19

Complexity Theory and NP-Completeness

The theory of NP-completeness is about a class of problems that have defied efficient (polynomial-time) algorithms, despite decades of intense research. Any problem for which the *most efficient known algorithm* requires exponential time[1] is called *intractable*. Whether NP-complete problems will *remain* intractable forever, or whether one day someone will solve *one* of the NP-complete problems using a polynomial-time algorithm remains one of the most important open problems in computing. The Clay Mathematics Institute has identified seven Millennium Problems, each carrying a reward of $1 million (US) for the first person or group who solves it; the '$\mathsf{P} = \mathsf{NP}$' problem is on this list. Stephen Cook provides an official description of this problem, and associated (extremely well-written) set of notes, also at the Clay web site [23].

The theory of *NP-completeness* offers a way to "bridge" these problems through efficient simulations (polynomial-time mapping reductions $\leq_P$ (Definition 16.4) going both ways between any two of these problems) such that *if* an efficient algorithm is found even for *one* of these problems, then an efficient algorithm is immediately obtained for *all* the problems in this class (recall our discussions in Chapter 1, page 11). Section 19.1 presents background material. Section 19.3 presents several theorems and their proofs. Section 19.4 provides still more illustrations that help avoid possible pitfalls. Section 19.5 discusses notions such as CoNP. Section 19.5 concludes.

[1] The best known algorithm for an intractable problem has complexity $\mathcal{O}(k^n)$ for an input of length n, and $k > 1$.

19.1 Examples and Overview

19.1.1 The traveling salesperson problem

The *traveling salesperson problem* is a famous example of an NPC problem. Suppose a map of several cities as well as the cost of a direct journey between any pair of cities is given.[2] Suppose a salesperson is required to start a tour from a certain city c, visit the other $n-1$ cities in some order, but visiting each city *exactly once*, and return to c while minimizing the overall travel costs. What would be the most efficient algorithm to calculate an optimal tour (optimal sequence of cities starting at c and listing every other city exactly once)?

- This problem is intractable.
- This problem has also been shown to be NPC.

The NPC class includes thousands of problems of fundamental importance in day-to-day life - such as the efficient scheduling of airplanes, most compact layout of a set of circuit blocks on a VLSI chip, etc. They are all intractable.

19.1.2 P-time deciders, robustness, and 2 vs. 3

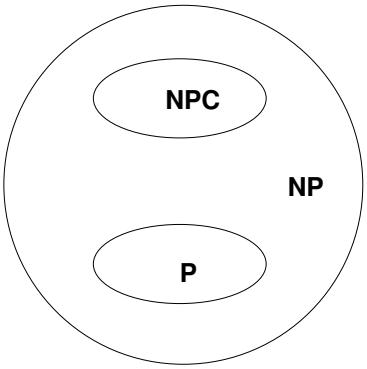

Fig. 19.1. Venn diagram of the language families P, NP, and NPC; these set inclusions are *proper* if $P \neq NP$ — which is an open question

[2] Assume that these costs, viewed as a binary relation, constitute a symmetric relation.

The space of problems we are studying is illustrated in Figure 19.1. *It is not know* whether the inclusions in Figure 19.1 are proper.[3] The oval drawn as P stands for problems for which polynomial time deciders exist.

Let

$$\text{TIME}(t(n)) = \{\ L\ \mid\ L \text{ is a language decided by an } \mathcal{O}(t(n)) \text{ time TM}\ \}$$

Then,

$$\text{P} = \cup_{k\geq 0}\ \text{TIME}(n^k).$$

As an example, the following language $L_{0^n1^n}$ is in P

$$L_{0^n1^n} = \{0^n1^n\ \mid\ n \geq 0\}$$

because

- it is in TIME(n^2), as per the following algorithm, A1:
  - Consider a DTM that, given $x \in \Sigma^*$, zigzags on the input tape, crossing off one 0 and then a (supposedly matching) 1. If the tape is left with no excess 0s over 1s or vice versa, the DTM accepts; else it rejects.
- It is even in $\mathcal{O}(n\ logn)$ as per the following algorithm, A2:
  - In each sweep, the DTM scores off every other 0 and then every other 1. This means that in each sweep, the number of surviving 0s and 1s is half of what was there previously. Therefore, $log(n)$ sweeps are made, each sweep spanning n tape cells. The stopping criterion is the same as with algorithm A1.

Another member of P is *TwoColor* (see Exercise 19.1); here two-colorable means one can color the nodes using two colors, with no two adjacent nodes having the same color:

$$TwoColor = \{\langle G\rangle\ \mid\ G \text{ is an undirected graph that is two\_colorable}\ \}$$

We shall define NP and NPC later in this chapter.

19.1.3 A note on complexity measurement

We will not bother to distinguish between N and $N\ log(N)$, lumping them both into the polynomial class. The same is true of $N^k\ log^m(N)$ for all k and m. Our complexity classification only has two levels: "polynomial" or "exponential." The latter will include the *factorial* function, and any such function that is harder than exponential (e.g., Ackermann's function).

[3] When the $1M Clay Prize winner is found, they would either assert this diagram to be exact, or simply draw one big circle, writing "P" within it – with a footnote saying "NP and NPC have been dispensed with."

19.1.4 The robustness of the Turing machine model

There are a mind-boggling variety of *deterministic* Turing machines: those that have a doubly-infinite tape, those that have multiple tapes, those that employ larger (but finite) alphabets, and even conventional deterministic random-access machines such as desktop and laptop computers (given an unlimited amount of memory, of course). All this variety does not skew our two-scale complexity measurement:

- P is invariant for all models that are polynomially equivalent to the deterministic single-tape Turing machine (which includes all these unusual Turing machine varieties)
- P roughly corresponds to the class of problems that are realistically solvable on modern-day random-access computers.

Hence, studying complexity theory based on deterministic single-tape Turing machines allows us to predict the complexity of solving problems on real computers.

