L$*

then we can conclude L is NP-Complete also.

Proof. As $L \in \mathcal{NP}$, it suffices to show that for all $J \in \mathcal{NP}$, $J \leq_P L$. Now, for any $J \in \mathcal{NP}$, as K is NP-complete, $J \leq_P K$, and we know that $K \leq_P L$. By Lemma 6.5.1, we conclude that $J \leq_P L$. □

Lemma 6.5.3 means that once we have shown one problem is NP-Complete, e.g. General Satisfiability, then subsequent problems can be shown NP-Complete by means of reductions from G-SAT, as already given in earlier sections.

Example 6.5.4. Universal NPC (U-NPC).
Input: $\langle V, x, 1^s, 1^t \rangle$, where V is a verification algorithm taking inputs x, C.
Question: Is there is a certificate $C(x)$, with $|C(x)| \leq s$, such that in at most t steps of computation $V(x, C(x))$ outputs "Yes"?

Claim 6.5.5. *U-NPC is NP-Complete.*

Proof. The verification algorithm for U-NPC, given a candidate certificate C, checks $|C| \leq s$, and then simulates $V(x, C)$ for up to t steps, outputting "Yes" if $V(x, C)$ outputs "Yes" in this time and outputting "No" otherwise.

U-NPC $\in \mathcal{NP}$ since any $\langle V, x, 1^s, 1^t \rangle \in$ U-NPC has a certificate $C(x)$ of size at most s, which is less than the length of the input instance $\langle V, x, 1^s, 1^t \rangle$. In addition, the verification algorithm for U-NPC runs in time polynomial in the length of its input, for it needs only $O(t)$ time for the simulation of V, as it will take $O(1)$ time to simulate one step of V.

To show $L \leq_P$ U-NPC for any $L \in \mathcal{NP}$ we argue as follows. As $L \in \mathcal{NP}$, there are polynomials p and q, and L has a polynomial time verifier V_L with $V_L(x, y)$ running in time at most $p(|x| + |y|)$, such that each $x \in L$ has a certificate $C(x)$ of length at most $q(|x|)$. Thus $x \in L$ exactly if $\langle V, x, 1^{q(|x|)}, 1^{p(|x|+q(|x|))} \rangle \in$ U-NPC.

So the algorithm $\mathcal{A}_{\text{L}}$ to recognize L, given a polynomial time algorithm $\mathcal{A}_{\text{U-NPC}}$ for U-NPC, proceeds as follows.

1. Construct $\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle$, where $|x| = n$.
2. Run $\mathcal{A}_{\text{U-NPC}}(\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle)$.
3. Report the answer from (2).

It remains to argue that $\mathcal{A}_{\text{L}}$ runs in time polynomial in $n = |x|$. For the purposes of this reduction $|V_L|$ can be viewed as a constant, as it is a fixed value for all $x \in L$, so $|\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle| = O(p(n + q(n)))$, a polynomial in n. Thus $\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle$ can be constructed in time polynomial in n. Consequently, $L \leq_P$ U-NPC, and this is true for every $L \in \mathcal{NP}$. □

6.5.1 Characterizing $\mathcal{P}$

In order to prove more problems are NP-Complete it will be helpful to first show that every language in $\mathcal{P}$ can be reduced to the following Circuit Evaluation (CE) problem.

Example 6.5.6. Circuit Evaluation (CE).

Input: C, a circuit comprising *and*, *or*, and *not* gates, with Boolean inputs $x_1, x_2, \cdots, x_n$ and a Boolean output y. (The circuit can be viewed as a directed graph, where the inputs are the names of vertices with no in-edges, the output is the name of a vertex with no out-edge, and every other vertex is labeled by one of the operators (*and*, *or*, or *not*); the latter vertices have one or two inedges as appropriate, but there are no limits on the number of outedges they could have.) See Figure 6.9 for an example of such a circuit.

Question: Given an assignment of Boolean values to the inputs, does the circuit output equal TRUE?

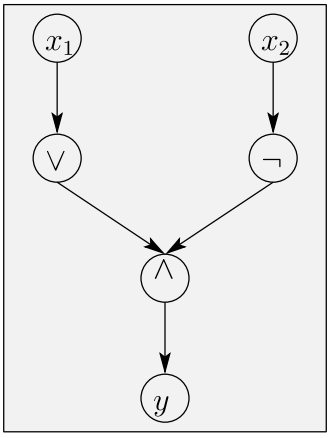

Figure 6.9: Example: A Boolean Circuit.

We now relate the Circuit Value Problem to the computation of a polynomial time Turing Machine.

Lemma 6.5.7. *Let $M = (\Sigma, A, V, start, F, \delta)$ be a 1-tape Turing Machine. Suppose that for each input x of length n, M terminates within T steps of computation. Then there is a circuit C, which takes inputs corresponding to the strings x of length n, whose size is $O(T^2) \cdot |A| \cdot |V|)$, and which outputs 1 if M recognizes x and 0 otherwise.*

Proof. Let $A = \{a_0, a_1, \ldots, a_{r-1}\}$ and $V = \{q_0, q_1, \ldots, q_{s-1}\}$.

C will have $T+1$ layers of nodes. Layer 0 will represent M's input configuration, and layer $t \geq 1$ will represent M's configuration after t steps of computation. In order to allow for the possibility that M's computation ends in fewer that T steps, we introduce the possibility of null moves once M has reached a recognizing vertex (vertices refer to locations in M's finite control, nodes to locations in C). A null move will leave M's current vertex unchanged, the current cell contents unchanged, and move the read/write head one cell to the right.

Note that as the computation now lasts exactly T steps, it uses at most $T + 1$ cells (and then only if the read/write head moves right on every step of the computation). For convenience, we number the cells from 0 to T, in left to right order. Now, we describe C's nodes in more detail.

Specification of C's nodes For each cell k, $0 \leq k \leq T$, for each time t, $0 \leq t \leq T$, there are r nodes c_{ik}^t, for $0 \leq i < r$, corresponding to the possible values in the cell.

> More specifically, $c_{ik}^t = 1$ (i.e. TRUE) exactly if after t steps of computation cell k holds character a_i.

Also, for each t, $0 \leq t \leq T$, there are an additional s nodes v_j^t, for $0 \leq j < s$, corresponding to the possible current vertices.

> More specifically, $v_j^t = 1$ exactly if after t steps of computation, the current vertex is q_j.

Finally, for each t, $0 \leq k \leq T$, there are an additional $T+1$ nodes h_k^t, for $0 \leq k \leq T$, corresponding to the possible current positions for the read/write head.

> More specifically, $h_k^t = 1$ exactly if after t steps of computation, the read/write head is over cell k.

For level $t = 0$, we ensure these properties by setting the node values to match M's starting configuration. The next step in the construction is to connect the nodes in level t to those in level $t + 1$ to preserve the above properties.

Each of the nodes we have described above, for $t > 0$, will be taking an or over its inputs. There are two cases to consider.

