Chapter 2

Basic Concepts

Chapter Outline: In this chapter, we discuss some basic concepts. We define reductions, NP-completeness, alternations, relativizations, leading to the Polynomial-time Hierarchy.

2.1 P and NP

In Chapter 1, we introduced deterministic and non-deterministic Turing machines and their complexity bounded versions. Building on these notions, we defined deterministic and non-deterministic time complexity classes. We continue the discussion by first recalling these classes.

Definition 2.1 *For any time-constructible function $f(n)$,*

$$\begin{aligned} \text{DTIME}[f] &= \{L| \textit{ for some deterministic TM } M, \textit{ } L = L(M) \textit{ and } M \textit{ runs in time } f(n)\} \\ \text{NTIME}[f] &= \{L| \textit{ for some non-deterministic TM } M, \textit{ } L = L(M) \textit{ and } M \textit{ runs in time } f(n)\} \end{aligned}$$

Since every deterministic TM can also be viewed as a non-deterministic TM, we have the following proposition.

Proposition 2.2 *For any function $f(n)$,* $\text{DTIME}[f] \subseteq \text{NTIME}[f]$.

We obtain the two central complexity classes, P and NP by considering deterministic and non-deterministic TMs that run in polynomial time:

Definition 2.3

$$\begin{aligned} \text{P} &= \bigcup_{f \in \text{Poly}} \text{DTIME}[f] \\ \text{NP} &= \bigcup_{f \in \text{Poly}} \text{NTIME}[f] \end{aligned}$$

where Poly is the set of all polynomials.

It follows from these definitions that P $\subseteq$ NP, and perhaps the greatest open problem in computer science asks whether this containment is in fact proper, namely NP $\neq$ P.

2.2 NP-Completeness

One of the most useful notions in complexity theory is that of completeness. In general, one can define complete languages for any complexity class. This section is devoted to a discussion of complete languages for the class NP. We start by defining another important concept called reduction.

DEFINITION 2.4 (KARP REDUCTION) *A language L_1 reduces to another language L_2 by a Karp reduction, denoted as $L_1 \leq_m^p L_2$, if there is a function $f : \Sigma^* \to \Sigma^*$, such that*

1. *$x \in L_1 \iff f(x) \in L_2$, and*
2. *f is deterministic polynomial time computable.*

Karp reduction is also known as *polynomial time many-one reduction*, which is the polynomial time version of the recursion theoretic notion of *many-one reduction*. There are other notions of reduction which will be discussed later. We now proceed to define the notion of NP-completeness.

DEFINITION 2.5 *A language L is said to be* NP*-hard (under Karp reductions) if every language $L' \in$ NP Karp reduces to L. L is said to be* NP*-complete if $L \in$ NP and it is* NP*-hard.*

We next exhibit a canonical NP-complete language A_{NP}. It is defined as follows.

$$A_{\text{NP}} = \{\langle M, w, 1^t\rangle |\ M \text{ is a non-deterministic TM that accepts } w \text{ within } t \text{ steps}\}$$

PROPOSITION 2.6 *A_{NP} is* NP*-complete.*

Proof. It is easy to see that A_{NP} is in NP. Given $\langle M, w, 1^t\rangle$ as input, we simulate M on w (non-deterministically) for t steps and *accept* the input if M accepts w. Clearly, this algorithm runs in non-deterministic polynomial time.

We next show that A_{NP} is NP-hard. Let $L \in$ NP via a non-deterministic TM M that runs in time $O(n^c)$, where c is some constant. Our reduction algorithm, given input w, simply outputs $\langle M, w, 1^{n^c}\rangle$. It is clear that $w \in L$ iff $\langle M, w, 1^{n^c}\rangle \in A_{\text{NP}}$. Our reduction algorithm takes (deterministic) time $O(n^c)$, and hence runs in polynomial time. ♣

We note that this proof can be easily adapted to many other complexity classes, as long as the class has an enumeration by "clocked" machines. For example, for the class PSPACE, we can define an enumeration of space bounded (by $n^j + j$) TMs $M_{\langle i,j \rangle}$, and form such a "universal language", which will be complete for PSPACE.

$$A_{\text{PSPACE}} = \{\langle M, w, 1^t \rangle |\ M = M_{\langle i,j \rangle} \text{ accepts } w \text{ within } t = |w|^j + j \text{ space}\}$$

Thus mere existence of complete languages for a standard complexity class which can be represented by an enumeration of TMs is no surprise. However these are not "natural" languages. Having shown the existence of an NP-complete problem, we turn our attention to "natural" or "real-world" problems that are NP-complete. The great importance of NP-completeness resides with these "natural" NP-complete problems. One such famous problem is SAT.

$$\text{SAT} = \{\varphi | \varphi \text{ is a satisfiable boolean formula } \}$$

The famous Cook–Levin theorem states that SAT is NP-complete.

THEOREM 2.7 (COOK–LEVIN) SAT *is* NP*-complete.*

The proof goes by showing that, given a polynomial time NTM M and an input w, the computation of the machine M on w can be encoded into a boolean formula so that M accepts w if and only if the formula is satisfiable. Moreover, if the machine runs in polynomial time, the length of formula will also be polynomial in the input length. The basic idea is to use Boolean variables and logical connectives $\vee, \wedge, \neg$ to form a propositional logic formula, which is in CNF (a conjunction of some disjunctions of literals), which "says" the computation $M(x)$ accepts for some non-deterministic computational path. For a formal proof of the theorem, we refer the reader to standard textbooks [2, 3, 4]. It really should come as no surprise that boolean logic can express this. But the impact of realizing SAT is such a universal problem for NP is tremendous.