19.1.5 Going from "2 to 3" changes complexity

It is a curious fact that in many problems, going from "2 to 3" changes the complexity from polynomial to seemingly exponential. For instance, K-colorability is the notion of coloring the nodes of a graph with K colors such that no two adjacent nodes have the same color. Two-colorability is in P, while three-colorability is NPC. This is similar to the fact that the satisfiability of 2-CNF formulas is polynomial (Section 18.3.8), while that of 3-CNF formulas is NPC (Theorem 19.8). The reasons are, not surprisingly, somewhat similar.

19.2 Formal Definitions

We now proceed to define the remaining ovals in Figure 19.1.

19.2.1 NP viewed in terms of verifiers

We now present the *verifier* view of NP, with the *decider* view presented in Definition 19.5.

Definition 19.1. *(Verifier view of* NP*)* A language L is in NP if there exists a deterministic polynomial-time Turing machine V_L called the *verifier*, such that given any $w \in \Sigma^*$, there exists a string $c \in \Sigma^*$ such that $x \in L$ exactly when $V_L(w, c)$ accepts.

Here, c is called a *certificate*. It also corresponds to a "guess," as introduced in Section 15.5.1 of Chapter 15 (some other equivalent terms are *witness*, *certificate*, *evidence*, and *solution*). According to Definition 19.1, the language

$$Clique = \{\langle G, k\rangle \mid G \text{ is an undirected graph having a } k\text{-clique}\}$$

is in NP because there exists a verifier V_{Clique} such that

$$Clique = \{\langle G, k\rangle \mid \text{There exists } c \text{ such that } V_{Clique} \text{ accepts } (G, k, c)\}.$$

Here, c is a sequence of k nodes. V_{Clique} is a polynomial-time algorithm that is captured by (as well as carried out by) a *deterministic* Turing machine. This DTM does the following: (i) checks that G is an undirected graph, and c is a list of k nodes in G, and (ii) verifies that the nodes in c indeed form a clique. Note that the ability to *verify* a guess in polynomial-time means that the length of c must be bounded by a polynomial in terms of input size.

Given G and k, all known practical algorithms take exponential time to *find out* which k nodes form a clique. On the other hand, given an arbitrary list of k nodes, *verifying* whether these form a clique is easy (takes a linear amount of time). Problems in NP share this property by definition. Recall our discussions in Section 15.5.1 about Mersenne primes that also shares this property of easy verifiability.[4]

19.2.2 Some problems are outside NP

It is indeed remarkable that there are problems where even *verifying a solution* is hard, taking an exponential amount of time with respect to all known algorithms for these problems! Clearly these problems *do not* belong to NP. For example, for the $\overline{Clique}$ problem defined in Chapter 17, efficient verifiers have, so far, remained impossible to determine. Intuitively, this seems to be because languages (problems) such as $\overline{Clique}$ seem to call for an enumeration of *all candidate list of k nodes* and an assertion that each such list, in turn, does *not* form a clique.[5] It seems that the certificates for these problems must be

[4] Students believe that every problem assigned to them is NP-complete in difficulty level, as they have to *find* the solutions. Teaching Assistants, on the other hand, find that their job is only as hard as P, as they only have to *verify* the student solutions. When some students confound the TAs, even verification becomes hard - something discussed in Section 19.2.2.

[5] Continuing with the analogy introduced in Chapter 17, we are being asked to prove that no group of k people know each other, as opposed to proving that *some* k people know each other.

exponentially long, because they are a concatenation of an exponential number of these candidate list of nodes mentioned above. The mere act of reading such certificates consumes an exponential amount of time! Of course, these are simply conjectures: it is simply not know at present how to prove that problems such as $\overline{Clique}$ cannot have succinct (polynomially long) certificates.

To sum up, problems whose solutions are easy to verify (NP) are, in some sense, *easier* than problems whose solutions are *not* easy to verify — even though finding the solution is hard in both cases. As Pratt puts it so eloquently in his paper where he proves that *primes* are in NP [98],

> "The cost of *testing* primes and composites is very high. In contrast, the cost of *selling* composites (persuading a potential customer that you have one) is very low—in every case, one multiplication suffices. The only catch is that the salesman may need to work overtime to prepare his short sales pitch.

19.2.3 NP-complete and NP-hard

Definition 19.2. *(NP-complete)* If a language L is shown to be in NP, *and furthermore*, if it can be shown that for every language $X \in \mathsf{NP}$, $X \leq_P L$, then L is NPC.

Therefore, showing a problem to be in NP is a prerequisite to showing that it is NPC.

Definition 19.3. *(NP-hard)* If for all $X \in \mathsf{NP}$ we have $X \leq_P L$, then L is said to be *NP-hard*.

From all this, "NPC" means "NPH" *and* "belongs to NP."

Note: If L is NP-hard, it means that L is *at least as hard as NP*. It is possible that L is so hard as to be *undecidable*, as is shown in Section 19.4.

19.2.4 NP viewed in terms of deciders

In Definition 19.1, NP was defined with the help of deterministic Turing machines V_L which are verifiers. There is an alternative definition of NP in terms of nondeterministic Turing machines, which is now presented, after presenting the notions of a *nondeterministic decider* and a *nondeterministic polynomial-time decider*.

Definition 19.4. *(NP decider)* An NDTM N_L with starting state q_0 is a nondeterministic decider for a language L exactly when for all $x \in \Sigma^*$, $x \in L$ if and only if N_L has an accepting computation history starting at q_0x. It is an NP (nondeterministic polynomial-time) decider for L if, for *all strings* $x \in L$, the length of the *shortest* accepting computation history starting at q_0x is $\mathcal{O}(n^k)$.

Note that we are not requiring this NDTM to terminate along all paths. Using the notion of '*shortest*,' we are able to ignore all other paths.

Definition 19.5. *(Decider view of* NP*)* NP is the class of decidable languages such that associated with each $L \in$ NP is a nondeterministic polynomial-time decider.

