Chapter 5

NP Completeness

5.1 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 define feasibility. To this end, we define feasibility to be Deterministic Polynomial Time, $P$ for short. This is by contrast with another class, Non-Deterministic Polynomial Time, $NP$ for short. As we will see, $P \subseteq NP$, and further it is an essentially universal belief that $P \subsetneq NP$; the hardest problems in $NP - P$, 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 to $O(\log n)$ bits, so repeated squaring of a number would not be a legitimate series of steps.

Definition 5.1.1. A program or algorithm has worst case running time $T(n)$ 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.)

5.1.1 The Class $P$

Definition 5.1.2. An algorithm runs in polynomial time if its running time $T(n)$ is bounded by a polynomial function of $n$.

Definition 5.1.3. A problem or language is in $P$ (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 \( n^{100} \) time 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 \( P \) 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 \( P \) 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 \( A \) can solve a problem of size \( n \) in one hour on a particular computer, if \( A \) 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 \( A \). By running twice as fast I mean that it performs twice as many operations in the given time. While if \( A \) runs in quadratic time (\( \theta(n^2) \)), then given a new computer that runs four times as fast, in one hour one could solve a problem that is twice as large; again, if \( A \) 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.

### 5.1.2 The Class NP

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

**Example 5.1.4. Hamiltonian Circuit.**

*Input:* A directed graph \( G = (V, E) \).

*Question:* Does \( G \) have a Hamiltonian Circuit, 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 Circuit 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 \( v_1, v_2, \ldots, v_n \), 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 Circuit, 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 Circuit(s). On the other hand, if the graph is not Hamiltonian, i.e. it does not have a Hamiltonian Circuit, then no attempted certificate will check out, for no proposed sequence of vertices will form a Hamiltonian Circuit. However, it might be that you are given a proposed certificate for a Hamiltonian graph which fails because it is not a Hamiltonian Circuit. 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 Circuit for this particular graph. You cannot conclude whether or not the graph is Hamiltonian from this.

**Example 5.1.5. 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 is 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 some pair of 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 5.1.6. Satisfiability.**
Input: \(F\), a CNF Boolean formula.

\(F\) is in CNF form if \(F = C_1 \land C_2 \land \cdots \land C_m\), where each clause \(C_i, 1 \leq i \leq m\), is an ‘or’ of literals:

\[ C_i = l_{i1} \lor l_{i2} \lor \cdots \lor l_{ij_i}, \]

where each \(l_{ik}\) is a Boolean variable or its complement (negation).

e.g. \(F_1 = (x_1 \lor x_2) \land (\overline{x_1} \lor \overline{x_2} \lor x_3) \land (x_3 \lor x_2); F_2 = x; F_3 = x \land \overline{x}.\)

Question: Is \(F\) satisfiable? That is, is there an assignment of truth values to \(F\)’s variables that causes \(F\) to evaluate to \(\text{TRUE}\)?

e.g. For \(F_1\), setting \(x_1 = \text{TRUE}, x_2 = \text{FALSE}, x_3 = \text{TRUE}\), causes \(F_1\) to evaluate to \(\text{TRUE}\); For \(F_2\), setting \(x = \text{TRUE}\) causes \(F_2\) to evaluate to \(\text{TRUE}\); for \(F_3\), no setting of the variables will cause \(F_3\) to evaluate to \(\text{TRUE}\).

Here, the additional information, or certificate, is the assignment of truth values to the formula’s variables that causes the formula to evaluate to \(\text{TRUE}\). Notice that if the formula
5.1. INTRODUCTION

never evaluates to \textsc{True}, then any proposed truth assignment for the Boolean variables will cause the formula to evaluate to \textsc{False}. In other words, any proposed certificate fails.

Again, just because you are given a truth assignment that cause the formula to evaluate to \textsc{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 \textsc{False} in this case).

**Definition 5.1.7.** An assignment \( \sigma \) for the variables of a Boolean formula \( F \) is said to be satisfying if it causes the formula to evaluate to \textsc{True}.

**Definition 5.1.8.** A language \( L \in \mathbf{NP} \) 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 \textsc{“Yes”}.
- While if \( x \notin L \), for every string \( s \) of length at most \( p(|x|) \), \( V(x, s) \) outputs \textsc{“No”}.

In addition, \( V(x, s) \) runs in time \( q(|x| + |s|) = O(q(n + p(n))) \), where \(|x| = n\).

**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 \( x \notin L \).

5.1.3 Discussion

It is not known whether \( \mathbf{P} = \mathbf{NP} \) or \( \mathbf{P} \subsetneq \mathbf{NP} \) (\( \mathbf{P} \subseteq \mathbf{NP} \) as every problem \( Q \) in \( \mathbf{P} \) 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 \( \mathbf{NP} \), and in particular for the so-called \( \mathbf{NP} \)-Complete problems, or \( \mathbf{NPC} \) problems for short, it is essentially universally believed that \( \mathbf{P} \neq \mathbf{NP} \). \( \mathbf{NPC} \) problems are the hardest problems in the class \( \mathbf{NP} \) 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, a 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 \( \mathbf{NP} \) 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 NPC problem, that is it is not believed to be a “hardest” problem in the class NP.

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

5.2 Reductions among NP Problems

As with undecidable problems, the tool to show problems are NPC problems is a suitable 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 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 so solved.

We begin with several examples of such reductions.

Example 5.2.1. 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 5.2.2. 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 $\overline{G} = (V, E)$. 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.

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) | G has an independent set of size k\}.