Case 1. The read/write head is over cell k, i.e. $h_k^t = 1$.
Suppose that $c_{ik}^t = 1$, $v_j^t = 1$, and $\delta(a_i, q_j) = (a_{i'}, q_{j'}, R)$. Then the circuit needs to ensure that $c_{i'k}^{t+1} = 1$, $v_{j'}^{t+1} = 1$, and $h_{k+1}^{t+1} = 1$. This is achieved by the edges shown in Figure 6.10.

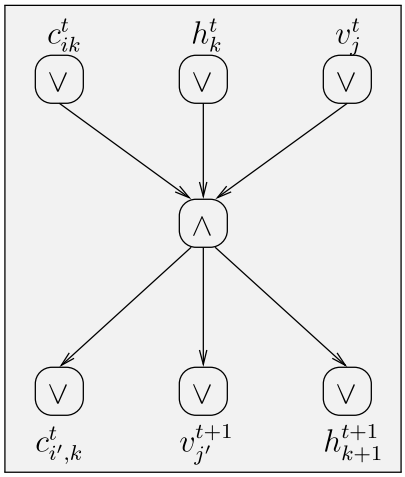

Figure 6.10: The circuit edges when the read/write head is over a cell.

If $\delta(a_i, q_j) = (a_{i'}, q_{j'}, L)$, one need make only a small change to the above construction, so as to ensure that $h_{k-1}^{t+1} = 1$ rather than $h_{k+1}^{t+1} = 1$.

Note that each of the nodes $c_{i'k}^{t+1}$, $v_{j'}^{t+1}$, h_{k+1}^{t+1} may have multiple inputs, corresponding to the multiple predecessor configurations that cause them to have value 1.

Of course, we don't know where the read/write head might be (at least not without simulating M). Thus the above construction is carried out for every triple (i, j, k), with a distinct $\wedge$-node for each triple.

Case 2. The read/write head is not over cell k, i.e. $h_k^t = 0$.
In this case the circuit needs to ensure that $c_{ik}^t = c_{ik}^{t+1}$. This is achieved by the nodes and edges shown in Figure 6.11. This part of the construction provides one more input to the node c_{ik}^{t+1}.

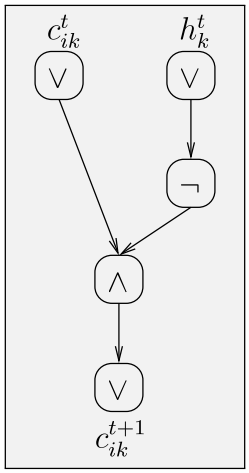

Figure 6.11: The circuit edges when the read/write head is not over a cell.

Again, the construction is carried out for every pair (i, k).

These constructions maintain the properties described in the specification, which ensures that they describe a configuration: For each k and t, there is exactly one $c_{ik}^t = 1$; i.e. cell k has content a_i after t steps of computation. For each t there is exactly one $v_j^t = 1$; i.e. the current vertex after t steps of computation is q_j. For each t there is exactly one $h_k^t = 1$; i.e. the read/write head is over cell k. And further the configuration C_t after t steps of computation is succeeded by the configuration C_{t+1} after $t + 1$ steps of computation: $C_t \vdash C_{t+1}$.

Finally, we need C to produce an output. This output should be 1 if M has reached a recognizing vertex, i.e. a vertex $q_j \in F$, and 0 otherwise. This is obtained by taking the or of the outputs of all the nodes v_j^T for $q_j \in F$.

It follows that C outputs 1 exactly if M recognizes its input x. □

Theorem 6.5.8. *Let $L \in \mathbf{P}$. Then there is a polynomial $r(n)$ and a family of circuits $C_1, C_2, \cdots$ with sizes $|C_n| \leq r(n)$, such that for each $x \in L$ with $|x| = n$,*

$$C_n(x) = \textsc{True} \iff x \in L.$$

Proof. Let $M = (\Sigma, A, V, \text{start}, F, \delta)$ be a Turing Machine recognizing L that runs in polynomial time $p(n)$, i.e. on each input of length n its computation uses at most $p(n)$ steps. Now we apply Lemma 6.5.7 with $T = p(n)$ to obtain a circuit C_n for inputs to M of size n. C_n's size can be deduced as follows. There are $(T + 1)^2|A|$ nodes c_{ik}^t, $(T + 1)|V|$ nodes v_j^t, and $(T + 1)^2$ nodes h_k^t. The number of $\wedge$ nodes not yet counted is $T(T+1) \cdot |A| \cdot |V| + T(T+1)|A|$, the number of $\neg$ nodes is $T(T+1)|A|$ (actually $T(T+1)$ would suffice), and there is one more $\vee$ node; the number of edges

is $6T(T+1)\cdot|A|\cdot|V|+4T(T+1)|A|+|F|$, where F is the set of recognizing vertices (necessarily, $|F|\leq|V|$). Thus $r(n)=O((p(n))^2\cdot|A|\cdot|V|)$. As A and V are independent of n, $r(n)=O(p(n)^2)$, which is a polynomial in n. □

6.6 More NP Complete Problems & Cook's Theorem

Example 6.6.1. Circuit Value (CV).

Input: C, a circuit comprising *and*, *or*, and *not* gates, with Boolean inputs $x_1, x_2, \cdots, x_n$ and a Boolean output y. (The circuit can be viewed as a directed graph, where the inputs are the names of vertices with no in-edges, the output is the name of a vertex with no out-edge, and every other vertex is labeled by one of the operators (*and*, *or*, or *not*); the latter vertices have one or two inedges as appropriate, but there are no limits on the number of outedges they could have. We allow some of the inputs to be preset to the values TRUE or FALSE.

Question: Is there an assignment of Boolean values to the unset inputs that causes the circuit output to evaluate to TRUE?

In the example in Figure 6.9, setting x_1 = TRUE and x_2 = FALSE causes the circuit output to evaluate to TRUE.

Claim 6.6.2. *Given a polynomial time algorithm for* 3-SAT *there is a polynomial time algorithm for* Circuit Value (CV)*:* CV $\leq_P$ 3-SAT.

Proof. Let C be the input to the CV algorithm, $\mathcal{A}_{\text{CV}}$. The algorithm proceeds as follows.

1. Compute E, where E is a Boolean formula with the property that
$$C \in \text{CV} \iff E \in \text{3-SAT}.$$
2. Run $\mathcal{A}_{\text{3-SAT}}(E)$.
3. Report the answer from (2).

$\mathcal{A}_{\text{CV}}$ constructs E using the same method as in Claim 6.4.9.

4. As argued in Claim 6.4.9, E can be satisfied exactly if C can be satisfied.
5. Clearly $\mathcal{A}_{\text{CV}}$ runs in polynomial time if $\mathcal{A}_{\text{3-SAT}}$ runs in polynomial time.

□

Theorem 6.6.3 (Cook's Theorem, 1972)**.** *CV is NP-Complete.*

Proof. Let $L \in \mathcal{NP}$. We need to show that $L \leq_P$ CV. That is, given a polynomial time algorithm for CV, we give a polynomial time algorithm for L.

As $L \in \mathcal{NP}$, we know that L has a polynomial time verifier, V_L say. By Theorem 6.5.8, there is a polynomial r, a circuit family $C_1, C_2, \cdots$, recognizing the inputs to V_L, where $|C_m| = O(r(m))$, and m is the size of V_L's input measured in bits.