Definition 19.4 is adopted in [44], [39], and [67]. In [111], different definitions (that consider the *longest* computation) are employed. The advantages of definitions that go by the shortest accepting computation history are the following:

- It allows NDTMs to be designed without special precautions that are irrelevant in the end. In particular, we do not need to define the NDTMs to avoid paths that are unbounded in length (see Section 19.2.5 for an example).
- It helps focus one's attention on positive outcomes (the $x \in L$ case), as well as the "guess and check" principle of Section 15.5.1, and, last but not least,
- It helps present and prove Theorem 19.6 very clearly.

Also, please note the following:

- A nondeterministic polynomial-time decider Turing machine N_L has *nondeterministic polynomial* runtime. 'Nondeterministic polynomial' is a different way of measuring runtime, different from how it is done with respect to DTMs, where run times are measured in terms of the number of steps taken by a DTM from start to finish over all inputs, where each input induces exactly one computation history. In case of NP, for each input, there could be *multiple* computation histories, and we simply focus on the shortest ones.

19.2.5 An example of an NP decider

These notions are best explained using an example; we choose Figure 15.3 for this purpose. Given any string of the form ww, this machine generates one guess (refer to Section 15.5.1) that correctly identifies and checks around the midpoint, ultimately leading to acceptance.

However, this machine also has many useless guesses that lead to rejection. In fact, *there is no a priori bound on the number of times this NDTM loops back to state* q2 *before exiting the loop!* It could, therefore, be guessing any point that is arbitrarily away from the left-end of the tape to be the midpoint! Of course we could easily have defined a "better" NDTM that rejects as soon as we are off the far right-end of the input. However, such "optimizations" are not helpful in any way; keep in mind that

- NDTMs are mathematical devices that only serve one purpose: to measure complexity in a manner that implements the "guess and check" idea discussed in Section 15.5.1 (and further discussed in this chapter).
- In its role as a theoretical device, it is perfectly normal for an NDTM to have a computation tree that has *an infinite number of branches out of its nondeterministic selection state(s).* However, since we measure the time complexity in terms of the shortest accepting computation history, these longer paths automatically end up getting ignored.

Theorem 19.6. *The verifier view of* NP *(Definition 19.1) and the decider view of* NP *(Definition 19.4) are equivalent.*

Proof outline:

With respect to the first part of the proof, we observe that an NDTM can always be designed to have a loop similar to the self-loop at state q2 of Figure 15.3. In this loop, it can write out *any* string from the tape alphabet and then call it "the certificate c" and then verify whether it is, indeed, a certificate. Now, if there *exists* any certificate at all, then one would be found in this manner. Furthermore, if there exists a polynomially bounded certificate, then again, it would be found since we heed only the shortest accepting computation history. For the second part of the proof, we let the certificate be tree paths, as described in Section 15.6.2.

Proof: Given V_L, we can build the NDTM N_L as follows:

```
N_L =
 Accept input w;
  Employ a nondeterministic loop, and write out
    a certificate string c on the tape; c is
    an arbitrary nondeterministically chosen finite string;
    c is written after the end of w;
  Run V_L on (w,c), and accept if V_L accepts.
```

Going by our definition of the *shortest accepting computation history*, there will be one certificate that works, since V_L has a certificate that works. Therefore, N_L will consist of the certificate generation phase followed by feeding V_L with w and c. The N_L thus constructed will have a nondeterministic polynomial runtime and decides L. □

• Given N_L, we can build the NDTM V_L as follows.

```
V_L =
 On input w, c,
  Use c to guide the selection among the
    nondeterministic transitions in N_L;
  Accept when N_L accepts.
```

In essence, c would be a sequence of natural numbers specifying which of the nondeterministic selections to make ("take the first turn; then the third turn; then the second turn; ...[6]). Now, if N_L has an accepting computation history, there is such a certificate c that leads to acceptance. Hence, the DTM V_L would be a deterministic polynomial-time verifier for w, c. □

19.2.6 Minimal input encodings

In measuring complexity, one must have a convention with regard to n, the length of the input. The conventions most widely used for this purpose are now explained through an example.. Consider the *Clique* problem again. To measure how much time it takes to answer this membership question, one must encode $\langle G, k \rangle$ "reasonably"—in a minimality sense. In particular, we should avoid encoding $\langle G, k \rangle$ in a unary fashion. Doing so can skew the complexity. Details are in Section 19.5.

19.3 NPC Theorems and proofs

Definition 19.2 defined $L \in \mathsf{NPC}$ in terms of a reduction from *all* $X \in \mathsf{NP}$. This may seem to be an infeasible recipe to follow, as there are $\aleph_0$ languages that are in NP. Historically, only *one* problem was solved using this tedious approach (detailed in Section 19.3.1). All other problems shown to be NPC were proved using Definition 19.7, which offers

[6] Imagine a boat in a lake being turned and pushed around by the hands of a giant, and its rudder limply rotating, following the motions of the boat. The motions of the rudder are analogous to c.

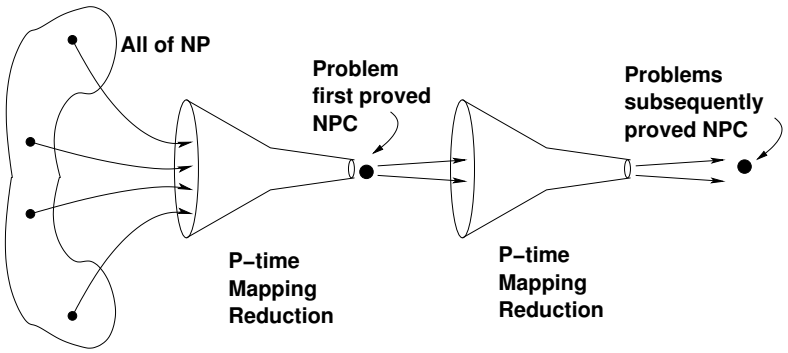

Fig. 19.2. Diagram illustrating how NPC proofs are accomplished. Definition 19.2 is illustrated by the reduction shown on the left while Definition 19.7 is illustrated on the right.

a much more practical recipe, assuming there exists at least one NPC language.[7]

Definition 19.7. *(NP-complete)* A language L is NPC if $L \in \mathsf{NP}$ and furthermore, for some other language $L^{'} \in \mathsf{NPC}$, we have $L^{'} \leq_P L$.