The algorithm $A_{IS}$, for Independent Set, performs 3 steps.

1. On input $(G, k)$, $A_{IS}$ computes the input $(\overline{G}, k)$ for the Clique algorithm, $A_{clique}$.
2. It runs $A_{clique}(\overline{G}, k)$. 
3. It uses the answer from Step 2 to determine its own answer (the same answer is this case).

Demonstrating the algorithm is correct also entails showing:

4. $A_{\text{IS}}$’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 5.2.2.

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

5. Assuming that $A_{\text{Clique}}$ runs in polynomial time, then showing that $A_{\text{IS}}$ also runs in polynomial time.

This is straightforward. For $(G, k)$ can be computed in $O(n^2)$ time, so Step 1 runs in polynomial time. For Step 2, as $A_{\text{Clique}}$ receives an input of size $O(n^2)$ and as $A_{\text{Clique}}$ 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 5.2.3.** Given a polynomial time algorithm for Independent Set there is a polynomial time algorithm for Clique.

**Example 5.2.4.** 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 5.2.5.** Given a polynomial time algorithm for Hamiltonian Circuit (HC) there is a polynomial time algorithm for Hamiltonian Path.

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

1. $A_{\text{HP}}$ builds a graph $H$ with the property that $H$ has a Hamiltonian Circuit exactly if $G$ has a Hamiltonian Path from $s$ to $t$.

2. It runs $A_{\text{HC}}(H)$.

3. It reports the answer given by $A_{\text{HC}}(H)$. 
CHAPTER 5. NP COMPLETENESS

$H$ 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 Circuit. For if $H$ has a Hamiltonian Circuit 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 circuit 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 Circuit $z, v'_1, v'_2, \cdots, v'_n$.

5. Clearly $A_{\text{hp}}$ runs in polynomial time if $A_{\text{hc}}$ runs in polynomial time.

Example 5.2.6. 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 5.2.7. 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 5.2.8. 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. $A_{\text{uhp}}$, the algorithm for the Hamiltonian Path problem, proceeds as follows.

1. $A_{\text{uhp}}$ builds 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. It runs $A_{\text{dbst}}(H, 3)$.

3. It reports the answer given by $A_{\text{dbst}}(H, 3)$.

$H$ 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 $(s'', s), (t'', t)$.

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 $H$. This means that the remaining edges in $T$ form a Hamiltonian Path in $G$ from $s$ to $t$, as each of $s$ and $t$ can have at most one more incident edge, and vertex $v \neq s, t$ can have at most two more incident edges; further, to achieve connectivity, these additional edges must be present.

5. Clearly $A_{\text{hp}}$ runs in polynomial time if $A_{\text{dbst}}$ runs in polynomial time.
5.2. REDUCTIONS AMONG NP PROBLEMS

5.2.1 The Structure of Reductions

We observe that all these constructions have the following form: given a polynomial time algorithm \( A_B \) for membership in \( B \) we construct a polynomial time algorithm \( A_A \) for membership in \( A \) as follows.

Input to \( A_A \): \( I \).

1. Construct \( f_{A \leq B}(I) \) in polynomial time.

2. Run \( A_B(f_{A \leq B}(I)) \).

3. Report the answer from Step 2.

We denote this construction by \( A \leq_P B \), or \( A \leq B \) for short. It is read as: \( A \) can be reduced to \( B \) in polynomial time.

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 \( A_B(f_{A \leq B}(I)) \) runs in time polynomial in \( |I| \), as \( A_B \) 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.

5.2.2 More Examples of Reductions

Example 5.2.9. Undirected Hamiltonian Path (UHP) and Undirected Hamiltonian Circuit (UHC) are the same problems as the corresponding problems for directed graphs, but the new problems are for undirected graphs.

Claim 5.2.10. 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, \( A_{DHP} \). The algorithm proceeds as follows.

1. \( A_{DHP} \) builds the triple \( (H, p, q) \), where \( H \) is an undirected graph, and \( p \) and \( q \) are vertices in \( H \) with the property that

\[ (G, s, t) \in DHP \iff (H, p, q) \in UHP. \]

2. It runs \( A_{UHP}(H, p, q) \).
3. It reports the answer given by \( A_{UHP}(H, p, q) \).

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 \( v^{in}, v^{mid}, v^{out} \), plus the edges \( (v^{in}, v^{mid}), (v^{mid}, v^{out}) \). And for each edge \( (v, w) \in E \), \( H \) receives edge \( (v^{out}, w^{in}) \). See Figure 1 for an illustration. Then \( A_{DHP} \) sets \( p = s^{in} \) and \( q = t^{out} \).

![Figure 5.1: Edges in H corresponding to edge (u, v) in G](image)

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 \( u_1, u_2, \ldots, u_n \), where \( s = u_1, t = u_n \), and \( u_1, u_2, \ldots, u_n \) is a permutation of the vertices in \( V \). Then the following path is a Hamiltonian Path from \( p \) to \( q \) in \( H \).

\[
u_1^{in}, u_1^{mid}, u_1^{out}, u_2^{in}, u_2^{mid}, u_2^{out}, \ldots, u_n^{in}, u_n^{mid}, u_n^{out}\]

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 \( (v^{in}, v^{mid}) \) and \( (v^{mid}, v^{out}) \). Thus the path has the form

\[
v_1^{in}, v_1^{mid}, v_1^{out}, v_2^{in}, v_2^{mid}, v_2^{out}, \ldots, v_n^{in}, v_n^{mid}, v_n^{out}\]

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

This shows the claim.