Let x be an input string to be tested for membership in L. Suppose that the n input bits for x are preset in circuit $C_{n+q(n)}$, where $q(n)$ is the polynomial bound on the bit-length of certificates used by V_L. This circuit can be simplified by computing the ouputs of all gates that are determined by these preset input bits, yielding a new circuit, $C(x, n)$ say. $C(x, n)$ has $q(n)$ unspecified inputs, corresponding to the bits for candidate certificates. So $C(x, n)$ is a legitimate input for the CV problem, and in addition, $x \in L$ exactly if $C(x, n) \in$ CV.

Thus the algorithm $\mathcal{A}_L$ to recognize L proceeds as follows.

1. $\mathcal{A}_L$ builds circuit $C(x, n)$.
2. $\mathcal{A}_L$ runs $\mathcal{A}_{\text{CV}}(C(x, n))$.
3. $\mathcal{A}_L$ reports the answer given by $\mathcal{A}_{\text{CV}}(C(x, n))$.

As already noted, this algorithm computes the correct output ($x \in L \iff C(x, n) \in$ CV). Further, it runs in polynomial time, for given x, $C(x, n)$ can be constructed in time polynomial in $|x|$. □

Corollary 6.6.4. GSAT, SAT, 3-SAT, VC, Clique, IS *are all NP-Complete.*

Proof. We have shown all these languages are in NP. Cook's Theorem, Lemma 6.5.1, and the individual reductions already shown yield the claimed result. □

Definition 6.6.5. *Language L is NP-hard if for all $K \in \mathcal{NP}$, $K \leq_P L$.*

Lemma 6.6.6. *If J is NP-hard and $J \leq_P L$, then L is NP-hard.*

Proof. This is proved in the same way as Lemma 6.5.3. □

6.7 Reduction Techniques

There are several techniques which arise repeatedly in making reductions from one NP-Complete problem to another, namely:

1. Generalization.
2. Restriction.
3. Local substitution.
4. Component construction.

The boundaries between one category and another are not always clearcut, but nonetheless they can be helpful in organizing the various constructions.

6.7.1 Generalization

Quite often, a problem one wants to show NP-Complete (NPC) is a generalization of a known NPC problem. For example, G-SAT is a generalization of SAT. Showing that the generalized problem is NP-hard is trivial: the identity reduction suffices. Or to put it another way, a polynomial time algorithm for the generalized problem is also a polynomial time algorithm for the more specialized problem.

6.7.2 Restriction

This is a complement to generalization: here one wants to show a restricted version of an NPC problem is also NPC. This is not always the case, so there is some work to do. One example is 3-SAT, which is a restricted version of SAT, which in turn is a restricted version of G-SAT. What is needed in the reduction is a way of encoding an instance of the general problem in the more restricted version. This is best seen by example.

Example 6.7.1. *Degree 4 Directed Hamiltonian Cycle* (4-DHC) is the Directed Hamiltonian Cycle problem on directed graphs where each vertex has degree (indegree plus outdegree) bounded by 4.

Claim 6.7.2. *Given a polynomial time algorithm for* 4-DHC *there is a polynomial time algorithm for* DHC (DHC $\leq_P$ 4-DHC).

Proof. Given a graph G, as input to $\mathcal{A}_{\text{DHC}}$, the main step (Step 1) is to construct a graph H, with degree bound 4, with the property that

$$G \in \text{DHC} \iff H \in \text{4-DHC}$$

and to do this in polynomial time.

Let d be the largest degree of any node in G, and suppose that $2^{h-1} < d \leq 2^h$. H is the following graph. First, there is a copy of each vertex in G. However, the edges are replaced by a more elaborate construction. For each node v in G with inedges $e_1, e_2, \cdots, e_k$, H receives cycles $C_h, C_{h-1}, \cdots C_2$ of $2^h, 2^{h-1}, 2^{h-2}, \cdots, 4$ vertices respectively, as illustrated in Figures 6.12 and 6.13. Each edge $e_i = (u, v)$ is replaced by an edge $e'_i = (u', v')$; the new edges are paired, and each pair will be incident on distinct alternate vertices in C_h. In turn, the remaining 2^{h-1} vertices in C_h are incident, two by two, on alternate vertices in C_{h-1}. This construction repeats, down to C_2, whose two outedges go to v.

A symmetric construction is used for the outedges.

Clearly all the vertices in H have degree at most 4. Further H has at most $O(|E|+|V|)$ vertices, where $G = (E, V)$, so $|H| = O(|G|)$, and further H is readily constructed in time $O(|G|)$.

Next, we argue that $G \in \text{DHC} \iff H \in \text{4-DHC}$.

First, suppose that G has a Hamiltonian Cycle C. Suppose it uses some edge $e = (u, v)$. We describe the path taken in H corresponding to edge (u, v). It consists of the edge $e' = (u', v')$ preceded by a path from u to u' and followed by a path from v' to v. We describe the latter path in more detail. e' enters cycle C_h at vertex v'. The path in H then goes round all the vertices in C_h, leaving it at the 2^hth vertex visited, taking an edge into C_{h-1}. C_{h-1} is then traversed, being left at the 2^{h-1}th vertex to go to C_{h-2}, and so forth, until eventually C_2 is traversed and departed by an edge into v. This causes all the vertices on the cycles on the "into" side of v to be traversed; similarly, on leaving v, all the vertices on the cycles on v's "out" side will be traversed. Thus this process creates a Hamiltonian Cycle for H.

Second, suppose that H has a Hamiltonian Cycle D. Let $v_1, v_2, \cdots, v_n$ be the order in which the original vertices of G are visited by D. Then $v_1, v_2, \cdots, v_n$ form a Hamiltonian Cycle of G. This follows because once the path reaches the "in" vertices of v_i, it goes through v_i and then visits the "out" vertices of v_i before proceeding to v_{i+1}'s vertices. As v_i can be traversed only once, all

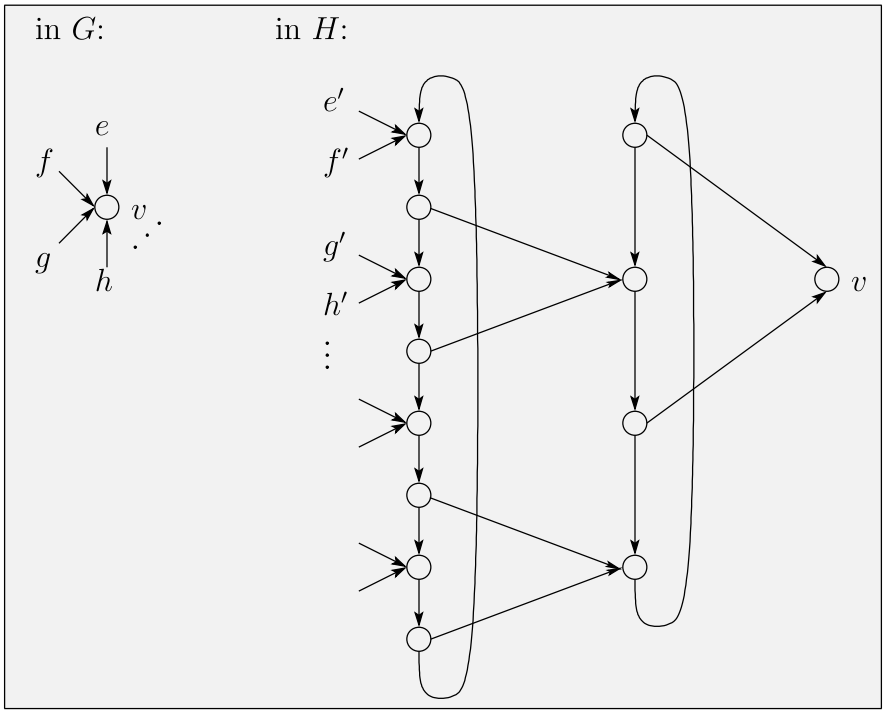

Figure 6.12: The Inedge Gadget for the 4-DHC Construction.