Following the work of Cook, Karp showed that many other "natural" problems are also NP-complete. It is with the realization of this abundance of "natural" NP-complete problems that the importance of NP-completeness is truly demonstrated. Apart from its theoretical importance, the notion of NP-completeness is also very useful from a practical perspective. If an NP-complete problem is shown to be solvable in polynomial time, then every problem in NP would also be solvable polynomial time, and hence, it would imply that NP = P. But, it is widely believed that NP $\neq$ P. One indication is perhaps that there are thousands of NP-complete problems and no poly-time algorithm has been found for any of them. (Although this is admittedly a weak argument in favor of the Conjecture.) So, showing a problem to be NP-complete is taken to be a strong indication that the problem is not solvable in polynomial time and hence computationally hard. In fact, following this scheme, over the past three decades, thousands of "practical" problems have been shown to be NP-complete. A book by Garey and Johnson [2], now a classic, includes a compendium of many of these NP-complete problems.

Going back to SAT, we saw that, given a boolean formula, it is NP-hard to decide whether it is satisfiable or not. In this formulation, we allow the input to be any arbitrary boolean formula. But in fact Cook's proof shows that it remains NP-hard, even if we restrict the input formula to be in conjunctive normal form (CNF) (i.e. the formula is an AND of ORs such as $(\overline{x_1} \vee \overline{x_2} \vee x_3 \vee x_4) \wedge (x_1 \vee \overline{x}_6)$). It is common to refer to the above version (CNF-SAT) simply as SAT.

Once we have established a certain problem Π to be NP-complete, to prove the NP-completeness of another problem Π' in NP, we can reduce the known NP-complete problem Π to the given problem Π'. This is simply a consequence of the transitivity of Karp reductions. The task is made easier if we start with a problem known to be NP-complete, and it has as restrictive a form as possible. For example, it is easier to give a reduction from the CNF version of SAT than from the general one. Proceeding along these line, we next show that an even more restricted version of SAT, namely 3-SAT, is also NP-complete.

3-SAT is the set of all satisfiable boolean expressions of the form

$$\varphi = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{3} l_{i,j}$$

where the literal $l_{i,j}$ is either x or $\overline{x}$, for some input variable x. Thus, $(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3 \vee x_4)$ is an example of a 3-SAT expression. We now reduce SAT to 3-SAT.

In order to transform SAT to 3-SAT, we use a construction termed the "butterfly construction". The idea is that for any clause that is longer than three components, we split the clause into clauses with 3 literals, by adding extra variables. Consider the following formula:

$$\varphi = (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6)$$

We simply split this formula into 3-clauses as follows:

$$\begin{aligned} \varphi &\rightarrow (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee x_4 \vee x_5 \vee x_6) \\ &\rightarrow (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge (\overline{y}_2 \vee x_4 \vee x_5 \vee x_6) \\ &\rightarrow (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge (\overline{y}_2 \vee x_4 \vee y_3) \wedge (\overline{y}_3 \vee x_5 \vee x_6) \end{aligned}$$

Call the last formula φ'. Now we notice the following: If we have assigned all x_i to be false in φ, then in order to satisfy φ', we can reason from left to right in φ' one clause at a time, and we see that all y_j must be true: x_1, x_2 are false, so y_1 is true; then $\overline{y}_1, x_3$ are false, and so y_2 is true; etc. But still this would leave the last clause false. (Imagine an air bubble being pushed from left to right.)

On the other hand, if we have assigned some x_i to be true, then we can "propagate" from this x_i on both sides by assigning each y_j appropriately so that all 3-literal clauses are satisfied. For example suppose x_3 is true. Then it looks something like the following, where $\uparrow$ represents true and $\downarrow$ represents false for the whole literal (including the not symbol):

This is the motivation for calling it the butterfly construction.

y_1	$\overline{y}_1$	x_3	y_2	$\overline{y}_2$	y_3	$\overline{y}_3$
↑	↓	↑	↓	↑	↓	↑

Given any instance of SAT in CNF, we apply the same transformation to each clause with length greater than 3 by adding new variables for each clause. If a clause has fewer than 3 literals, there is a trivial transformation to make a conjunction of 3-clauses equivalent to it. Then it is clear that the original formula is satisfiable iff the transformed formula is satisfiable.

We have proved the following theorem.

THEOREM 2.8 *3-SAT is* NP*-complete.*

One can also easily prove that the problem remains NP-complete if we restrict to SAT instances with exactly k literals per clause, for any $k \geq 3$. This is called k-SAT problem. It is interesting to note that 2-SAT is decidable in polynomial time as it can be formulated as a graph of logical implications and one can find if the implications are consistent or not in polynomial time.

Exercise: Show that 2-SAT is in P.

With the NP-completeness of 3SAT in hand, we can show that many other problems are NP-Complete by reducing 3-SAT to them.

2.2.1 NP-completeness of VERTEXCOVER, INDEPENDENTSET, CLIQUE

We begin by defining VERTEXCOVER, or VC:

$$\text{VC} = \{\langle G, k\rangle \mid \exists S \subseteq V(G), |S| \leq k, \text{ every edge of } G \text{ is incident to at least one vertex of } S\}$$

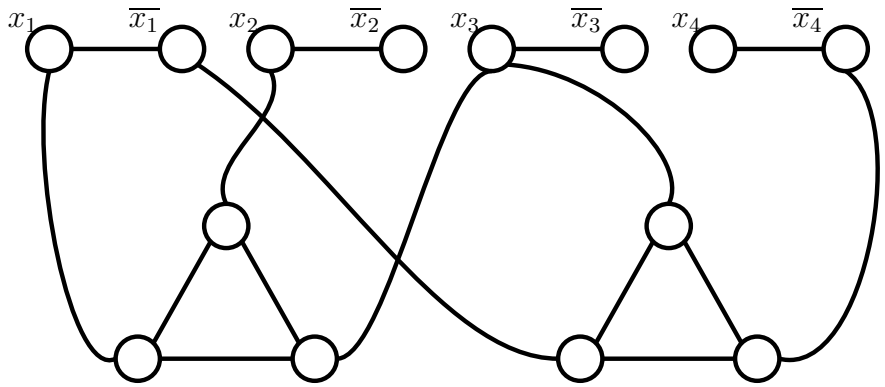
We now utilize a gadget-based construction to reduce SAT to VERTEXCOVER. The idea of such a construction is to create objects in one system (in this case, we'll be creating specialized graphs) that correspond to the fundamental objects in another (clauses and variables), such that a certain relation among the constructed objects in the first system exists if and only if another relation exists among all objects in the second. In the case of the construction for VC, we want certain vertices to be covered iff the assignments to variables they represent correspond to a satisfying assignment.