5. Clearly \( A_{DHP} \) runs in polynomial time if \( A_{UHP} \) runs in polynomial time.

Example 5.2.11. 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 \( U \)?
5.2. REDUCTIONS AMONG NP PROBLEMS

Claim 5.2.12. 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. 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 5.2.13. 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 5.2.14. 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, $A_{3\text{-SAT}}$. The algorithm proceeds as follows.

1. $A_{3\text{-SAT}}$ computes $(G, k)$, where $G$ is a graph and $k$ is an integer, with the property that $F \in 3\text{-SAT} \iff (G, k) \in \text{IS}$.

2. It runs $A_{\text{IS}}(G, k)$.

3. It reports the answer given by $A_{\text{IS}}(G, k)$.

Let $F = C_1 \land C_2 \land \cdots \land C_l$, and suppose that $F$ has variables $x_1, x_2, \ldots, x_n$. $G$ receives the following vertices. For each variable $x$ appearing in $F$ it has vertices $v_x$ and $v_{\overline{x}}$ connected together. For each clause $C_j = a \lor b \lor c$ it has vertices $u_{a_j}, u_{b_j}, u_{c_j}$ connected in a triangle (if the clause is $C_j = a \lor b$ 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, each vertex $u_{a_j}$ is joined to vertex $v_{\overline{a}}$ and each vertex $u_{b_j}$ is joined to $v_x$. See Figure 5.2 for an example. Finally, $A_{3\text{-SAT}}$ sets $k = n + l$.

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

Suppose that $F$ has a satisfying assignment $\sigma$. If $x_i = \text{TRUE}$ in $\sigma$, then $v_{x_i}$ is put in the independent set $I$, while if $x_i = \text{FALSE}$ in $\sigma$, then $v_{\overline{x_i}}$ is put in $I$. This adds $n$ vertices to $I$, and none of these vertices have connecting edges. In addition, for each clause $C_j = a \lor b \lor c$, at least one of its literals must evaluate to \text{TRUE}; without loss of generality, let $a$ be one of these \text{TRUE} literals. Then $u_{a_j}$ is added to $I$. This adds another $l$ vertices to $I$, and none of these $l$ vertices are connected to each other. Suppose that $a = x_i$. Then $u_{a_j} = u_{x_i}$. Thus $u_{x_i}$ is connected to $v_{\overline{x_i}}$ and not to $v_{x_i}$. But
Figure 5.2: The graph for the Independent Set construction for $F = (x_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor \overline{x}_2 \lor \overline{x}_3) \land (x_2 \lor x_3)$. 
$v_{x_i}$ is in $I$. We can conclude that the first $n$ vertices put into $I$ are not connected to any of the second collection of $l$ vertices added to $I$. Consequently, these vertices form an independent set of $n + l$ vertices.

Conversely, suppose that $G$ has an independent set $I$ of size $n + l$. We describe a satisfying assignment $\sigma$ for $F$.

$I$ can include at most one of $v_x$ and $v_{\overline{x}}$ for each $x$, as these vertices are connected. This provides up to $n$ vertices in $I$. Likewise, for each clause $C_j = a \lor b \lor c$, it 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 up to another $l$ vertices in $I$. As no other vertices are available, $I$ must include $n$ of the first group of vertices and $l$ of the second group.

If $v_x \in I$, then $x$ is set to True in $\sigma$, and if $v_{\overline{x}} \in I$ then $x$ is set to False in $\sigma$. Next, we show that $\sigma$ is a satisfying assignment for $F$. Suppose that $u_a^j \in I$ and that $a = x_i$. Then $v_{x_i} \in I$ (as $u_a^j$ is connected to $v_{\overline{x_i}}$) and so $x_i$ was set to True in $\sigma$, causing $C_j$ to evaluate to True; while if $a = \overline{x_i}$, then $v_{\overline{x_i}} \in I$ (as $u_a^j = u_{\overline{x_i}}^j$ is connected to $v_{x_i}$) and so $x_i$ was set to False in $\sigma$, again causing $C_j$ to evaluate to True. As this holds for all $l$ clauses, $F$ evaluates to True when its Boolean assignments are given by $\sigma$.

This shows the claim.

5. Clearly $A_{3\text{-SAT}}$ runs in polynomial time if $A_{\text{IS}}$ runs in polynomial time.
A_G-SAT computes $E$ in two steps. First, it rewrites $F$ as the equivalent Boolean formula $D$ in which all the “nots” are being applied directly to the literals. For example, if $F = \neg[(\neg(x_1 \lor x_2) \land x_3) \lor (x_3 \land \overline{x}_2)]$, then $D = [(x_1 \lor \overline{x}_2) \lor x_3] \land (x_3 \lor x_2)$. $D$ is computed by applying the rules $\neg(x \lor y) = \overline{x} \land \overline{y}$ and $\neg(w \land z) = \overline{w} \lor \overline{z}$, starting at the outermost level. As $D$ always has the same Boolean value as $F$, $F$ is satisfiable exactly if $D$ is satisfiable: $F \in \text{G-SAT} \iff D \in \text{G-SAT}$.

Next $A_{G:\text{-SAT}}$ 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 an expression tree. For the above example, the corresponding expression tree is shown in Figure 5.3.