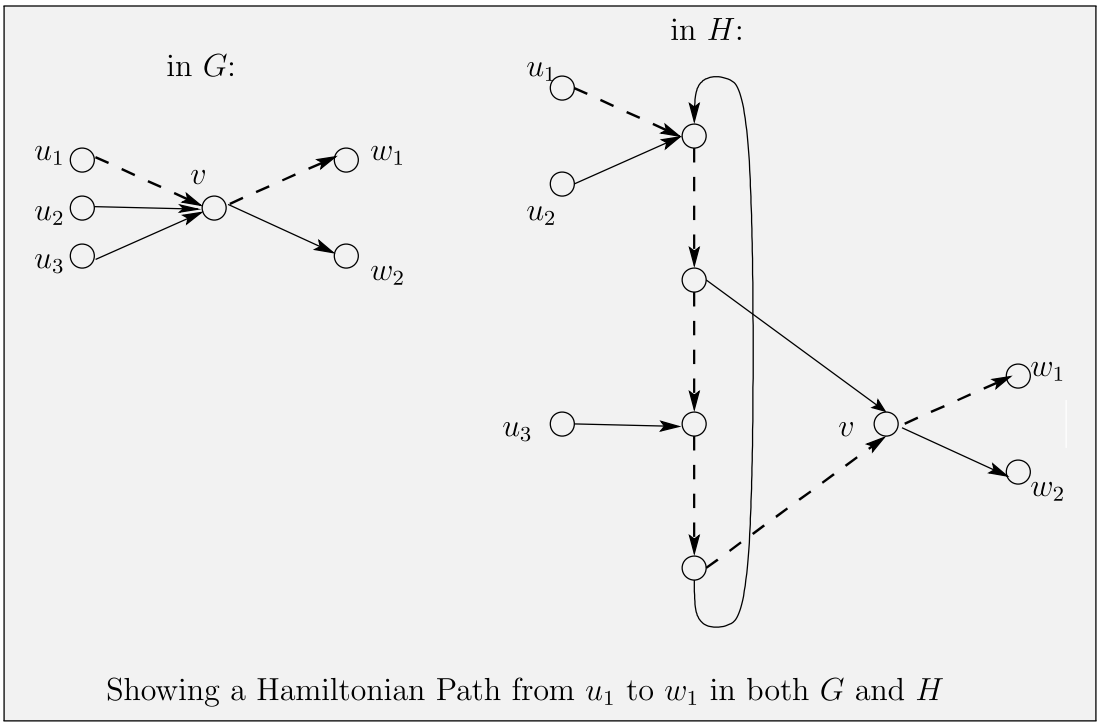

Figure 6.13: The Inedge Gadget for an Example Vertex.

its "in" and "out" vertices must be visited right before and right after v_i is visited. Thus the path from v_i to $v_{i+1 \bmod n}$ in D must go directly from the "out" vertices for v_i to the "in" vertices for $v_{i+1 \bmod n}$ and hence must use edge $(v'_i, v'_{i+1 \bmod n})$, which implies that $(v_i, v_{i+1 \bmod n})$ is an edge of G.

This demonstrates the claim that $G \in \text{DHC} \iff H \in \text{4-DHC}$. □

6.7.3 Local Replacement

In local replacement, each element of the original problem instance is replaced by a small structure in the new problem instance. We have already seen several examples of this, including the constructions showing DHP $\leq_P$ UHP and SAT $\leq_P$ 3-SAT. The latter is also an instance of restriction.

Example 6.7.3. *Dominating Set* (DS).
Input: (G, k), where $G = (V, E)$ is an undirected graph and $k \leq |V|$ is an integer.
Question: Does G have a dominating set $U \subseteq V$ of size k, that is a set U of vertices of size k, such that every vertex is either in U or adjacent to a vertex in U.
Note that this problem seems quite similar to the Vertex Cover problem, but it is a different question that is being asked.

Claim 6.7.4. *Given a polynomial time algorithm for* DS *there is a polynomial time algorithm for* VC.

Proof. Given an input (G, k) to $\mathcal{A}_{\text{VC}}$, the main step (Step 1) is to construct a pair (H, h) with the property that

$$(G, k) \in \text{VC} \iff (H, h) \in \text{DS}$$

and to do this in polynomial time.

H is obtained as follows. For each vertex v in G a copy, also called v, is put in H. For each edge $e = (u, v)$ in G, an "edge" vertex e is put in H; in addition, edges (u, e) and (v, e) are created in H. Finally, H receives two more vertices: y and z; y is connected to every vertex v, where v is a copy of a vertex in G, and z is connected only to y. h is set equal to $k + 1$. An example of graph H is shown in Figure 6.14.

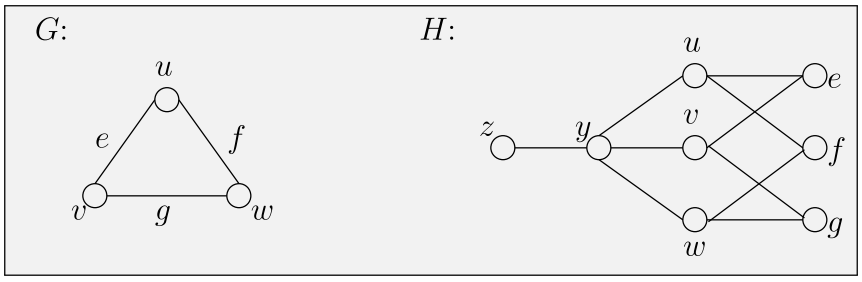

Figure 6.14: The Vertex Cover to Dominating Set Reduction; An Example Graph.

Clearly $|H| = O(|G|)$ and H can be constructed in $O(|G|)$ time.
Next, we argue that $(G, k) \in \text{VC} \iff (H, h) \in \text{DS}$.

First, suppose that G has a vertex cover U of size k. Then we claim that U plus the vertex y forms a Dominating Set for H. Since U is a vertex cover it covers every edge in G. Thus U covers all the "edge" vertices in H. y covers the remaining vertices in H.