This definition is equivalent to Definition 19.2 because if $L^{'} \in \mathsf{NPC}$, we have $\forall X \in \mathsf{NPC} : X \leq_P L^{'}$, and $\leq_P$ is transitive. The "funnel diagram" in Figure 19.2 illustrates this approach.

In order to identify the very first NPC problem, we do need to go by Definition 19.2. The first problem that had this 'honor' was 3-CNF satisfiability ("3-SAT"), as Cook and Levin's celebrated proof shows. Recall from Section 18.3.7 general discussions about 3-CNF.

19.3.1 NP-Completeness of 3-SAT

Theorem 19.8. 3-CNF satisfiability is NP-complete.

3-SAT is in NP

We go by Definition 19.1. Consider a satisfying assignment σ for a given 3-CNF formula φ. Clearly, σ is of polynomial length, and verifying that it satisfies φ takes polynomial-time through a verification algorithm that substitutes into φ as per σ and simplifies the formula to $true$ or $false$.

[7] Like the proverbial 'chicken and the egg,' we assume the first egg, - er, first chicken

For any $L \in \mathsf{NP}$, $L \leq_P$ 3-SAT

We sketch the proof emphasizing the overall structure of the proof as well as some crucial details. For example, (i) we show how the *existence* of a deterministic polynomial time algorithm for 3-SAT implies the *existence* of such an algorithm for *any* problem in NP, and (ii) we show how, given a specific NP problem such as *Clique* and given a deterministic polynomial algorithm for 3-SAT, we can obtain a deterministic polynomial algorithm for *Clique*.

Consider some $L \in \mathsf{NP}$. Then there exists an NDTM decider N_L for L. What we have to show is that there *exists* a polynomial-time mapping reduction f from L to 3-CNF such that given N_L and an arbitrary $w \in \Sigma^*$, there *exists* a 3-CNF formula $\varphi_{L,w}$ that can be obtained from N_L and w using f, such that $\varphi_{L,w}$ is satisfiable if and only if $w \in L$.

> *Punchline:* Therefore, if one were to find a polynomial-time algorithm for 3-CNF satisfiability, there would now be a polynomial-time algorithm for *every* L in NP.

To prove the existence of the mapping reduction alluded to above, refer to Figure 19.3. Consider the computation of N_L on some $w \in \Sigma^*$ starting from the instantaneous description ID0 $= q_0 w$. If $w \in L$, there is an accepting computation history that starts with $q_0 w$ and is of polynomial length (we do not know this length exactly; all we know is that it is a polynomial with respect to $|w| = n$). Let this polynomial be

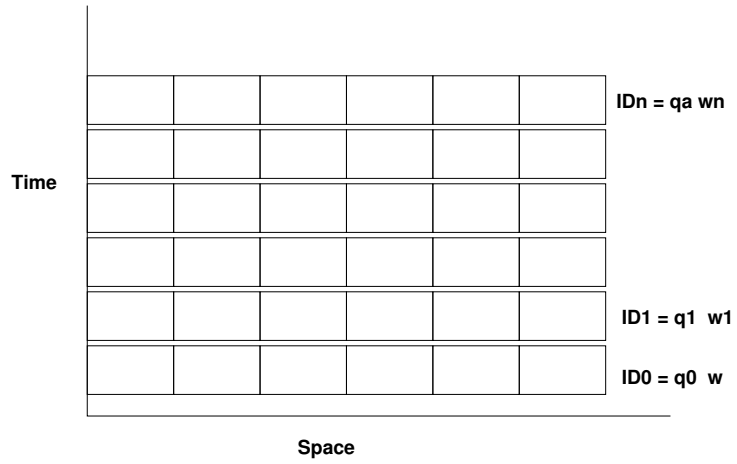

Fig. 19.3. Proof of the Cook-Levin Theorem

$p(n)$. If $w \notin L$, then no accepting computation history (of any length!) exists.

Imagine a computation history (sequence of IDs) of length $p(n)$. In it, (i) only a polynomial amount of new tape cells get written into (the "space" axis) and (ii) the accepting computation history is of polynomial length (the "time" axis; see Figure 19.3 for an illustration). This figure illustrates the computation starting from q_0w as a sequence of IDs starting with ID0 at the bottom. Now, observe that if $w \in L$, there must exist a $p(n) \times p(n)$ matrix as shown in Figure 19.3 that (i) starts with q_0w at the bottom, (ii) ends with an accepting ID at the top, and (iii) has any two adjacent rows related by a Turing machine transition rule. Now, it is clear that there are basically two rules, as mentioned in Section 17.3.6: (i) one rule corresponds to a right-hand move ($\delta(q, a) = (r, b, R)$), and (ii) another corresponding to a left-hand move ($\delta(q, a) = (r, b, L)$ for every $c \in \Gamma$). Furthermore, the effect of these moves on adjacent IDs can be captured through "windows" that change across two IDs. For instance, for ID_{i+1} obtained from ID_i through a right move, a 2×2 window comes into play, and for a left move, a 3×2 window comes into play, as mentioned in Section 17.3.6. It is also clear that the contents of each cell in this $p(n) \times p(n)$ matrix can be represented using a finite number of cell level Boolean variables.