Now, we give the construction. We reduce 3SAT to VC. We will find the limitation on the size of the clauses to be advantageous here. Assume there are n variables $x_1, x_2, \ldots, x_n$ and m clauses. First, we construct vertices $x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n}$. These vertices will correspond to the possible truth assignments to the n variables present in the formula. We add edges $(x_1, \overline{x_1}), (x_2, \overline{x_2}), \ldots, (x_n, \overline{x_n})$, which forces the constraint that we must select at least one value for each variable. We call these the intra-variable edges.

Then, we construct vertices $\hat{l}_{11}, \hat{l}_{12}, \hat{l}_{13}, \hat{l}_{21}, \hat{l}_{22}, \hat{l}_{23}, \ldots, \hat{l}_{m1}, \hat{l}_{m2}, \hat{l}_{m3}$, which correspond to the literals of each clause. These nodes are connected in a triangle for each clause: $(\forall i, 1 \leq i \leq m)\{(\hat{l}_{i1}, \hat{l}_{i2}), (\hat{l}_{i2}, \hat{l}_{i3}), (\hat{l}_{i3}, \hat{l}_{i1}))\}$. We call these the intra-clause edges.

Finally, each literal node $\hat{l}_{ij}$ in a clause is connected to its corresponding vertex l_{ij}. For example, if l_{ij} is x_k, we have $(\hat{l}_{ij}, x_k)$; and similarly, if l_{ij} is $\overline{x_k}$, it will be $(\hat{l}_{ij}, \overline{x_k})$. We call these the inter-formula edges.

Here is how $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_4})$ would be represented:



Now we'll step back and reason about our construction. Suppose the formula represented is satisfiable. Then we can construct a vertex cover of size $n + 2m$. We pick the n vertices corresponding to a satisfying truth assignment σ. (If x_1 is set to "true" in σ, include the vertex labelled x_1 in the vertex cover, else include the one labelled $\overline{x_1}$.) Then, in each triangle corresponding to a clause, we cover two literals which include all literals which are set false. Note that if the formula is satisfied under σ, then any clause has at least one literal which is true, hence at most two literals which are false. With one vertex for each pair of x_i and $\overline{x_i}$, and 2 vertices in each triangle, we note that all of the intra-variable and intra-clause edges will be covered no matter what the assignment. We note also that all the inter-formula edges are covered as well, if σ is a satisfying assignment: An inter-formula edge connecting a variable node to a satisfied literal is covered by the variable node; an inter-formula edge connecting a variable node to an unsatisfied literal is covered by the literal node in the triangle for the clause. So we have a vertex cover of $n + 2m$ vertices.

Conversely, suppose there exists a covering with $n + 2m$ vertices. We show that this covering must represent a satisfying assignment. In order to cover all of the intra-variable and intra-clause edges, we need at least $n+2m$ vertices covered, 1 for each variable and 2 for each clause. Therefore these $n+2m$ vertices consist of exactly one for each pair $(x_i, \overline{x_i})$, and exactly two for each triangle. But this implies that only 2 of the 3 inter-formula edges for each clause are covered by literal nodes in the triangle for the clause, the third inter-formula edge must be covered by the node representing the literal. Hence, if we assign to true any literal precisely when it is in the vertex cover, the formula is satisfied.

Therefore, the formula is satisfiable iff there exists a cover of size $n+2m$. We have proved

THEOREM 2.9 *VC is* NP*-complete.*

We also note that INDEPENDENTSET is NP-Complete. First we define the problem:

$$\text{IS} = \{\langle G, \ell \rangle \mid \exists S \in V(G), |S| \geq \ell \text{ such that } \forall u, v \in S : (u, v) \notin E(G)\}$$

Observe that, if C is a vertex cover of a graph G, then its complement $V - C$ is an independent set, and vice versa. Thus, a graph has a vertex cover of size $\leq k$ if and only if it has an independent set of size $\geq n - k$. We conclude that INDEPENDENTSET is NP-complete. They are essentially the same problem.

THEOREM 2.10 *IS is* NP*-complete.*

Finally, we note that CLIQUE is NP-Complete. CLIQUE is also essentially the same problem as INDEPENDENTSET:

$$\text{CLIQUE} = \{\langle G, \ell \rangle \mid \exists S \in V(G), |S| \geq \ell \text{ such that } \forall u, v \in S : (u, v) \in E(G)\}$$

As such, by flipping the edges of G (construct a G' such that $V(G') = V(G)$ and $(u, v) \in E(G') \iff (u, v) \notin E(G)$), we see that by the VC reduction, CLIQUE is also NP-Complete.

THEOREM 2.11 CLIQUE *is* NP*-complete.*

2.2.2 HAMILTONIANCIRCUIT

Another graph problem we can define is HAMILTONIANCIRCUIT, or HC. We define a Hamiltonian circuit to be an ordering of the vertices of a graph G as $v_1, v_2, \ldots, v_n$, such that we can travel through all the vertices once and return to where we started: $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1) \in E(G)$. Then,

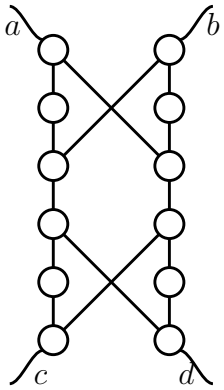
$$\text{HC} = \{\langle G \rangle \mid \exists \text{ a Hamiltonian circuit in } G\}$$

It is straightforward that HC is in NP, as we can verify that an ordering of the vertices represents a cycle in polynomial time.