Next, suppose that H has a dominating set U' of size h. We modify U' as follows. If $z \in U'$ then it is replaced by y; this ensures all the non-edge vertices in H are covered. Likewise, if any edge vertex $(u, v) = e \in U'$ then it is replaced by either one of u or v; e continues to be covered. Let U'' be the resulting set; U'' is also a dominating set for H and it has size at most h. Next, we note that $y \in U''$ as z is covered. Let $U = U'' - \{y\}$. So $|U| \leq h - 1 = k$. We claim that U is a vertex cover for G. This follows because in H every edge vertex is covered by U, and thus in G every edge is covered by U.

This demonstrates the claim that $(G, k) \in \text{VC} \iff (H, h) \in \text{DS}$. □

6.7.4 Component or Gadget Construction

In gadget construction, each element of the original problem instance is replaced by a possibly quite elaborate structure in the new problem instance. We have seen an example of such a reduction, namely 3-SAT $\leq_P$ IS; the next section gives another example. The separation of local replacement and gadget construction is not clearcut, but by the latter we intend constructions which are not just a modest tweak of the original input.

6.8 Examples of More Elaborate Reductions

6.8.1 Vertex Cover to Directed Hamiltonian Cycle

Claim 6.8.1. *Given a polynomial time algorithm for* Directed Hamiltonian Cycle (DHC) *there is a polynomial time algorithm for* Vertex Cover (VC)*:* VC $\leq_P$ DHC.

Proof. Let (G, k) be the input to the VC algorithm, $\mathcal{A}_{\text{VC}}$. The algorithm proceeds as follows.

1. Compute H, where H is a directed graph, with the property that
$$(G, k) \in \text{VC} \iff H \in \text{DHC}.$$
2. Run $\mathcal{A}_{\text{DHC}}(H)$.
3. Report the answer given by $\mathcal{A}_{\text{DHC}}(H)$.

Let $G = (V, E)$. $\mathcal{A}_{\text{VC}}$ obtains H as follows. For each edge $e = (u, v) \in E$, it adds four vertices to H, namely $(u, e, 0), (u, e, 1), (v, e, 0), (v, e, 1)$, connected as shown in Figure 6.16(a); this is called an *edge gadget*. Then, for each vertex $u \in V$, with incident edges $e_1, e_2, \cdots, e_l$, the corresponding vertices, two per edge, are connected as shown in Figure 6.16(b). This can be thought of as corresponding to u's adjacency list, except that there are two entries per edge; we call this sequence of $2k$ nodes the *adjacency list* for u (in H). Finally, k *selector* vertices $s_1, s_2, \cdots, s_k$ are added to H, together with edges from each s_i to the first vertex on each vertex u's "adjacency list" (i.e. edges $(s_i, (u, e_1, 0))$, and edges from the last vertex on each vertex u's adjacency list to each s_i (i.e. edges $((u, e_l, 1), s_i)$. This completes the description of graph H. An example of a pair $(G, 2)$ and the corresponding H are shown in Figure 6.15.

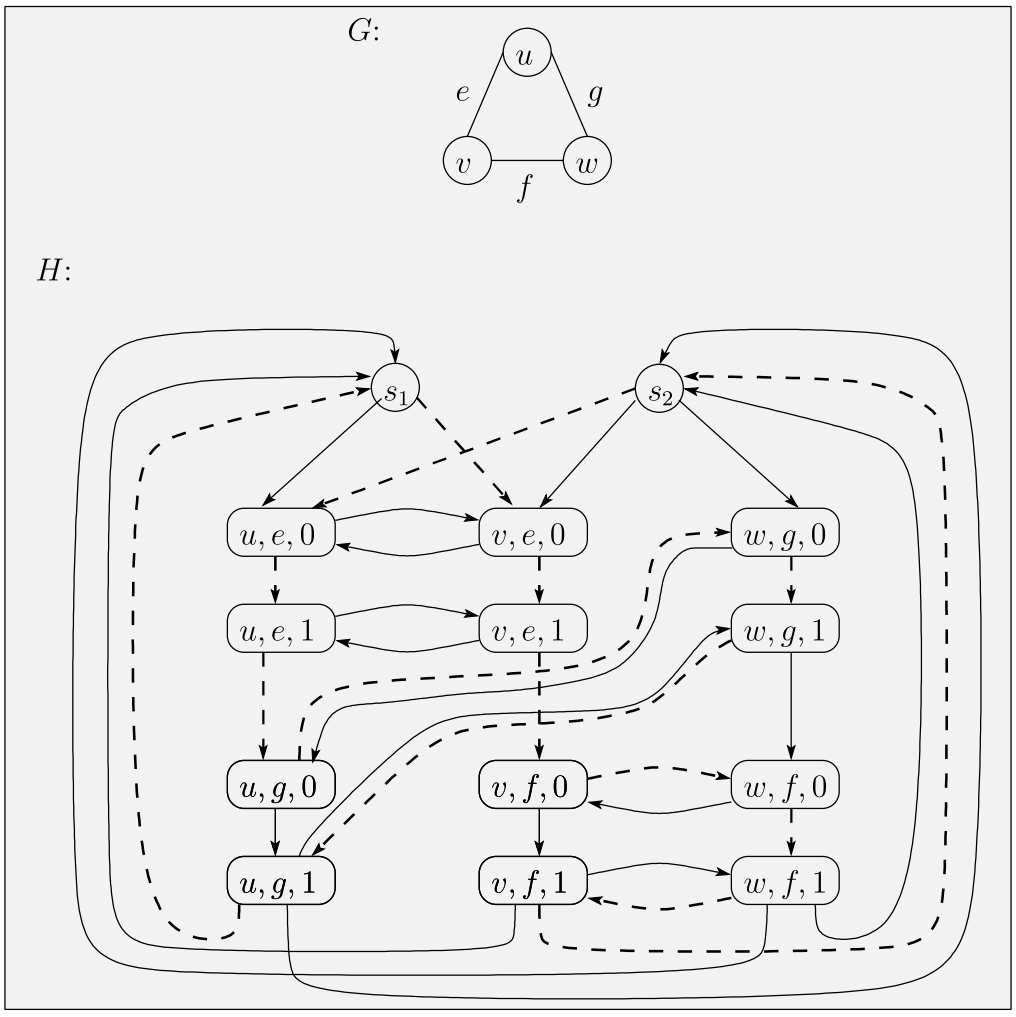

Figure 6.15: The graph H corresponding to $(G, 2)$. A Hamiltonian Cycle is shown with dashed edges.