$E$ 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. For a parent node with operator $\lor$ and corresponding variable $y$, with children having corresponding variables $z_1, z_2, \cdots, z_k$, several clauses are included in $E$, clauses that evaluate to $\text{TRUE}$ exactly if $y = z_1 \lor z_2 \lor \cdots \lor z_k$. The following clauses suffice:

$$B = (\overline{y} \lor z_1 \lor z_2 \lor \cdots \lor z_k) \land (y \lor \overline{z}_1) \land (y \lor \overline{z}_2) \land \cdots \land (y \lor \overline{z}_k).$$

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

- $y$ is set to $\text{TRUE}$; then $\overline{y}$ is $\text{FALSE}$ and one of $z_1, z_2, \cdots, z_k$ must be set to $\text{TRUE}$.
- $y$ is set to $\text{FALSE}$; then all of $\overline{z}_1, \overline{z}_2, \cdots, \overline{z}_k$ are set to $\text{TRUE}$, i.e. all of $z_1, z_2, \cdots, z_k$ are set to $\text{FALSE}$.

Similarly, for a parent node with operator $\land$ and corresponding variable $y$, with children having corresponding variables $z_1, z_2, \cdots, z_k$, the following clauses are put into $E$:

$$B = (\overline{y} \lor z_1) \land (\overline{y} \lor z_2) \land \cdots \land (\overline{y} \lor z_k) \land (y \lor \overline{z}_1 \lor \overline{z}_2 \lor \cdots \lor \overline{z}_k).$$

For if $B$ evaluates to $\text{TRUE}$ then one of the following two situations applies:
5.2. REDUCTIONS AMONG NP PROBLEMS

- $y$ is set to $\text{True}$; then all of $z_1, z_2, \cdots, z_k$ must be set to $\text{True}$.
- $y$ is set to $\text{False}$; then at least one of $\bar{z}_1, \bar{z}_2, \cdots, \bar{z}_k$ is set to $\text{True}$, i.e. at least one of $z_1, z_2, \cdots, z_k$ is set to $\text{False}$.

So $E$ 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 $D \in \text{G-SAT} \iff E \in \text{SAT}$.

First, suppose that $E$ is satisfiable. Let $\tau$ be a satisfying assignment of Boolean values for $E$. Choose the variables in $D$ to receive the Boolean values given by $\tau$. By construction, if $E$ evaluates to $\text{True}$, 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 $\text{True}$ by assumption, so must the clause $(r)$, and consequently $D$ evaluates to $\text{True}$.

Now suppose that $D$ is satisfiable. Let $\sigma$ be a satisfying assignment of Boolean values for $D$. Then the following Boolean assignment causes $E$ to evaluate to $\text{True}$: Use $\sigma$ 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 $\sigma$.

As $D$ evaluates to $\text{True}$ this means that the root of $D$’s expression tree evaluates to $\text{True}$, and hence so does clause $(r)$ in $E$. The remaining clauses in $E$ evaluate to $\text{True}$ exactly if the variable at each parent node $v$ has Boolean value equal to the $\lor$ or $\land$, 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 $\text{True}$.

This shows the claim.

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

Claim 5.2.17. Given a polynomial time algorithm for $3$-Satisfiability ($3$-$\text{SAT}$) there is a polynomial time algorithm for Satisfiability ($\text{SAT}$).

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

1. $\mathcal{A}_{\text{SAT}}$ computes $E$, where $E$ is a CNF formula, with the property that

$$F \in \text{SAT} \iff E \in 3\text{-SAT}.$$ 

2. It runs $\mathcal{A}_{3\text{-SAT}}(E)$. 

3. It reports the answer given by $A_{3\text{-SAT}}(E)$.

$A_{\text{SAT}}$ obtains $E$ as follows. First, any clause containing 3 or fewer literals is simply copied to $E$. Then for a longer clause $C = l_1 \lor l_2 \lor \cdots \lor l_k$, $k > 3$, it puts the following clauses into $E$:

$$D = (l_1 \lor l_2 \lor z_1) \land (\overline{z}_1 \lor l_3 \lor z_2) \land \cdots \land (\overline{z}_{k-4} \lor l_{k-2} \lor z_{k-3}) \land (\overline{z}_{k-3} \lor l_{k-1} \lor l_k).$$

To see why this works it suffices to note that with the same Boolean assignments to the literals $l_1, l_2, \cdots, l_k$, $C$ and $D$ either can both evaluate to True, or neither can evaluate to True. For if $l_i$ is set to True, then setting $z_1, z_2, \cdots, z_{i-2}$ to True, and $z_{i-1}, z_i, \cdots, z_{k-3}$ to False causes $D$ to evaluate to True, while if all of $l_1, l_2, \cdots, l_k$ are set to False, then $D$ reduces to $(z_1) \land (\overline{z}_1 \lor z_2) \land \cdots \land (\overline{z}_{k-4} \lor z_{k-3}) \land (\overline{z}_{k-3})$, which always evaluates to False (for to make the first clause evaluate to True requires $z_1$ to be True, then to make the second clause evaluate to True requires $z_2$ to be True, and so on, causing all of $z_3, \cdots, z_{k-3}$ to be set to True; but then the final clause cannot evaluate to True).

4. Next we argue that $F \in \text{SAT} \iff E \in \text{3-SAT}$. 

First, suppose that $F$ is satisfiable, using Boolean assignment $\sigma$. Then this assignment is applied to $E$ together with the extension to the new variables as described in the previous paragraph. Clearly this causes $E$ to evaluate to True also.

Next, suppose that $E$ is satisfiable, using Boolean assignment $\tau$. The common variables in $F$ are given the same assignment. Now suppose for a contradiction that some clause $C$ in $F$ evaluated to False. As argued above, the corresponding collection of clauses $D$ in $E$ would evaluate to False; but $E$ evaluates to True and so every such $D$ must evaluate to True also. So in fact $C$ evaluates to True also.