Given all this, the crux of our proof is that given this $p(n) \times p(n)$ matrix, we can build our 3-CNF formula $\varphi_{L,w}$ involving the cell level Boolean variables such that this formula is true exactly when the matrix represents an accepting computation history (we also refer to this 3-CNF formula as a "circuit," connoting a digital combinational circuit that can serve as a decoder for an accepting computation history matrix). The actual construction of this formula is tedious and skipped here (but may be found in scores of other web references or books). On page 357, we illustrate what this formula achieves with respect to the *Clique* example.

What we have sketched thus far is the *existence* of a mapping reduction that yields $\varphi_{L,w}$ such that this formula is satisfiable exactly when $w \in L$. If $w \notin L$, there is no matrix that will represent an accepting computation history, and hence $\varphi_{L,w}$ will *not* be satisfiable. □

A detail about not knowing $p(n)$

A point that may vex the reader is this: how do we know how big a circuit (3-CNF formula) to build, given that the circuit has to sit on top of the $p(n) \times p(n)$ matrix, hoping to decode its contents, when we don't even know $p(n)$ concretely? Fortunately, this step is not necessary—all

we care for is the *existence* of a family of circuits, one circuit for each value of $p(n)$. There would, in this family, exist a circuit that "works" for every language L in NP, and correspondingly there would be a family of f functions that produced this family of circuits. Therefore, for each possible value of $p(n)$, there *exists* an f function that produces a matrix of size $p(n) \times p(n)$ and a 3-CNF formula that acts on this matrix; and hence, the desired mapping reduction $L \leq_P$ 3-CNF *exists* for each $L \in$ NP. The *existence* of this mapping reduction for every $L \in$ NP allows us to claim that 3-SAT is in NPC.

What if 3-SAT is in P?

It is good to be sure that formal proofs mean something concrete; to this end, we subject our discussions above to an acid test. Suppose 3-SAT is in P (which is an open question; but we entertain this thought to see what happens) and let the decider be $DP_{3-\text{CNF}}$. Because of what NP-completeness means, we should now be in a position to argue that there exists a deterministic polynomial-time algorithm for any $L \in$ NP. How do we achieve that?

Fortunately, this result is immediate. Since the polynomial $p(n)$ exists, a mapping reduction to yield $\varphi_{L,w}$ exists. We can obtain this formula using the mapping reduction, feed it to $DP_{3-\text{CNF}}$, and return the accept/reject decision of $DP_{3-\text{CNF}}$. Therefore, a deterministic polynomial-time algorithm for an arbitrary $L \in$ NP exists, which would then mean $P = NP$.

Illustration on *Clique*

We would like to take our acid test even further: suppose 3-SAT is in P; let us find a polynomial algorithm for $Clique$. Since $Clique \in$ NP, it has a nondeterministic polynomial time decider, say N_{Clique}. We can design a specific NDTM decider for $Clique$. One of the most straightforward designs for N_{Clique} would be to have an NDTM nondeterministically write out k nodes on the tape and check whether these nodes indeed form a k-clique. Now, there are *many* (exponentially many!) choices of k nodes to write out on the tape. However, after writing out one of those guesses on the tape, N_{Clique} would engage in a polynomially bounded checking phase. The 3-CNF formula $\varphi_{L,w}$ that would be synthesized for this example will have the following properties:

- It will be *falsified* if the first ID (bottom row of the matrix) is not the starting ID which, in our example, would be $q_0\langle G, k\rangle$.

- It will be falsified if any two adjacent rows of the matrix are not bridged by a legitimate transition rule of the NDTM.
- It will be falsified if the final row (topmost) is not an accepting ID.
- The formula will straddle a matrix of size $p(n) \times p(n)$, where the value of $p(n)$ will be determined by the nature of the algorithm used by N_{Clique} for *Clique*, and the value of k. In particular, $p(n)$ will equal the number of steps taken to write out some sequence of k nodes nondeterministically, followed by the number of steps taken to check whether these nodes form a clique.

In short, an accepting computation history will deposit a bit-pattern inside this matrix to make the Boolean formula emerge true. Said another way, the *satisfiability* of the formula will indicate the *existence* of an accepting computation history (the existence of a selection of k nodes that form a clique). Therefore, if $DP_{3-\text{CNF}}$ exists, *Clique* can be solved in polynomial-time.

> To reflect a bit, the existence of $DP_{3-\text{CNF}}$ is a *tall order* because it gives us a mechanism to encode exponential searches as polynomially compact formulas (such as $\varphi_{L,w}$ did) and conduct this exponential search in polynomial-time! This is one reason why researchers strongly believe that deciders such as $DP_{3-\text{CNF}}$ do not exist.

19.3.2 Practical approaches to show NPC

The most common approach to show a language L to be NPC is to use Definition 19.7 and reduce 3-SAT to L, and then to show that $L \in \mathsf{NP}$. This has led many to observe that 'NP-complete problems are 3-SAT in disguise.' Many other source languages for reduction (besides 3-SAT) are, of course, possible.

Illustration 19.3.1 Let$Hampath =$

$\{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$.

It can be shown that $Hampath \in \mathsf{NP}$, and further 3-SAT$\leq_P Hampath$. This establishes that $Hampath$ is NPC.