4. We argue that H has a Hamiltonian Cycle exactly if G has a vertex cover of size k.

   First, suppose that G has a vertex cover of size k, $U = \{u_1, u_2, \cdots, u_k\}$ say. A preliminary, and incomplete Hamiltonian Cycle C' is given by the following cycle: It starts at s_1 and then goes through the adjacency list for vertex u_1, then goes to s_2, then traverses the adjacency list for u_2, and so on, eventually traversing the adjacency list for u_k, following which it completes the cycle by returning to s_1. C' needs to be modified to include the nodes on the adjacency lists of vertices outside the vertex cover U. The nodes in H which have not yet been traversed all correspond to the v end of edges (u, v) with just one endpoint, u, in U. (For as U forms a vertex cover, there is no edge with zero endpoints in U.) In the example in Figure 6.15, the vertices not on C' are $(w, g, 0)$, $(w, g, 1)$, $(w, f, 0)$, and $(w, f, 1)$. To include them, it suffices to replace the edge $((u, g, 0), (u, g, 1))$ with the sequence of three edges obtained by traversing $(u, g, 0), (w, g, 0), (w, g, 1), (u, g, 1)$ in that order, and the edge $((v, f, 0), (v, f, 1))$ with the sequence of three edges obtained by traversing $(v, f, 0), (w, f, 0), (w, f, 1), (v, f, 1)$ in that order. The result is still a cycle, but now it goes through every vertex of H exactly once, so it forms a Hamiltonian Cycle.

   Next, suppose that H has a Hamiltonian Cycle C. Let $(u_i, e_{u_i,1}, 0)$ be the vertex following s_i on C, for $1 \leq i \leq k$. Then we claim that $U = \{u_1, u_2, \cdots, u_k\}$ forms a vertex cover for G. To see this we need to understand how C traverses the adjacency list of each of the u_i.

   We claim that there are three ways for C to traverse the edge gadget for edge $e = (u, v)$ in G, as illustrated in Figure 6.17. For there are two vertices by which the gadget can be entered ($(u, e, 0)$ and $(v, e, 0)$), and two by which it can be exited ($(v, e, 1)$ and $(u, e, 1)$). Either it is entered twice and exited twice, in which case the corresponding entry and exit vertices are joined by single edges (Figure 6.17(a)) or it is entered and exited once, in which case all four vertices lie on the path between the corresponding entry and exit vertices (Figure 6.17(b) or (c)); the latter 4-vertex paths are called *zig-zags*.

   Let $e_1, e_2, \cdots, e_l$ be the edges incident on some $u = u_i \in U$, and suppose the edges appear in that order on u's adjacency list in H. As the first edge gadget on u's adjacency list in H is entered by edge $(s_i, (u, e_1, 0))$, C must go directly or by zig-zag from $(u, e_1, 0)$ to $(u, e_1, 1)$, then directly to $(u, e_2, 0)$, then directly or by zig-zag from $(u, e_2, 0)$ to $(u, e_2, 1)$, then directly to $(u, e_3, 0)$, and so on until it reaches $(u, e_l, 1)$, from where it goes to some $s_{i'}$, $i' \neq i$. By renumbering indices if needed, we can allow $i' = i + 1 \bmod k$.

   Thus C traverses k adjacency lists directly; any other nodes on C are reached by means zig-zags. These other nodes correspond to edge endpoints for edges with one endpoint in the traversed adjacency lists. This shows that for each such edge one endpoint is in U. As all nodes in H are traversed, it follows that every endpoint in a non-traversed adjacency list must have its other endpoint in U. Consequently U forms a vertex cover.

   This shows the claim.

5. Clearly $\mathcal{A}_{\text{VC}}$ runs in polynomial time if $\mathcal{A}_{\text{DHC}}$ runs in polynomial time.

□

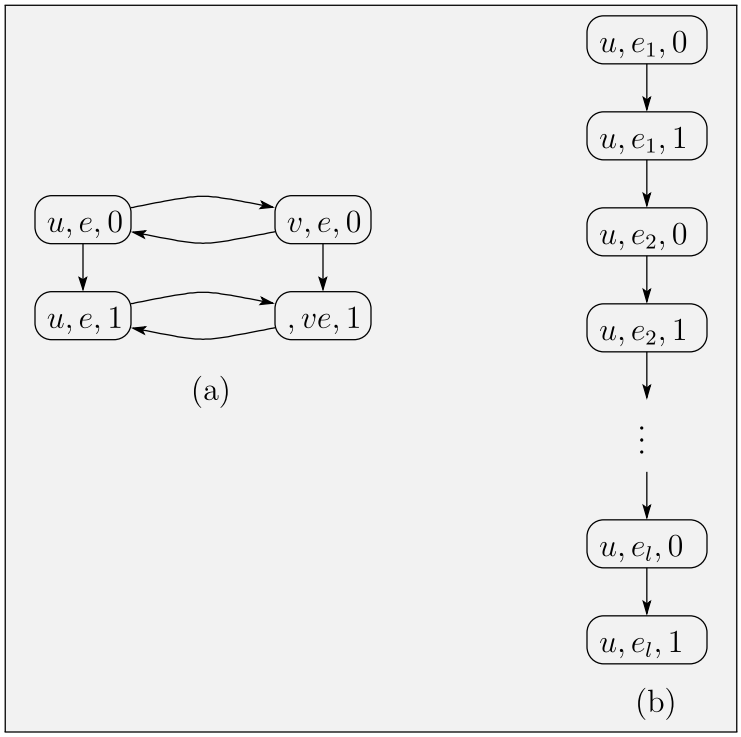

Figure 6.16: The structure in H due to edge (u, v) in G and to u's adjacency list for edges $e_1, e_2, \cdots, e_l$.

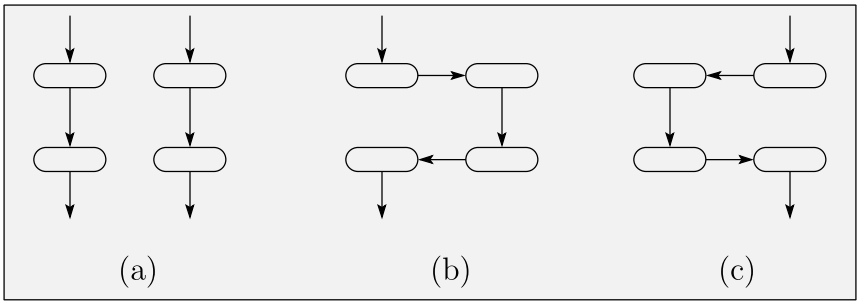

Figure 6.17: The Three Possible Ways of Traversing the Edge Gadget.

Exercises

1. Let Half-SAT be the following language:

$$\text{Half-SAT} = \{F \mid F \text{ is a CNF formula with } 2n \text{ variables and there is a satisfying assignment in which } n \text{ variables are set to TRUE and } n \text{ variables are set to FALSE}\}.$$

Show that Half-SAT is NP-Complete, that is (i) show that Half-SAT has a polynomial time verifier, and (ii) Supposing that you were given a polynomial time algorithm for Half-SAT, use it to give a polynomial algorithm for SAT.

Sample Solution. i. The certificate comprises an assignment σ of truth values to the variables in F. Since $|\sigma| = O(|F|)$, the certificate has length linear in $|F|$.

To check that a candidate certificate is valid, the verifier checks (a) that F is a CNF formula with an even number of variables; (b) that the certificate has exactly n variables set to TRUE and n set to FALSE; (c) that the assignment of truth values in the certificate causes F to evaluate to TRUE. It is straightforward to implement the verifier to run in polynomial time, and clearly it outputs "Yes" exactly when $F \in$ Half-SAT.