This shows the claim.

5. Clearly $A_{\text{SAT}}$ runs in polynomial time if $A_{3\text{-SAT}}$ runs in polynomial time.

We turn to a less obvious reduction.

**Claim 5.2.18.** Given a polynomial time algorithm for Directed Hamiltonian Circuit (DHC) there is a polynomial time algorithm for Vertex Cover (VC): $\text{VC} \leq_P \text{DHC}$.

**Proof.** Let $(G, k)$ be the input to the VC algorithm, $A_{\text{vc}}$. The algorithm proceeds as follows.

1. $A_{\text{vc}}$ computes $H$, where $H$ is a directed graph, with the property that $(G, k) \in \text{VC} \iff H \in \text{DHC}.$

2. It runs $A_{\text{DHC}}(H)$. 

5.2. REDUCTIONS AMONG NP PROBLEMS

3. It reports the answer given by $A_{DHC}(H)$.

Let $G = (V, E)$. $A_{vc}$ 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 5.4(a); this is called an edge gadget. Then, for each vertex $u \in V$, with incident edges $e_1, e_2, \ldots, e_l$, the corresponding vertices, two per edge, are connected as shown in Figure 5.4(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 $s_1, s_2, \ldots, s_k$ 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 5.5.

![Diagram](a)

![Diagram](b)

Figure 5.4: The structure in $H$ due to edge $(u, v)$ in $G$ and to $u$’s adjacency list for edges $e_1, e_2, \ldots, e_l$.

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

First, suppose that $G$ has a vertex cover of size $k$, $U = \{u_1, u_2, \ldots, u_k\}$ say. A preliminary, and incomplete Hamiltonian Circuit $C'$ 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
Figure 5.5: The graph $H$ corresponding to $(G, 2)$. 
5.2. REDUCTIONS AMONG NP PROBLEMS

list for \( u_k \), following which is 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 \( U \).) The vertices not on \( C' \) are \((v, e, 0)\) and \((v, e, 1)\). To include them, it suffices to replace the edge \(((u, e, 0), (u, e, 1))\) with the sequence of three edges obtained by traversing \((u, e, 0), (v, e, 0), (v, e, 1), (u, e, 1)\) 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 \( U = \{u_1, u_2, \ldots, u_k\} \) 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 5.6. 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 5.6(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 5.6(b) or (c)); the latter 4-vertex path is called a zig-zag.

Let \( e_1, e_2, \ldots, e_l \) 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 \((u, e_1, 0)\), \( C \) 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 \( i' = i + 1 \) mod \( k \).

![Figure 5.6: The Three Possible Ways of Traversing the Edge Gadget.](image-url)
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 \( A_{\text{vc}} \) runs in polynomial time if \( A_{\text{dhc}} \) runs in polynomial time.

---

**Claim 5.2.19.** Given a polynomial time for General Satisfiability (G-SAT) there is a polynomial time algorithm for Directed Hamiltonian Circuit (DHC): DHC \( \leq \) SAT.

**Proof.** Let \( G \) be the input to the DHC algorithm, \( A_{\text{dhc}} \). The algorithm proceeds as follows.

1. \( A_{\text{dhc}} \) computes \( F \), where \( F \) is a logical formula, with the property that

\[
G \in \text{DHC} \iff F \in \text{G-SAT}.
\]

2. It runs \( A_{\text{g-sat}}(F) \).

3. It reports the answer given by \( A_{\text{g-sat}}(F) \).

\( A_{\text{dhc}} \) obtains \( F \) as follows. Let \( G = (V, E) \) and \(|V| = n\). \( F \) has \( E \cdot V \) variables \( y^i_{u,v} \), \( 1 \leq i \leq n \), and \((u,v) \in E\). The meaning of a variable is that \( y^i_{u,v} = \text{TRUE} \) if and only if \((u,v)\) is the \( i \)th edge in the Hamiltonian Circuit (on choosing an arbitrary first edge). We write \( F \) as an “and” of subformulas which ensure that \( F \) can evaluate to \text{TRUE} if and only if \( G \) has a Hamiltonian Circuit.

First, \( F^{I_1}_v \) ensures node \( v \) has at least one incoming edge.

\[
F^{I_1}_v = \bigvee_{u : (u,v) \in E} y^i_{u,v}.
\]

\( F^{I_2}_v \) ensures node \( v \) has at most one incoming edge.

\[
F^{I_2}_v = \bigwedge_{t \neq u, (u,v) \in E} \bigl( \overline{y}^i_{u,v} \lor \overline{y}^j_{t,v} \bigr) \land \bigwedge_{u : (u,v) \in E} \bigl( \overline{y}^i_{u,v} \lor \overline{y}^j_{u,v} \bigr) \land \bigwedge_{1 \leq i < j \leq n} \bigl( \overline{y}^i_{u,v} \lor \overline{y}^j_{u,v} \bigr).
\]

Second, there are analogous constraints to ensure \( v \) has exactly one outgoing edge:

\[
F^{O_1}_v = \bigvee_{w : (v,w) \in E} y^i_{v,w}.
\]
5.2. REDUCTIONS AMONG NP PROBLEMS

\[ F_v^{O_2} = \bigwedge_{w \neq x} (\overline{y}_{v,w}^i \lor \overline{y}_{v,x}^j) \land \bigwedge_{w : (v, w) \in E} (y_{v,w}^i \lor \overline{y}_{v,w}^j), \]
\(1 \leq i, j \leq n\)