Illustration 19.3.2 Let us prove that *Clique* is NPC. First of all, $Clique \in \mathsf{NP}$ as captured by the verifier V_{Clique} on page 349. To show that *Clique* is NPH, we propose the mapping reduction from 3-SAT into

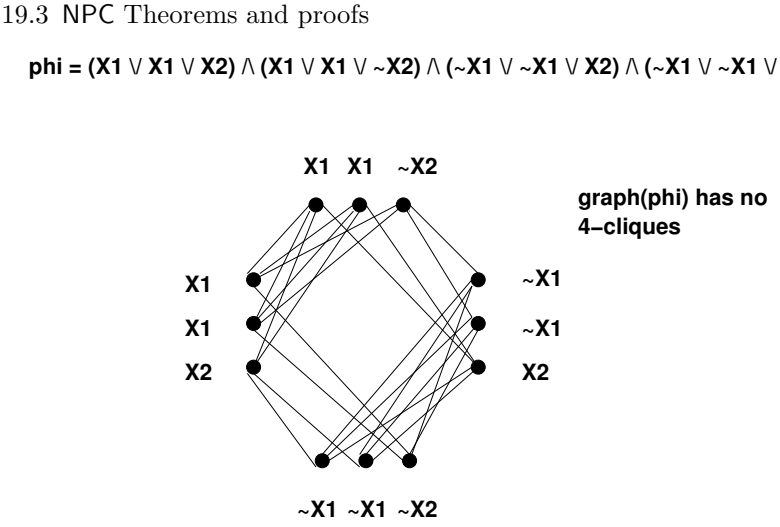

Fig. 19.4. The Proof that *Clique* is NPH using an example formula $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (x_1 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_2)$

Clique as captured in Figure 19.4. Basically, for every clause, we introduce an "island" of nodes with each node in the island labeled with the same literal as in the clause. There are no edges among the nodes of an island. Between two islands, we introduce an edge between every pair of literals that can be simultaneously satisfied (are not complementary literals).

Suppose $\varphi \in$ 3-SAT. This means that there is an assignment that satisfies every clause. Let l_i be the literal that is set to true by the assignment in clause c_i, for every $i \in C$, where C is the number of clauses (for uniqueness, we may select the literal with the lowest index that is set to true in each clause). In this case, by construction, $graph(\varphi)$ will have a clique connecting the nodes $l_0, l_1, \ldots, l_{C-1}, l_0$.

Suppose $\varphi \notin$ 3-SAT. Now suppose we assume that $graph(\varphi)$ has a k-clique. The existence of a k-clique means that by following the edges of the clique, it should be possible to pick one literal per clause such that all these literals (and hence these clauses) can be simultaneously satisfied. However, from Theorem 18.1, we know that given any unsatisfiable CNF formula φ, one can pick an arbitrary assignment σ, and be assured that $\sigma(\varphi)$ (the formula under the assignment) has one clause all of whose literals are true, and another clause all of whose literals are false. The clause that has all its literals false will prevent there being a k-clique. To confirm all this, in Figure 19.4 we observe that there are no 4-cliques.

19.4 NP-Hard Problems can be Undecidable (Pitfall)

What happens if someone shows L to be NPH but neglects to show $L \in \mathsf{NP}$, and yet claims that $L \in \mathsf{NPC}$? To show the consequences of this mistake rather dramatically, we will show that the language of Diophantine equations, *Diophantine*, is NP-hard (NPH). Briefly, *Diophantine* is the set of Diophantine equations that have integer roots. An example of such an equation is $6x^3z^2 + 3xy^2 - x^3 - 10 = 0$. This language was shown to be undecidable by Yuri Matijasević in a very celebrated theorem. Hence, if someone forgets to show that a language L is in NP, and yet claims that L is NPC, he/she may be claiming that something undecidable is decidable! (Recall that all NPC problems are decidable.) In short, NP-completeness proofs *cannot* be deemed to be correct unless the language in question is shown to belong to NP.

19.4.1 Proof that Diophantine Equations are NPH

We follow the proof in [27] to show that the language *Diophantine* below is NPH:

$Diophantine = \{p \mid p \text{ is a polynomial with an integral root}\}$

The mapping reduction 3-SAT $\leq_P$ *Diophantine* is achieved as follows. Consider a 3-CNF formula φ:

- Each literal of φ, x, maps to integer variable x.
- Each literal $\overline{x}$ maps to expression $(1 - x)$.
- Each $\vee$ in a clause maps to . (times).
- Each clause is mapped as above, and then *squared*.
- Each $\wedge$ maps to $+$.
- The resulting equation is set to 0.
- Example: map

$$\varphi = \ (x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee \overline{y})$$

to

$$E = \ (x.y)^2 + (x.(1-y))^2 + ((1-x).(1-y))^2 = 0.$$

- To argue that this is a mapping reduction, we must show that φ is satisfied iff E has integral roots. Here is that proof:
  - For the forward direction, for any assignment of a variable v to $true$, assign v in the integer domain to the integer 0; if v is $false$, use integer 1. In our example, $x = true, y = false$ satisfies φ, and so choose $x = 0, y = 1$ in the integer domain. This ensures that $(x.y)^2$ is zero. Proceeding this way, every satisfying assignment has the property of leaving the entire summation of expressions 0, thus satisfying the integer equation.

– For the reverse direction, note that $E = 0$ means that each product term in the integer domain is 0 (since squares can't be negative). For example, if xy is a product term in the summation of E, we may have $x = 45$ and $y = 0$. The Boolean assignment for this case is found as follows: for every integer variable x that is zero, assign the corresponding Boolean variable x to $true$; for integer variable x that is non-zero, assign the Boolean variable x to $false$. For example, if we have $x = 0$ in a product term $x.y$, we assign Boolean x to $true$. This ensures that $(x \vee y)$ is true. Also, in $x.(1-y)$ if $x = 45$ and $y = 1$, we assign y to $false$ and x to false. This ensures that $(x \vee \neg y)$ is true. We can easily check that this construction ensures that $E = 0$ exactly when the corresponding φ has a satisfying assignment.

19.4.2 "Certificates" of Diophantine Equations

In order to visualize the transition from being NPC, to being outside NP but still decidable, and finally to being undecidable, let us discuss Diophantine equations in the context of certificates. Consider the language $Hampath$; clearly, every member of this language has a polynomial certificate. This certificate is a simple path (path without repeating nodes) connecting s and t that visits every other node. Languages such as $\overline{Hampath}$ do not have, as best as is known, polynomial certificates. However, exponentially long certificates do exist; these certificates list every simple path connecting s and t, thus providing cumulative evidence that there *does not* exist a Hamiltonian path.