Finally, note that if there is a valid certificate then the input is in Half-SAT, and if the input $I \in$ Half-SAT, then there is a satisfying assignment σ with half the variables set to TRUE and half set to FALSE, and σ is a valid certificate.

ii. Let F' be the input to SAT. The algorithm $\mathcal{A}_{\text{SAT}}$ to test if $F' \in$ SAT builds a CNF formula F with the property that

$$F' \in \text{SAT} \iff F \in \text{Half-SAT}.$$

It then runs the algorithm for Half-SAT, $\mathcal{A}_{\text{H-SAT}}$, on input F and reports the answer as its own result.

F is built as follows. Suppose that F' has variables $x_1, x_2, \cdots, x_n$. F will have variables $x_1, x_2, \cdots, x_n$ and $y_1, y_2, \cdots, y_n$. The idea is that $x_i = \overline{y_i}$ for all i, $1 \leq i \leq n$. This is enforced by including clauses $(x_i \vee y_i) \wedge (\overline{x_i} \vee \overline{y_i})$ for $1 \leq i \leq n$. For $(x_i \vee y_i) \wedge (\overline{x_i} \vee \overline{y_i})$ evaluates to TRUE only if $x_i =$ TRUE and $\overline{y_i} =$ TRUE or if $x_i =$ FALSE and $\overline{y_i} =$ FALSE, i.e. only if $x_i = \overline{y_i}$.

Clearly F can be built in polynomial time. Also, clearly, if $\mathcal{A}_{\text{H-SAT}}$ runs in polynomial time, then so does $\mathcal{A}_{\text{SAT}}$.

Now we argue that $F' \in$ SAT $\iff$ $F \in$ Half-SAT.

It is readily seen that if F' has a satisfying assignment σ, setting truth values for the x_i in F according to σ, and then for the y_i according to the rule $y_i = \overline{x_i}$ produces a satisfying assignment for F with exactly n variables set to True. Thus $F' \in$ SAT implies $F \in$ Half-SAT.

Likewise, a satisfying assignment for F restricted to the x variables is a satisfying assignment for F'. Thus $F \in$ Half-SAT implies $F' \in$ SAT.

This shows that $F' \in$ SAT $\iff$ $F \in$ Half-SAT.

2. Let Non Tautology be the following problem.
   Input: A DNF formula F (i.e., a boolean formula which is an "or" of clauses, where each clause is an "and" of boolean variables and their complements).
   Question: Does F have a non-satisfying assignment, that is an assignment of truth values to its Boolean variables that causes the formula to evaluate to FALSE?
   e.g. $F_1 = (x_1) \vee (x_2)$ has the non-satisfying assignment $x_1 =$ FALSE, $x_2 =$ FALSE; $F_2 = (x_1 \wedge x_2) \vee (\overline{x}_1 \wedge \overline{x}_2)$ has the non-satisfying assignment $x_1 =$ FALSE, $x_2 =$ TRUE.

   Show that Non Tautology has a polynomial time verifier. That is, given a candidate certificate C of size polynomial in n, where n is the input size, there is a polynomial time algorithm to check whether C certifies that the input F is in the language Non Tautology, and furthermore, for each $F \in$ Non Tautology, there is a polynomial-sized certificate C for F.

3. Let Subset Sum be the following problem.
   Input: A collection of n not necessarily distinct positive integers and a target integer t, where the integers are given in binary form.
   Question: Is there a subset of the collection, such that the numbers in the subset sum to exactly t?
   e.g. Let the collection of integers be $\{1, 2, 2, 6\}$ and the target be 5. The subset adding up to the target is 1,2,2.

   Show that Subset Sum has a polynomial time verifier.

4. The Traveling Peddler (TP) is the following problem.
   Input: A directed graph $G = (V, E)$, where each edge has a non-negative integer length, and an integer b.
   Question: Does G have a Hamiltonian Cycle of length at most b?

   Show that Traveling Peddler has a polynomial time verifier.
   Note: This problem is often called the Traveling Salesman problem.

5. Let Composites be the following problem.
   Input: An n-bit integer m.
   Question: Is m a composite, that is a non-trivial product of two integers; in other words are there integers r and s, with $r, s \neq 1$, where $m = rs$?

   Show that Composites has a polynomial time verifier.
   Comment. Interestingly, there is a polynomial time algorithm (called a *primality test*) to test this property; however this algorithm does this without identifying a pair r and s of divisors for m.

6. Suppose that you were given a polynomial time algorithm for Directed Hamiltonian Path (DHP). Using it as a subroutine, give a polynomial time algorithm for Undirected Hamiltonian Path (UHP). (Claim 6.4.2 demonstrates the reduction in the opposite direction).

7. Suppose that you were given a polynomial time algorithm for Undirected Hamiltonian Path (UHP). Using it as a subroutine, give a polynomial time algorithm for Undirected Hamiltonian Cycle (UHC).

8. Suppose that you were given a polynomial time algorithm for Traveling Peddler. Using it as a subroutine, give a polynomial time algorithm for Directed Hamiltonian Cycle. See Problem 4 for the definition of the Traveling Peddler problem.

9. Let k-Color be the following problem.
   Input: An undirected graph G.
   Question: Can the vertices of G be colored using k distinct colors, so that every pair of adjacent vertices are colored differently?

   Suppose that you were given a polynomial time algorithm for $(k+1)$-Color. Use it to give a polynomial algorithm for k-Color. This means that you need to provide a polynomial time algorithm that on input G, a graph, needs to construct another graph H, with the property that $G \in k\text{-Color} \iff H \in (k+1)\text{-Color}$.

10. Let Equal Subset Sum be the following problem.
    Input: A collection of n not necessarily distinct positive integers, $a_1, a_2, \cdots, a_n$, given in binary form.
    Question: Is there a subset of the collection, such that the numbers in the subset sum to exactly $\frac{1}{2}\sum_{i=1}^{n} a_i$?

    Suppose that you were given a polynomial time algorithm for Equal Subset Sum. Use it as a subroutine to give a polynomial time algorithm for Subset Sum. See Problem 3 for the definition of the Subset Sum problem.

11. Let Half-Clique be the following problem.
    Input: An undirected graph $H = (W, F)$, with $n = |W|$ being an even integer.
    Question: Does H have a size $n/2$ clique?

    Suppose that you were given a polynomial time algorithm for Half-Clique. Using it as a subroutine, give a polynomial time algorithm for Clique. This means that you need to provide a polynomial time algorithm that on input (G, k), G an undirected graph and k an integer, constructs another graph H, with the property that $(G, k) \in \text{Clique} \iff H \in \text{Half-Clique}$.

12. Let 2-Satisfying Assignments be the following problem.
    Input: A CNF formula F (i.e., a Boolean formula which is an "and" of clauses, where each clause is an "or" of Boolean variables and their complements).
    Question: Does F have two distinct satisfying assignments?
    e.g. $F_1 = x_1 \vee x_2$ has the satisfying assignments $x_1 = \textsc{True}$, $x_2 = \textsc{False}$ and $x_1 = \textsc{False}$, $x_2 = \textsc{True}$; $F_2 = (x_1 \vee x_2) \wedge (\overline{x}_1)$ has just one satisfying assignment, namely $x_1 = \textsc{False}$, $x_2 = \textsc{True}$.