The constraint \(\land_v (F_v^{I_1} \land F_v^{I_2} \land F_v^{O_1} \land F_v^{O_2})\) if True ensures that every vertex has exactly one incoming and one outgoing edge, and hence that \(G\) has a collection of one or more disjoint cycles that go through every vertex exactly once. To ensure that there is one cycle, subformula \(F^A\) is used to force the \([(i + 1) \mod n]\)th edge to follow the \(i\)th edge, for each \(i, 1 \leq i \leq n\).

\[ F^A = \bigvee_{1 \leq i \leq n} [y_{v,w}^i \Rightarrow \bigvee_{w : (v, w) \in E} y_{v,w}^{(i+1) \mod n}] \]

Note that a formula \(C \Rightarrow D\) can be rewritten as \(\overline{C} \lor D\), so

\[ F^A = \bigvee_{1 \leq i \leq n} [\overline{y}_{v,w}^i \lor \bigvee_{w : (v, w) \in E} \overline{y}_{v,w}^{(i+1) \mod n}] \]

Finally,
\[ F = F^A \land \bigwedge_v (F_v^{I_1} \land F_v^{I_2} \land F_v^{O_1} \land F_v^{O_2}). \]

4. We argue that \(G \in \text{DHC} \iff F \in \text{G-SAT}\).

First, suppose that \(F \in \text{G-SAT}\). Then if \(y_{u,v}^i = \text{TRUE}\), \((u, v)\) is made the \(i\)th edge in the proposed Hamiltonian Circuit for \(G\). The truthfulness of \(F_v^{I_1}\) and of \(F_v^{I_2}\) ensures that exactly one edge into \(v\) is selected, and similarly \(F_v^{O_1}\) and \(F_v^{O_2}\) ensures that exactly one edge out of \(u\) is selected. Thus \(n = |V|\) edges must be selected. Finally \(F^A\) ensures that the head of the \(i\)th edge and the tail of the \((i + 1)\)st edge (mod \(n\)) are the same vertex, ensuring that the edges form a Hamiltonian Circuit.

Second, suppose that \(G \in \text{DHC}\). Let \(v_1, v_2, \ldots, v_n\) be a Hamiltonian Circuit for \(G\). We set \(y_{v_i,v_{i+1}}^i\) to \(\text{TRUE}\), and all other \(y\) variables to \(\text{FALSE}\). This causes \(F_v^{I_1}, F_v^{I_2}, F_v^{O_1}, F_v^{O_2}\) to evaluate to \(\text{TRUE}\) for each \(v \in V\), and causes \(F^A\) to evaluate to \(\text{TRUE}\) also. So \(F\) evaluates to \(\text{TRUE}\).

This shows the claim.

5. Clearly \(A_{\text{HC}}\) runs in polynomial time if \(A_{\text{G-SAT}}\) runs in polynomial time.
5.3 NP-Completeness

We begin by showing that reductions compose.

**Lemma 5.3.1.** 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, $J \leq_P L$. \hfill \Box

**Definition 5.3.2.** A language $L$ is NP-complete ($L \in \text{NPC}$) if

- $L \in \text{NP}$, and
- For every $J \in \text{NP}$, $J \leq_P L$.

**Lemma 5.3.3.** If $K$ is NP-complete and we show that

- $L \in \text{NP}$, and
- $K \leq_P L$

then we can conclude $L$ is NP-Complete also.

*Proof.* It suffices to show that for all $J \in \text{NP}$, $J \leq_P L$. But as $K$ is NP-complete, $J \leq_P K$, and we know $K \leq_P L$. By Lemma 5.3.1, we conclude $J \leq_P L$. \hfill \Box

Lemma 5.3.3 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 5.3.4.** 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 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 5.3.5.** 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 $\in \text{NP}$ since any $\langle V, x, 1^s, 1^t \rangle$ 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 \( L \in \text{NP} \) we argue as follows. As \( L \in \text{NP} \), there are polynomials \( p \) and \( q \), \( L \) has a polynomial time verifier \( V_L \) with \( V_L(x,s) \) running in time at most \( p(|x| + |s|) \), such that each \( x \in L \) has a certificate \( C(x) \) of length at most \( q(|x|) \). Thus \( x \in L \) exactly if \( \langle V, x, 1^q(n), 1^{p(n+q(n))} \rangle \in \text{U-NPC} \).

So the algorithm \( A_L \) to recognize \( L \), given a polynomial time algorithm \( A_{\text{U-NPC}} \) for U-NPC, proceeds as follows.

1. Construct \( \langle V, x, 1^q(n), 1^{p(n+q(n))} \rangle \).
2. Run \( A_{\text{U-NPC}}(\langle V, x, 1^q(n), 1^{p(n+q(n))} \rangle) \).
3. Report the answer from (2).

It remains to argue that \( A_L \) 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 \( |\langle V, x, 1^q(n), 1^{p(n+q(n))} \rangle| = O(p(n + q(n))) \), a polynomial in \( x \). Thus \( \langle V, x, 1^q(n), 1^{p(n+q(n))} \rangle \) can be constructed in time polynomial in \( n \). Consequently, \( L \leq P \) U-NPC, and this is true for every \( L \in \text{NP} \).