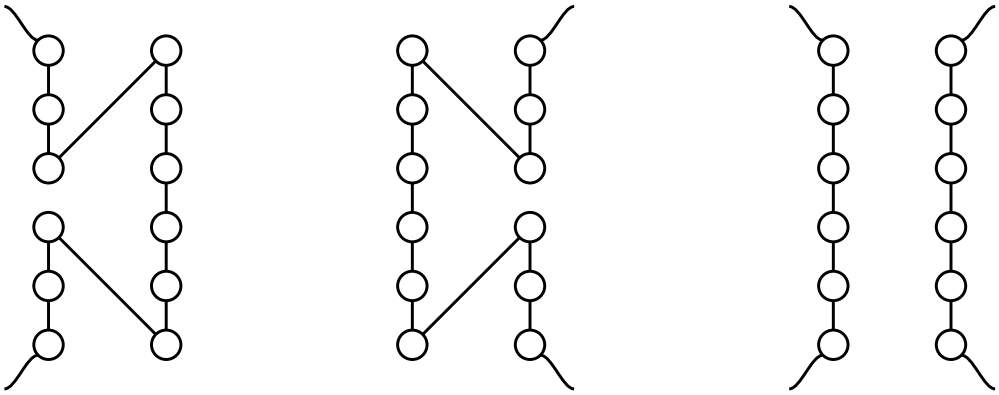
Reduction from VC

What follows is a proof of the NP-Completeness of HC. As with VC, we will utilize a gadget-based construction. However, instead of reducing from 3SAT, we will reduce from VC.

Consider the following graph, which we will refer to as our gadget:



There are only three ways to traverse this graph (which can be verified through inspection): (1) An *a*-*b*-*d*-*c* pass— enter through a and exit through c, passing through b and d; or (2) a *b*-*a*-*c*-*d* pass— enter through b and exit through d; or (3) traverse in two passes: first in an *a*-*c* pass, and then in a *b*-*d* pass. In all cases reordering of entry and exit is possible. Here are the three graphs corresponding to these paths:



Given G, an instance for VC, we construct G', an instance for HC. For each edge $e = (u, v)$ in G, we use one such gadget in G' representing the edge e. The left half of this gadget represents u and the right half represents v. For each vertex u, we link together in some arbitrary order, top to bottom, all of the u halves of the gadgets that represent edges incident to v. (We'll refer to this as a *linked vertex chain*.) Finally, let k be the size of the VC in question, then we add vertices $a_1, \ldots, a_k$ and connect each of them to both the start and end of each linked vertex chain.

The intention is to represent how the vertices are covered in G by the traversals of each linked vertex chain emanating from $a_1, \ldots, a_k$ in G'. In the above pictures, if u is one of the k vertices in the VC, then for each gadget on a linked vertex chain for the vertex u, we will traverse either all the vertices (as in the first choice of traversal) when (u, v) is covered by u alone, or half of the vertices (as in the third choice of traversal) when (u, v) is covered by both u and v. If u is not in the VC, but v is, then all the vertices of this gadget will be traversed as in the middle choice of traversal.

We now give a short argument for the correctness of this construction. First, we show how to traverse the graph if there is a vertex cover of size k. We note that if there is a cover of size less than k, it can be easily extended into a cover of size exactly k simply by adding extra vertices to the optimal cover, as they cannot hurt. So we assume $k \leq n$ and have a cover of size k. For the ith vertex (say u) in the cover, we start at a_i, trace out the linked vertex chain for u before returning to a_{i+1}, and if $i = k$ we return back to a_1, completing the tour. For each linked vertex chain, say of u, we traverse each gadget corresponding to an edge (u, v) as follows. If the edge is covered by u alone, then we traverse in an *a-b-d-c* pass (where the *a-c* side corresponds to u), and if the edge is covered by both u and v, then we traverse in an *a-c* pass. Clearly, from a VC of size k we get a Hamiltonian Circuit.

Now, we claim that if we have a cycle for the graph G', there is a vertex cover of size k in the original graph G. If we had a cycle, it would have to include all of the a_i vertices and all of the gadget vertices. But we know that in between each of the visits to the a_i vertices, we can only visit one linked vertex chain corresponding to one vertex u in G. We claim that these k vertices form a vertex cover in G. For each gadget on this linked vertex chain of u we can see by the way it was visited which edges the vertex u covers in G. By the way how each gadget can be traversed, it follows that indeed these k vertices form a vertex cover in G.

THEOREM 2.12 *HC is* NP*-complete.*

2.3 Polynomial Hierarchy

We defined NP to be the class of all languages accepted by non-deterministic polynomial time TMs. We can give an equivalent characterization of NP.

DEFINITION 2.13 (Σ_1^p) *A language L is in Σ_1^p if there exists a boolean binary predicate $D(\cdot, \cdot)$ computable in (deterministic) polynomial time, and a polynomial q such that, for all input x,*

$$x \in L \Longleftrightarrow \exists y \in \{0,1\}^{q(|x|)}[D(x, y) = 1]$$

It is easy to see that NP $= \Sigma_1^p$. Suppose $L \in \Sigma_1^p$. We design an NP machine that given an input x, simply guesses a string y of length $q(|x|)$ and then accepts if $D(x, y) = 1$. On the other hand, suppose $L \in$ NP, via an NP machine M that runs in time $q(n)$. Then, for any input x,

$$x \in L \iff (\exists \text{ a computational path } p \text{ of length } q(|x|)\ )[\ M \text{ accepts } x \text{ along the path } p\ ]$$

Clearly, $L \in \Sigma_1^p$.