    Suppose that you were given a polynomial time algorithm for 2-Satisfying Assignment. Using it as a subroutine, give a polynomial time algorithm for SAT. This means that you need to provide a polynomial time algorithm that on input F, a Boolean CNF formula, constructs a Boolean CNF formula $\widetilde{F}$ such that

    $$F \in \text{SAT} \iff \widetilde{F} \in \text{2-Satisfying Assignments}$$

    i.e. F is satisfiable if and only if $\widetilde{F}$ has at least 2 distinct satisfying assignments.

13. Let Kernel be the following problem.
Input: A directed graph $G = (V, E)$.
Question: Does G have a kernel? A kernel is a subset K of vertices such that no edge joins two vertices in the kernel and for every vertex $v \in V - K$ there is a vertex $u \in K$ such that $(u, v) \in E$.

    Suppose that you were given a polynomial time algorithm for Kernel. Using it as a subroutine, give a polynomial time algorithm for 3-SAT.
Hint: Create a directed triangle for each clause with one vertex per literal (even if the clause has fewer than 3 literals), and a directed cycle for each variable, with one vertex for each truth assignment. Connect the vertices in the cycles to the vertices in the triangles appropriately. If F is satisfiable the kernel will contain one vertex from each triangle and one vertex from each cycle.

14. Suppose that you were given a polynomial time algorithm for Satisfiability, that is an algorithm that reports whether or not the input is a satisfiable CNF formula.

    Use it to give a polynomial time algorithm to find a satisfying assignment σ if the formula is satisfiable.
Hint. Let $x_1, x_2, \cdots, x_m$ be the variables in the formula. Consider an algorithm that proceeds as follows: First, determine if F is satisfiable. If so, construct a satisfying assignment σ in n iterations, with each iteration determining the truth value for one more variable. The first iteration determines if there is a satisfying assignment with $x_1 = \textsc{True}$; if so it sets $x_1 = \textsc{True}$ and otherwise it sets $x_1 = \textsc{False}$. Be careful as to how you continue, for note that even if each of $x_1 = \textsc{True}$ and $x_2 = \textsc{True}$ occur in satisfying assignments, it need not be the case that they both occur in a single satisfying assignment (e.g. $F = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$).

15. Let 3-Color be the following problem.
Input: A undirected graph $G = (V, E)$.
Question: Does G have a 3-coloring, that is can the vertices of G be colored using 3 colors in such a way that every pair of adjacent vertices have distinct colors.

    Suppose that you were given a polynomial time algorithm for 3-Color. Using it as a subroutine, give a polynomial time algorithm for 3-SAT.

    To do this you need to provide a polynomial time algorithm that on input F, a Boolean 3-CNF formula, constructs a graph G such that F is satisfiable if and only if G is 3-colorable.

    G will contain one instance of the Color Defining Gadget (Definer for short) shown in Figure 6.18 (a triangle). The colors given to the Definer (T, F and N) are viewed as corresponding to $\textsc{True}$, $\textsc{False}$, and $\textsc{Neutral}$. For each variable in F, G will contain a pair of vertices in a Variable gadget connected to the Definer as shown. Supposing that the Definer vertices are colored T, F, N, as shown, what colors do the two vertices in the Variable gadget corresponding to the literals x and $\overline{x}$ take on?

    Now consider the OR gadget. Suppose that its bottom two nodes are both colored T; what color(s) can its top node take on? What if they are both F? What if one is T and one is F? Use two OR gadgets to simulate the evaluation of a clause. By connecting the top node of the combined two OR gadgets to the Definer, and identifying the bottom nodes with the nodes in the Variable gadgets, make the OR-gadgets used for the clause 3-colorable exactly if the clause

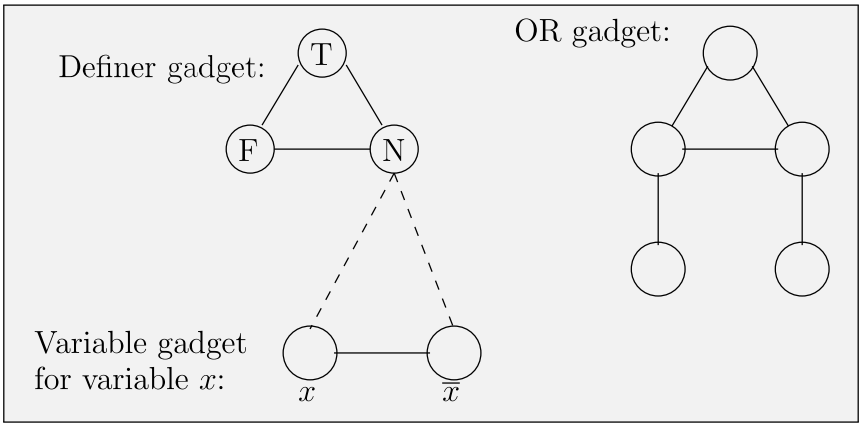

Figure 6.18: The Gadget for problem 15.

evaluates to TRUE on the corresponding truth setting for the Boolean variables. Remember to handle clauses of two and one literals also.

16. Suppose that you were given a polynomial time algorithm for G-SAT. Use it to give a polynomial algorithm for Clique.

    That is, given a graph G and an integer k, the task is to give an algorithm to determine if G has a clique of size k. To this end, your algorithm will construct a Boolean formula $F_{G,k}$ such that $F_{G,k}$ is satisfiable if and only if G has a clique of size k. The main task is to describe how to construct $F_{G,k}$ in polynomial time. Broad-brush, this construction is analogous to that of Claim 6.5.7.

17. Let No-Tri-VC be the vertex cover problem limited to undirected graphs which have no triangles (collections of 3 vertices u, v, w with all three edges $(u, v), (v, w), (w, v)$ present).

    Suppose that you were given a polynomial time algorithm for No-Tri-VC. Use it to give a polynomial algorithm for Vertex Cover. This means that you need to provide a polynomial time algorithm that on input (G, k), where G is an undirected graph and $k \geq 0$ is an integer, needs to construct an undirected triangle-free graph H and an integer l, with the property that $(G, k) \in \text{VC} \iff (H, l) \in \text{No-Tri-VC}$.

18. The Timetable Problem. The input has several parts: an integer t, a list $F_1, F_2, \cdots, F_k$ of k final exams to schedule, a list of students $S_1, S_2, \cdots, S_n$; in addition, each student is taking some subset of exams, specified in a list SL_i for student i, $1 \leq i \leq n$. The task is to schedule the exams so that a student is scheduled for at most one exam in any given time slot. The problem is to determine if there is a schedule using only t time slots.

    Show that the Timetable Problem is NP-Complete.

    Hint: Try reducing 3-Color to the Timetable Problem (see Problem 15). That is, you have the following task. Given a graph $G = (V, E)$, an input to the 3-Color problem, you need to create an input T for the Timetable problem, such that G is 3-colorable if and only if T is schedulable. The main issues in the construction are to decide what in the timetable corresponds to a vertex of G and what corresponds to an edge. Don't forget to choose a suitable value for t.