### 5.3.1 Characterizing P

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

**Example 5.3.6.** Circuit Evaluation (CE).

Input: \( C \), a circuit comprising \( \text{and} \), \( \text{or} \), and \( \text{not} \) gates, with Boolean inputs \( x_1, x_2, \ldots, x_n \) 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 (\( \text{and} \), \( \text{or} \), or \( \text{not} \)); the latter vertices have one or two inedges as appropriate, but there are no limits on the number of outedges they could have.)

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

**Lemma 5.3.7.** Let \( L \in \text{P} \). Suppose that there is a program \( Q \) that recognizes \( L \), which runs in worst case time \( p(n) \), where \( p \) is a polynomial, and which uses a memory of size \( M(n) \), where each word of memory uses \( O(\log n) \) bits. Then there is a family of circuits \( C_1, C_2, \ldots \) of size \( |C_n| = O(p(n) \cdot M(n) \cdot \log n) \) such that for each \( x \in L \) with \( |x| = n \),

\[
C_n(x) = \text{True} \iff x \in L.
\]

**Proof.** Consider one step of \( Q \)'s computation. This entails reading \( O(1) \) words from memory, computing on them (e.g. adding them, comparing them, or any other standard operation) and then writing \( O(1) \) words to memory. In addition, the program counter, which indicates
the next line of \( Q \) to be executed, is updated (either an increment, or, in the event of a branch, the value of the line number the branch statement indicates). The program is included in the information held in memory. Note that most of the contents of memory are unchanged by this one step of computation.

In principle it is not hard to give a circuit to carry out such a step of computation (after all this is what a computer does and it can be thought of as a very large circuit). In outline, the circuit needs to do the following. First, it selects the statement to execute based on the value of the program counter, PC (the logical formula implementing this is (PC = \( x \)) \( \rightarrow \) (“Chosen statement” = statement \( x \))). Then the circuit implements the chosen statement; it will read the inputs by a method similar to that used for the reading of the statement, carry out the operation (using the appropriate standard circuit for the operation), and write its output by a method similar to that used for the reading of the statement. Finally, it updates the program counter.

Suppose that \( Q \) uses memory locations in the range \([1 \cdots M(n)]\) on inputs of size \( n \) (necessarily \( M(n) \geq n \)), and suppose that each word of memory uses \( w = O(\log n) \) bits (this is a standard assumption). It is not hard to implement the above computation with a circuit of size \( O(M(n) \cdot w + w^2) = O(M(n) \log n) \).

Iterating this circuit \( p(n) \) times allows \( Q \)'s computation on an input of size \( n \) to be simulated. All that remains is to set the circuit’s inputs, which are the initial memory contents, to the values at the start of \( Q \)'s computation. The output of the circuit is simply the value of the Boolean variable that \( Q \) uses to indicate its output (or the trivial translation of the Recognize/Reject output pair to True/False).

There is one more detail to handle. On input \( x \), \( Q \) may run in less than \( p(n) \) steps, but the circuit as constructed computes exactly \( p(n) \) steps of \( Q \)'s computation. To handle this, \( Q \) is modified to perform no-ops once it finishes its computation.

Thus the size of the circuit is at most \( O((p(n)) \cdot M(n) \cdot \log n) \).

**Lemma 5.3.8.** Let \( L \in \mathbb{P} \). Then there is a program \( Q \) recognizing \( L \), which has worst case running time \( q(n) \) for some polynomial \( q \), and which uses only memory locations in the range \([1 \cdots q(n)]\).

**Proof.** Let \( \overline{Q} \) be a program recognizing \( L \) that runs in worst case polynomial time \( p(n) \). Suppose that \( \overline{Q} \) accesses only memory locations in the range \([1 \cdots M(n)]\). Necessarily, \( \overline{Q} \) can access at most \( O(p(n)) \) distinct memory locations in \( p(n) \) steps, so \( \overline{Q} \) can be simulated by another program \( Q \) that stores the memory locations accessed by \( \overline{Q} \) in a balanced binary search tree, indexed by the memory locations. The total memory used by \( Q \) is of size \( O(p(n)) \), and \( Q \) can use consecutive memory locations, for example in the range \([1 \cdots O(p(n))]\). \( Q \) takes time \( O(\log p(n)) = O(\log n) \) to simulate one memory access by \( \overline{Q} \), so \( Q \) runs in time \( O(p(n) \log n) \). Setting \( q(n) = O(p(n) \log n) \) proves the result.

**Theorem 5.3.9.** Let \( L \in \mathbb{P} \). Then there is polynomial \( r(n) \) and a family of circuits \( C_1, C_2, \cdots \) of size \( |C_n| \leq r(n) \) such that for each \( x \in L \) with \( |x| = n \),

\[
C_n(x) = \text{True} \iff x \in L.
\]
Proof. Let $Q$ be the program recognizing $L$ given by Lemma 5.3.8. $Q$ uses memory locations $[1 \cdots q(n)]$, and runs in time $O(q(n))$ for some polynomial $q$. Now apply Lemma 5.3.7 with $M(n) = O(q(n))$, $p(n) = O(q(n))$, and $r(n) = O(p(n) \cdot M(n) \cdot \log n)$.

### 5.3.2 More NPC Problems

**Example 5.3.10.** Circuit Value (CV).

Input: $C$, a circuit comprising $\text{and}$, $\text{or}$, and $\text{not}$ 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 ($\text{and}$, $\text{or}$, or $\text{not}$); the latter vertices have one or two in-edges as appropriate, but there are no limits on the number of out-edges they could have.

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

**Claim 5.3.11.** Given a polynomial time algorithm for General Satisfiability (G-SAT) there is a polynomial time algorithm for Circuit Value (CV): $\text{CV} \leq \text{G-SAT}$.