In the above formulation of NP, the predicate D is usually called a *verifier* and if $D(x, y) = 1$, then y is called a *certificate* or *proof* for the input x. For example, in the

case of SAT, we can construct a verifier as follows. Given a formula φ and a truth assignment σ, accept (φ, σ), if σ satisfies φ. Any satisfying truth assignment of φ is a certificate.

We can generalize the definition of Σ_1^p to define a larger class as follows.

DEFINITION 2.14 (Σ_2^p) *A language L is in Σ_2^p if there exists a boolean predicate D computable in polynomial time, such that*

$$x \in L \Longleftrightarrow \exists y \forall z [D(x, y, z) = 1]$$

where $|y|$ and $|z|$ are polynomially bounded in $|x|$.

We will denote by $\exists^p y$ for $\exists y$ where $|y|$ is polynomially bounded in $|x|$. By padding we may assume there exists some polynomial $q(\cdot)$, such that $\exists^p y$ means $\exists y \in \{0,1\}^{q(|x|)}$. Again by padding we may assume the same polynomial $q(\cdot)$ works for $\exists^p y$ and $\forall^p z$.

We next give a natural example of a language in Σ_2^p. We say that a boolean formula φ is minimal if there are no shorter formulas that compute the same boolean function. For example, the formula $(x_1 \vee x_2) \vee (x_1 \vee x_3)$ is not minimal, because it has a smaller equivalent formula: $x_1 \vee (x_2 \vee x_3)$. Let MEE (Minimum Equivalent Expression) represent the set of all minimal boolean formulas, i.e., a boolean expression for which there are no shorter boolean expressions equivalent to it. Then the language $\overline{\text{MEE}}$, the complement of MEE, is a language in Σ_2^p. We can express this language as,

$$\varphi \in \overline{\text{MEE}} \iff \exists \psi \forall \sigma [|\psi| < |\varphi| \text{ and } \varphi(\sigma) = \psi(\sigma)]$$

In the above definition, σ refers to various truth assignments and $\varphi(\sigma)$ is the value of the formula for the truth assignment σ.

DEFINITION 2.15 (Π_2^p) *If L is in Σ_2^p, we say that L^c, the complement of L, is in Π_2^p. Equivalently, L is in Π_2^p if there exists a boolean predicate Q computable in polynomial time, such that*

$$x \in L \Longleftrightarrow \forall y \exists z [Q(x, y, z) = 1]$$

where $|y|$ and $|z|$ are polynomially bounded in $|x|$.

A natural example for the class Π_2^p is MEE, the set of all minimal propositional boolean formula. We can express MEE as,

$$\varphi \in \text{MEE} \Longleftrightarrow \forall \psi \exists \sigma [|\psi| < |\varphi| \longrightarrow \psi(\sigma) \neq \varphi(\sigma)]$$

We extend the above definitions to define the classes Σ_k^p and Π_k^p. For any integer $k \geq 0$, we define

DEFINITION 2.16 (Σ_k^p AND Π_k^p) *A language L is in Σ_k^p if there exists a $(k+1)$-ary boolean predicate D computable in polynomial time, such that*

$$x \in L \Longleftrightarrow \exists^p y_1 \forall^p y_2 \ldots Q^p y_k [D(x, y_1, y_2, \ldots, y_k) = 1]$$

where Q^p is either $\exists^p$ for k odd or $\forall^p$ for k even, and the superscript p denotes that all $|y_i|$ are polynomially bounded in $|x|$. In the above definition, $\exists$ and $\forall$ alternate. We say that a language L is in Π_k^p, if its complement L^c is in Σ_k^p.

It is clear that

LEMMA 2.17 *For any integer $k \geq 0$,*

$$\Sigma_k^p \cup \Pi_k^p \subseteq \Sigma_{k+1}^p \cap \Pi_{k+1}^p.$$

DEFINITION 2.18 (POLYNOMIAL TIME HIERARCHY (PH)) PH$= \bigcup_k \Sigma_k^p = \bigcup_k \Pi_k^p$

A picture of the polynomially time hierarchy is shown in Figure 2.1.

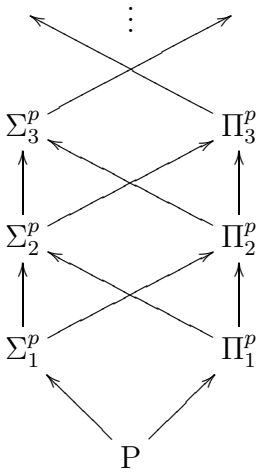


Figure 2.1: Polynomial Time Hierarchy

It is a big open problem whether the polynomially hierarchy is infinite, i.e., whether it has infinitely many distinct levels. It is quite conceivable that the hierarchy collapses to some k^{th} level, meaning, for all $r \geq k$, $\Sigma_r^p = \Sigma_k^p$. Then as we shall see, equivalently, PH $= \Sigma_k^p$. But, most people believe that it is infinite. One can prove a simple result related to collapsing the hierarchy.

THEOREM 2.19 *If $\Sigma_k^p = \Pi_k^p$ for some k, then* PH$= \Sigma_k^p = \Pi_k^p$ *(the polynomial time hierarchy collapses to the level k.)*

Comment: We note that $\Sigma_k^p = \Pi_k^p$ implies, and is implied by $\Sigma_k^p \subseteq \Pi_k^p$, as well as $\Pi_k^p \subseteq \Sigma_k^p$. For, say $\Sigma_k^p \subseteq \Pi_k^p$ and $L \in \Pi_k^p$, then $L^c \in \Sigma_k^p$. Thus $L^c \in \Pi_k^p$. Hence $L = (L^c)^c \in \Sigma_k^p$.