It is evident that *Diophantine* is recursively enumerable (it is TR) but not recursive (it is not decidable). One may attempt to build a nondeterministic machine M_{Dio}, of as yet unclear status, to process membership of a given Diophantine equation in *Diophantine*: M_{Dio} guesses a certificate in the guess-generation phase consisting of guessed values for the variables of the equation. M_{Dio} then plugs in these values and checks whether the given equation is satisfied (equals 0). For an undecidable languages such as *Diophantine*, certificates exist, but are unbounded for the case where the equation has a solution. When the equation has no solution, the certificates are infinite (one has to list every possible value for the variables and show that they do not solve the equation).

In summary, polynomial, exponential, and unbounded certificates correspond to three classes of hardness.

19.4.3 What other complexity measures exist?

There are many other complexity metrics such as space complexity and circuit (parallel) complexity. It must be relatively obvious what the term 'space complexity' means: how much *space* (memory) does an algorithm require? In this context, please note that *space is reusable while time is not.* This means that the term space complexity refers to the *peak* space requirement for an algorithm. There is also the fundamentally important result, known as *Savich's Theorem*, that says that nondeterministic Turing machines can be simulated on top of deterministic Turing machines with only a polynomial added cost. This is in contrast with time complexity where we do not know whether nondeterministic Turing machines can be simulated on a deterministic Turing machine with only a polynomial added cost.

The term 'circuit complexity' may be far from obvious to many. What it pertains to is, roughly, *how easy a problem is to parallelize.* In circuit complexity, the intended computation is modeled as a Boolean function, and the *depth* of a combinational circuit that computes this function is measured. Problems such as *depth-first search* are, for instance, not easy to parallelize (log-depth circuits cannot be found), whereas *breadth-first search* easy to parallelize under this complexity measure. Log-depth circuits, in a sense, help assess how easy it is to divide and conquer a problem.

19.5 NP, CoNP, etc.

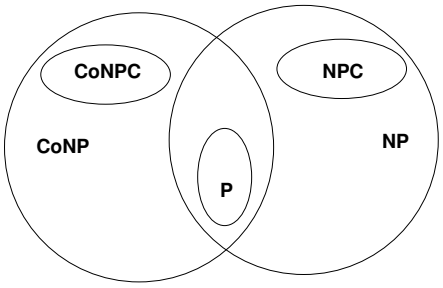

Fig. 19.5. The language families P, NP, and NPC. All these set inclusions are likely to be proper

A language L is said to be CoNP exactly when $\overline{L}$ is in NP. Similarly, L is said to be CoNPC exactly when $\overline{L}$ is in NPC. Figure 19.5 depicts

these additional language classes and their likely containments. To illustrate these ideas, consider the following languages which are both subsets of positive natural numbers $\{1, 2, 3, \ldots\}$:
The language

$$Primes = \{n \mid (n > 1) \wedge (\exists p, q > 1 : n = p \times q \Rightarrow p = 1 \vee q = 1\}.$$

The language

$$Composites = \overline{Primes},$$

where the complementation is with respect to positive naturals.
Composites is clearly in NP because there exists a P-time verifier for this language, given a certificate which is a pair of natural numbers suggested to be factors. In [98], Pratt proves that *Primes* are also in NP; he shows this result by demonstrating that there are polynomially long proofs for primes (given a prime p, a polynomially long sequence of proof steps can serve to demonstrate that p is such). Furthermore, he showed that such a proof for *Primes* can be checked in polynomial-time. Now, *Composites* is in CoNP because *Primes* is in NP, and *Primes* is in CoNP because *Composites* is in NP. The question now is: could either of these languages be NPC? Theorem 19.9 below, shows that even if there exists one such language, then NP and CoNP would become equal—a result thought to be highly unlikely. Surely enough, in 2002, Agrawal et al. [1] proved that *Primes* are in P (and hence *Composites* are also in P). Theorem 19.9 has helped anticipate the direction in which some of the open problems in this area would resolve.

Illustration 19.5.1 *(A Caveat)* Please bear in mind that recognizing a number to be composite in polynomial-time *does not*, by itself, give us the ability to *find* its prime factors in polynomial-time. Therefore, all public key crypto systems are still safe, despite Agrawal et al.'s result. Factoring a composite number into its prime factors can be expressed as a language

$$PrimeFactors = \{(x, i, b) \mid i^{th} \text{ bit of prime factorization of } x \text{ is } b\}.$$

Here, it is assumed that the prime factors of x are arranged in a sequence. Clearly, we do not want the *PrimeFactors* language to be in P. It can be easily shown that this language is in NP, however. □

Theorem 19.9. L is in NPC and L is in CoNP if and only if NP = CoNP.

Proof:
• To show that if L is NPC and CoNP then NP=CoNP.

- Assume L is NPC; therefore,
  . L is in NP
  . For all L_1 in NP, we have $L_1 \leq_P L$. Now, assuming $\overline{L}$ is in NP (because L is in CoNP), we have $\overline{L} \leq_P L$.
- Now, we are about to embark on showing the NP =CoNP part. For that, consider an arbitrary $L^{'}$ in NP. Then $L^{'} \leq_P L$.
- Now, using the result of Exercise 16.9, $\overline{L'} \leq_P \overline{L}$. Also $\overline{L'} \leq_P \overline{L} \leq_P L$.
- Now, since there is an NP decider for L, there is an NP decider for $\overline{L'}$ also, using the above mapping reduction chain; in other words $L^{'}$ is in CoNP.
- Now, consider an arbitrary $L^{'}$ in CoNP. This means that $\overline{L'}$ in NP. Since L is NPC, we have $\overline{L'} \leq_P L$. From this we have $L^{'} \leq_P \overline{L}$.
- Using the fact that $\overline{L} \leq_P L$, we have $L^{'} \leq_P \overline{L} \leq_P L$, or that there is an NP decider for $L^{'}$.
- Hence, NP = CoNP.