**Proof.** Let $C$ be the input to the CV algorithm, $A_{CV}$. The algorithm proceeds as follows.

1. $A_{CV}$ computes $F$, where $F$ is a logical formula, with the property that

$$C \in \text{CV} \iff F \in \text{G-SAT}.$$

2. It runs $A_{G-SAT}(F)$.

3. It reports the answer given by $A_{G-SAT}(F)$.

$A_{CV}$ constructs $F$ using the same method as in Claim 5.2.16. It first pushes all the not operations to the “leaves” of the circuit (the vertices with no in-edges). Then it creates a new variable for each vertex of the circuit labeled by a logical operator, and for each such vertex it creates a collection of clauses that evaluate to True exactly if the inputs and output of the operator are as specified by the operator (e.g. for an $\text{and}$, the output is the $\text{and}$ of the inputs). Finally, it also includes the clause $(y)$.

4. As argued in Claim 5.2.16, $F$ can be satisfied exactly if $C$ can be satisfied.

5. Clearly $A_{CV}$ runs in polynomial time if $A_{G-SAT}$ runs in polynomial time.

**Theorem 5.3.12.** (Cook’s Theorem, 1972). $\text{CV}$ is NP-Complete.
Proof. Let \( L \in \text{NP} \). We need to show that \( L \leq_P \text{CV} \). That is, given a polynomial time algorithm for \( \text{CV} \), we give a polynomial time algorithm for \( L \).

As \( L \in \text{NP} \), we know that \( L \) has a polynomial time verifier, \( V_L \) say. By Theorem 5.3.9, there is a polynomial \( r \), a circuit family \( C_1, C_2, \ldots \), 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 outputs 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 \( \text{CV} \) problem, and in addition, \( x \in L \) exactly if \( C(x, n) \in \text{CV} \).

Thus the algorithm \( A_L \) to recognize \( L \) proceeds as follows.

1. \( A_L \) builds circuit \( C(x, n) \).
2. \( A_L \) runs \( A_{\text{CV}}(C(x, n)) \).
3. \( A_L \) reports the answer given by \( A_{\text{CV}}(C(x, n)) \).

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

A more formal proof, as was given by Cook, could be based on a Turing Machine based definition of polynomial time. Then a polynomial time verification algorithm can be instantiated as a polynomial time Turing Machine. A corresponding circuit, or in this case a G-SAT formula \( F \), is written, where \( F \) can evaluate to \( \text{TRUE} \) exactly if there is a certificate for the given input and verification algorithm. The details of \( F \) are similar to those of the formula in the proof of Claim 5.2.19.

**Corollary 5.3.13.** \( \text{GSAT, SAT, 3-SAT, VC, Clique, IS, DHC, UHC, DHP, UHP} \) are all \( \text{NP-Complete} \).

**Proof.** We have shown all these languages are in \( \text{NP} \). Cook’s Theorem, Lemma 5.3.1, and the individual reductions already shown yield the claimed result. \( \square \)

**Definition 5.3.14.** Language \( L \) is \( \text{NP-hard} \) if for all \( K \in \text{NP} \), \( K \leq_P L \).

**Lemma 5.3.15.** If \( J \) is \( \text{NP-hard} \) and \( J \leq_P L \), then \( L \) is \( \text{NP-hard} \).

**Proof.** This is proved in the same way as Lemma 5.3.3. \( \square \)
5.4 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.

5.4.1 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.

5.4.2 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 5.4.1.** *Degree 4 Directed Hamiltonian Circuit (4-DHC)* is the Directed Hamiltonian Circuit problem on directed graphs where each vertex has degree (indegree plus outdegree) bounded by 4.

**Claim 5.4.2.** 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 $A_{\text{DHC}}$, 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 $e_1, e_2, \ldots, e_k$, $H$ receives cycles $C_h, C_{h-1}, \ldots, C_2$ of $2^h, 2^{h-1}, 2^{h-2}, \ldots, 4$ vertices respectively, as illustrated in Figure 5.7. Each edge $e_i = (u, v)$ is replaced by an edge $e_i' = (u', v')$; the new edges are incident on 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 $v$.

![Figure 5.7: The Inedge Gadget for the 4-DHC Construction.](image)

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 Circuit $C$. 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 $2^h$-th 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 $H$. 
5.4. REDUCTION TECHNIQUES

Second, suppose that $H$ has a Hamiltonian Circuit $D$. 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 the path from $v_i$ to $v_{i+1 \mod n}$ in $D$ must go directly from the “out” vertices for $v_i$ to the “in” vertices for $v_{i+1 \mod n}$ and hence must use edge $(v'_i, v'_{i+1})$, which implies that $(v_i, v_{i+1 \mod n})$ is an edge of $G$.

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

5.4.3 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 $\text{DHP} \leq_P \text{UHP}$ and $\text{SAT} \leq_P \text{3-SAT}$. The latter is also an instance of restriction.

Example 5.4.3. 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 5.4.4. 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 to be $k + 1$. An example of graph $H$ is shown in Figure 5.8.

![Figure 5.8: The Vertex Cover to Dominating Set Reduction; An Example Graph.](image-url)
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 vertex cover $U'$ 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 $(G, k) \in \text{VC} \iff (H, h) \in \text{DS}$. 

5.4.4 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. Again, we have seen examples of such reductions, including 3-SAT $\leq_P \text{IS}$ and VC $\leq_P \text{DHC}$. 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.