Proof. To get an idea of the proof, let us suppose that $\Sigma_1^p = \Pi_1^p$. We want to prove that $\Sigma_2^p \subseteq \Sigma_1^p$. Then by Lemma 2.17, $\Sigma_2^p \subseteq \Pi_2^p$, and thus $\Sigma_2^p = \Pi_2^p$, by the comment above.

Consider a language $L \in \Sigma_2^p$. Now for a string x,

$$x \in L \iff \exists^p y \forall^p z P(x, y, z)$$

Rewrite this in terms of another predicate $Q(\langle x, y\rangle, z)$, where Q first "unravels" the $\langle x, y\rangle$ into x and y and calls P. Hence, we have

$$x \in L \iff \exists^p y \forall^p z Q(\langle x, y\rangle, z)$$

Observe however, that $\{\langle x, y\rangle | \forall^p z Q(\langle x, y\rangle, z)\}$ is a language in Π_1^p, which is equal to Σ_1^p by our assumption. Hence we can write the language as $\{\langle x, y\rangle | \exists^p z E(\langle x, y\rangle, z)\}$ for some other polynomial deterministic predicate E. So, we have,

$$x \in L \iff \exists^p y \exists^p z E(\langle x, y\rangle, z)$$

which can be rewritten as

$$x \in L \iff \exists^p \langle y, z\rangle E'(x, \langle y, z\rangle)$$

where the predicate E' does the following: it first unravels the $\langle y, z\rangle$ into y and z, and then pairs x and y to form $\langle x, y\rangle$. It then calls E with $\langle x, y\rangle$ and z. This shows that L is a language in Σ_1^p, and hence, $\Sigma_2^p \subseteq \Sigma_1^p$.

In the general case, a similar proof works by induction that $\Sigma_k^p = \Pi_k^p$ implies that $\Sigma_{k+1}^p = \Pi_{k+1}^p$. ♣

The notion of an oracle plays an indispensable role in complexity theory. Next, we discuss oracles. Later we shall present an equivalent characterization of PH using oracles.

2.4 Oracle Turing Machines

An *oracle Turing machine* is a multi-tape Turing machine with a special tape called the query tape. The TM also has three special states called $q_?$ (the query state), q_{yes} and q_{no} (the answer states). Let A be a predetermined language. The computation of an oracle TM with oracle A is defined as follows. During its computation, the TM may write a string z on the query tape and enter the query state. In the next step, the machine will enter the state q_{yes} if $z \in A$ and will enter the state q_{no} if $z \notin A$. Informally, whether or not the query string z is in A is determined automatically. We say that the TM asks the query to an oracle for A and the oracle "magically" gives the correct answer in one step. Aside from this ability to ask the oracle about a query string, the TM is otherwise the same as before.

The idea is to abstract away that part of complexity of determining membership in A from the complexity for the machine. The machine is bounded by its own complexity bound for all the other steps in its computation, including to decide what string to ask the oracle and to write down the query. In particular, if the machine is polynomial time bounded, it can only make queries of polynomially bounded length.

Another way to explain the concept of oracle is to think in terms of algorithms. An algorithm M, with an oracle A, denoted M^A, is like a usual algorithm, except that, it gets a "magic box" that would answer any question M can ask regarding membership in A, in unit time. We think of the oracle as a procedure, and M can make function calls to the procedure. (Strictly speaking though, an oracle A need not even be a computable set.)

We define P^A to be the set of all languages that can be computed in deterministic polynomial time, given oracle access to the language A. NP^A is defined similarly. We can define this notion of oracle to classes other than P and NP, as we shall see later. We can further generalize, to have complexity classes as oracles. For example, if $\mathcal{C}$ is a complexity class, $\mathrm{NP}^{\mathcal{C}}$ is defined to be,

$$\mathrm{NP}^{\mathcal{C}} = \bigcup_{A \in \mathcal{C}} \mathrm{NP}^A$$

We discussed Karp reduction (or p-time many-one reduction) before. A different reduction called Cook reduction (or p-time Turing reduction) can be defined via oracles. We say that a language B *Cook reduces* to a language A in polynomial time, if there is a deterministic polynomial time algorithm that computes B, given A as an oracle. We denote this by $B \leq_T^p A$. In other words, $B \leq_T^p A$ iff $B \in \mathrm{P}^A$.

2.5 A Characterization of PH

THEOREM 2.20 $\mathrm{NP}^{\mathrm{SAT}} = \Sigma_2^p$

Proof. We first show that $\Sigma_2^p \subseteq \mathrm{NP}^{\mathrm{SAT}}$. Let L be a language in Σ_2^p. So there is a polynomial time computable predicate $D(\cdot, \cdot, \cdot)$ such that,

$$x \in L \iff \exists^p y \forall^p z[D(x, y, z) = 1]$$

where the lengths of y and z are bounded by $q(|x|)$ for some polynomial q. We can design a NP oracle machine M that can decide L when given an oracle for SAT. Given x, M first guesses a string y, then asks the question "Is it true that, $\forall^p z[D(x, y, z) = 1]$", where both y and z are polynomially bounded by $q(|x|)$. By Cook's Theorem of SAT being NP-complete, this can be transformed into a SAT query. (Strictly speaking an UNSAT query.) The SAT oracle can answer this question. If the answer from SAT is 'no', i.e., $\forall^p z[D(x, y, z) = 1]$", M accepts x. Such a y will exist iff $x \in L$. So M decides L correctly.

We next show that $\mathrm{NP}^{\mathrm{SAT}} \subseteq \Sigma_2^p$. Let $L \in \mathrm{NP}^{\mathrm{SAT}}$ and M be an NP Turing machine that decides L, given access to a SAT oracle. Each path in M may ask many queries to the oracle. We first construct an equivalent NP machine M' that asks at most one SAT question on each computational path, and it makes the query at the end of the path.