• To show that if NP =CoNP, then there exists an L that is NPC and CoNP. This is straightforward: consider any NPC language L; it would be CoNP because L is in NP and NP =CoNP.

Chapter Summary

We discussed the theory of NP-completeness, going through practical techniques to show that a problem is NP-complete. We now discuss the question of input encodings postponed in Section 19.2.6. In the setting of input encodings, there are basically two classes of problems:

- Strongly NPC: Those problems where the problem remains NPC even if the input is encoded in unary. Almost every NPC problem we have studied (e.g., *Clique*, 3-SAT, etc.), is strongly NPC. In addition, the 3-partition problem (discussed momentarily), several problems in the context of the game of Tetris [34], and several scheduling problems are strongly NPC.
- Not Strongly NPC, or *pseudo polynomial*: There are problems where encoding the input in unary can give a polynomial algorithm. The 2-partition problem is an example which has a pseudo polynomial algorithm.

The 2-partition problem is: Given a finite set $A = \{a_1, a_2, \ldots, a_n\}$ of positive integers having an overall sum of $2b$, is there a subset $A^{'}$ of

A that sums exactly to b (in other words, $A \setminus A^{'}$ and $A^{'}$ sum to b)? Note that we only determine the existence of such a subset—not *which* subset it is. Analogously, the 3-partition problem seeks three disjoint subsets that contain all the elements of A and sum to equal values.

2-partition is known to be NP-complete. However, there is a straightforward dynamic programming algorithm to solve the 2-partition problem, which is now briefly discussed. Stating things in genera, let g be the 'goal' in terms of a subset of the a_i's adding up to g. Let $T(j, g)$ denote the assertion that the sum of $\{a_1, \ldots, a_j\}$ is exactly g. We now write a recursive recipe to compute the truth of $T(i + 1, g)$. This falls into two cases:

1. We do not include a_{i+1}, and T(i, g); or
2. We include a_{i+1}, and the remaining elements add up to $g - a_{i+1}$.

We build a dynamic programming table following the above recurrence, as follows:

- $T(i + 1, g) = T(i, g) \ \vee \ ((a_{i+1} \leq g) \wedge T(i, g - a_{i+1}))$
- $T(1, g) = ((g = 0) \ \vee \ g = a_1)$

Now, the answer we seek—whether there exists a subset of a_1 through a_n that adds up to b—is the value of $T(n, b)$ when the above algorithm finishes.

This algorithm has complexity $\mathcal{O}(n.b)$, as there are that many entries to be filled in the memoization table T. Note that by encoding the problem in $O(n.b)$ bits, we can achieve polynomial-time solution. However, a *reasonable encoding* of this problem takes only $n\ log(b)$ bits. Therefore, by "bloating" the input representation, 2-partition can be solved in polynomial-time. No such luck awaits strongly NP-complete problems—they cannot be solved in polynomial-time even with a unary input representation (the most bloated of input representations). Further work on this topic may be easily found on the internet.

Exercises

19.1. Show that $TwoColor \in \mathsf{P}$.

19.2. Suppose we write a program that traverses a "tape" of n cells, numbered 1 through n. The program performs n traversals of the tape, with the ith traversal sequentially examining elements i through n. What is the runtime of such a program in the Big-O notation?

19.3. 1. Let $k = 2$. Estimate the magnitudes of x^k (polynomial) k^x (exponential) complexity growth for $x = 1, 2, 5, 10, 50$ and 100.

2. Estimate $2^{2^{2^{\cdots^{2^2}}}}$ (i times) for various i (note how to read this tower: begin with 2, and keep taking '$2^{previous}$').

19.4.

1. Draw an undirected graph of five nodes named a, b, c, d, e such that every pair of nodes has an edge between them (such graphs are called "cliques" - the above being a 5-clique).
2. What is the number of edges in an n-clique?
3. Which n-cliques are planar (for what values of n)?
4. What is a *Hamiltonian cycle* in an undirected graph?
5. Draw a directed graph G with nodes being the subsets of set $\{1, 2\}$, and with an edge from node n_i to node n_j if either $n_i \subseteq n_j$ or $|n_i| = |n_j|$. $|S|$ stands for the *size* or *cardinality* of set S.
6. How many strong components are there in the above graph? A strong component of a graph is a subset of nodes that are connected pairwise (reachable from one another).
7. What is the asymptotic time complexity of telling whether a directed graph has a cycle?

19.5. A Hamiltonian cycle in a graph with respect to a given node n is a tour that begins at n, visits all other nodes exactly once, returning to n. In a 5-clique, how many distinct Hamiltonian cycles exist? How about in an n-clique?

19.6.

1. Suppose you are sent into a classroom where n (honest) students are seated, and are patiently awaiting your arrival. You are charged by your boss with determining whether *some* k of these students know each other – any such subset of k students will do. Suppose you pick exactly one random subset of k of these students and each pair within this subset tells you that they know each other. Can you now report back your answer to the boss?
2. The above is a *nondeterministic* algorithm to check whether a graph has a k-clique. What would a *deterministic* algorithm be?
3. Suppose what you are charged with is to assure that *no k-subset* is such that all its members know each other pairwise. Suppose you pick exactly one random subset of k of these students and listen to them pairwise as to whether they know each other or not. Can you now report back any answer to your boss? How many more queries would you need as a function of n and k?

19.7. Define the language $HalfClique$ to be the set of input encodings $\langle G \rangle$ such that G is an undirected graph having a clique with at least

$n/2$ nodes, where n is the number of nodes in G. Show that $HalfClique$ is NPC.

19.8. Show that $\neq$-sat, as defined in Section 18.3.7, is NPC.

19.9. *Problems pertaining to* NPC *abound in various places; rather than repeat them, we leave it to the student / teacher to find and assign them suitably. We close off by assigning a rather interesting proof to read and understand.*

In his MS thesis, Jason Cantin (Wisconsin) proves that the problem of verifying memory coherence is NPC [15, 16]. Read and understand his proof.