Consider a single path in M. This path may ask multiple questions to the oracle. Consider the first such question "$\varphi \in$ SAT?". At this point M' does not ask the oracle a question. Instead, it guesses either "yes" or "no" as a non-deterministic move. On the branch it guesses "yes", it proceeds to guess a satisfying assignment to the queried formula φ. If the guessed assignment satisfies φ, then M' proceeds with the computation of M with q_{yes}. If the guessed assignment does not satisfy φ, then this non-deterministic rejects. On the branch it guesses "no" it simply takes this "no" as the oracle answer and proceeds with q_{no} in M, while remembering that it guessed "$\varphi \notin$ SAT". Subsequent queries to SAT are handled similarly.

Consider any path in M' that survived till the end and completed simulating a path in M. This path is similar to the path in M, except that it has assumed that certain formulae were unsatisfiable. Note that the fact that it has survived implies that all its assumptions that certain formulae were satisfiable have already been verified. If the path in M being simulated *rejects*, then M' should do so as well. Now suppose the path in M being simulated *accepts*. The path in M' has to decide whether or not to accept. The path (since it has survived) would have verified that the formulae it had assumed to be satisfiable were indeed satisfiable. It now needs to verify whether the formulae it assumed unsatisfiable are indeed unsatisfiable. It does this by asking a *single* SAT query. Suppose the formulas were $\varphi_1, \varphi_2, \ldots, \varphi_k$. M' asks the query $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_k$. Notice that all the formulae are unsatisfiable iff the above formula is unsatisfiable. If the oracle confirms that all formulae guessed to be unsatisfiable are indeed unsatisfiable, then this path accepts.

Now M' has an accepting path iff there is an accepting path in M. Note that an accepting path of M' corresponds to some valid computational path of M, since all the guessed answers to the queries are verified at the end before M' accepts. Now the acceptance of M' can be formulated as a Σ_2^p expression: $\exists$ a path p in M' such that p accepts and for all truth assignments σ, the formula asked at the end of p is not satisfied by σ. ♣

One can extend Theorem 2.20 to other levels of the hierarchy. The above proof gives the gist of the idea, "guess, delay and verify" queries. However, to carry this out formally for all levels of PH, it is more expedient to use some structural formalism. This is a taste of that branch of complexity theory called *Structural Complexity Theory*. There is a certain elegance to the generality of the arguments given; on the negative side, one can be blinded by the formalism and overlook the "real" point of the argument. Here, despite all the formalism, the "real" point of the argument is "guess, delay and verify" queries.

We define an operator $\exists$.

DEFINITION 2.21 *Given any language L, and a polynomial p, define*

$$\exists.L = \{x \mid (\exists^p y)\langle x, y\rangle \in L\},$$

and for any class $\mathcal{C}$*,*

$$\exists.\mathcal{C} = \{\exists.L \mid L \in \mathcal{C}\},$$

where as usual $\exists^p y$ *denotes* $\exists y \in \{0,1\}^{p(|x|)}$*.*

Similarly we can define the operator $\forall$.

DEFINITION 2.22 *Given any language* L*, and a polynomial* p*, define*

$$\forall.L = \{x \mid (\forall^p y)\langle x, y\rangle \in L\},$$

and for any class $\mathcal{C}$*,*

$$\forall.\mathcal{C} = \{\forall.L \mid L \in \mathcal{C}\},$$

where as usual $\forall^p y$ *denotes* $\forall y \in \{0,1\}^{p(|x|)}$*.*

Clearly $\text{co}\forall.\mathcal{C} = \exists.\text{co}\mathcal{C}$. Also $\exists.\forall.\text{P} = \Sigma_2^p$, etc.

We want to prove the following theorem.

THEOREM 2.23 *For any* k*,*

$$\text{NP}^{\text{NP}^{\cdot^{\cdot^{\cdot^{\text{NP}}}}}} = \Sigma_k^p,$$

where the height of the "tower" on LHS is k *(the number of NP's). This is also true relativized to any oracle.*

This is established in a sequence of claims.

Claim 1: for any oracle set A,

$$\text{NP}^A = \exists.\text{P}^A$$

It is trivial that $\exists.\text{P}^A \subseteq \text{NP}^A$. The NP machine simply guesses the string in the definition of $\exists$. and does the P^A computation.

Given a language $L = L(N^A) \in \text{NP}^A$ for some NP machine N, we want to put its acceptance criterion in the form of $\exists.\text{P}^A$. It would be quite trivial if N only asks its queries at the end of a computational path. So, we modify N to N', where on each path N' simulates N and, whenever N makes a query N' simply guesses an answer. At the end of a computational path of N, N' accepts iff N accepts and all the query guesses are correct. ♣

Claim 2: for any oracle set A,

$$\text{NP}^{\exists.\text{P}^A} = \exists.\forall.\text{P}^A$$

One direction is easy, $\exists.\forall.\text{P}^A \subseteq \text{NP}^{\exists.\text{P}^A}$. A simulating oracle NP machine can guess the string in $\exists$., and use its oracle in $\exists.\text{P}^A$ to answer queries in $\forall.\text{P}^A$. Note that oracles in complementary classes have the same power.

Now consider any $L = L(N^B)$, where $B \in \exists.\mathrm{P}^A$. Let $B = \{w \mid (\exists^p y)D(w,y)\}$, where D is in P^A. We wish to express the acceptance of N^B on x in $\exists.\forall.\mathrm{P}^A$. So we will guess ($\exists$) a polynomially long non-deterministic path p, as well as polynomially many answer bits $b_1, \ldots, b_m$, and some y_i (one for each $b_i = 1$ for the i th query "$w_i \in B$?"), such that, ($\forall$) for any z_j (one for each $b_j = 0$ for the j th query "$w_j \in B$?"), the following predicate holds which is computable in P^A— Machine N on x along path p asks some m queries $w_1, \ldots, w_m$ such that for the i th query, if $b_i = 1$ then $D(w_i, y_i)$, if $b_i = 0$ then $\neg D(w_i, y_i)$, and, given the answers being $b_1, \ldots, b_m$, N accepts x. (Note this last use of "some m queries ..." is not a use of quantifiers; the P^A computation simply follows the path p to arrive at its first query w_1, and depending on b_1 and its verification by D, proceed to find its second query w_2, etc.)
♣

Of course since Claim 1 and 2 hold for all oracles A, it also holds when we replace A by any class $\mathcal{C}$.

Claim 3: for any oracle A,

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \subseteq \Sigma_k^{p,A},$$

where the number of NP in the tower on LHS is k.

If $k = 1$ or 2, Claim 3 is direct from Claim 1 and 2. Assume $k > 2$. Claim 1 gives

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \subseteq \mathrm{NP}^{\exists.\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}},$$

where the height of the "tower" of NP on LHS is k and the number of NP on top of P on RHS is $k-2$. Apply Claim 2 to RHS we get

$$\mathrm{NP}^{\exists.\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}} \subseteq \exists.\forall.[\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}] \tag{2.1}$$

where the number of NP on top of P on the RHS is $k-2$. This tower is clearly contained in $\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}$ with a tower of NP of height $k-1$, which, by induction is contained in $\Sigma_{k-1}^{p,A}$. However $\mathrm{P}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}}$ is closed under complement, so that it is also contained in $\Pi_{k-1}^{p,A}$. Now "plug in" in (2.1) we get

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \subseteq \exists.\forall.\Pi_{k-1}^{p,A} = \Sigma_k^{p,A}$$

♣

Now the reverse direction.

Claim 4: for any oracle A,

$$\mathrm{NP}^{\mathrm{NP}^{\cdot^{\cdot^{\cdot^{\mathrm{NP}^A}}}}} \supseteq \Sigma_k^{p,A},$$

where the height of the "tower" of NP on LHS is k.

This is simple: RHS is $\exists.\Pi_{k-1}^{p,A}$. By induction $\Pi_{k-1}^{p,A} \subseteq \text{coNP}^{\cdot^{\cdot^{\cdot^{\text{NP}^A}}}}$ where the tower has height $k-1$. But as oracles, complementary classes have the same power, so that $\text{NP}^{\mathcal{C}}$ can handle $\exists.\Pi_{k-1}^{p,A}$, where $\mathcal{C} = \text{NP}^{\cdot^{\cdot^{\cdot^{\text{NP}^A}}}}$ where the tower in $\mathcal{C}$ has height $k-1$. ♣

Claim 3 and 4 proves Theorem 2.23.

2.6 Complete Problems for Σ_k^p

We have seen canonical complete languages for the classes NP and PSPACE (see A_{NP} and A_{PSPACE} defined in Section 2.2). Similarly, one can construct canonical complete languages for each level of the polynomial hierarchy Σ_k^p. In this section, we exhibit "natural" complete languages for each Σ_k^p.

As usual, we say that a language L is complete (under polynomial time reductions) for Σ_k^p, if $L \in \Sigma_k^p$ and every language in Σ_k^p Karp reduces to L. The familiar Cook's theorem already gives us a complete language for $\Sigma_1^p = \text{NP}$, namely the language SAT. We can get complete languages for higher levels of the hierarchy by generalizing SAT.

DEFINITION 2.24 *A quantified boolean formula with k alternations over a set of n boolean variables $X = \{x_1, x_2, \ldots, x_n\}$ is an expression of the form*

$$F = \exists X_1 \forall X_2 \exists X_3 \ldots Q X_k [f(x_1, x_2, \ldots x_n)],$$

where f is a (quantifier free) boolean formula over the n variables and $X_1, X_2, \ldots X_k \subseteq X$ is a partition of X. Here, Q is the existential quantifier if k is odd, and it is the universal quantifier if k is even.

We say that F is *valid* if the above expression is logically true. Meaning, there exists some choice of values for the variables in X_1 such that, for all choices of values for X_2, there exists some choice of values for X_3 such that, so on ..., the formula f evaluates to true.

Let QBF_k be the set of all valid quantified boolean formulas with k alternations. It is quite easy to adapt the proof of Cook's theorem to show that for any $k \geq 1$, QBF_k is Σ_k^p-complete.

THEOREM 2.25 *For any $k \geq 1$, QBF_k is Σ_k^p-Complete.*

2.7 Alternating Turing Machines

Another way of looking at the Polynomial Time Hierarchy is to use Alternating Turing Machines. An alternating TM is a nondeterministic Turing machine where each state is

labeled as existential or universal. It functions the same way as a non-deterministic Turing machine does, but the notion of acceptance is defined differently.

Recall that the *configuration* of an ordinary Turing machine is a tuple consisting of the current state of the Turing machine, the symbols on the tape, and the position of the head. As the Turing machine processes the input, it moves from one configuration to another. In an Alternating Turing machine, we define an *existential configuration* as one in which the current state of the Turing machine is an existential state, and a *universal configuration* as one in which the state is a universal state.

Acceptance is defined recursively: A final configuration is an accepting or rejecting configuration iff the state is accepting or rejecting. For non-final configurations, the acceptance is defined as: An existential configuration of the Turing machine is an accepting configuration if there is an accepting configuration among its next configurations in one step. A universal configuration is an accepting configuration if all of its next configurations are accepting configurations. Finally x is accepted iff the beginning configuration is an accepting configuration.

The relation between alternating Turing machines and quantifier alternations is as follows. The proof is left as an exercise.

THEOREM 2.26 *Σ_k^p is precisely the set of languages L accepted by polynomial time alternating Turing machines with k alternations with the starting configuration as an existential configuration. Similarly, Π_k^p is the set of languages L accepted by polynomial time alternating Turing machines with k alternations with a universal starting configuration.*

Exercise: Prove Theorem 2.26