

Introduction to the Theory of NP-Completeness

Rave Harpaz
Pattern Recognition Laboratory
Department of Computer Science
The Graduate Center
The City University of New York
365 Fifth Avenue
New York, N.Y. 10016

Oct. 10 2003

1 Introduction

Computational complexity is a branch of theoretical computer science that is concerned with classifying problems according to the computational resources, specifically time and memory, required to solve them. Since time requirements are often a dominant factor in determining whether a particular algorithm is efficient or not, most results in computational complexity focus on time complexity. This measure, in turn, depends on the algorithm's implementation as well as the computer on which the program is running. The theory of computational complexity and in particular the theory of *NP-completeness* provides us with a notion of complexity that is independent of implementation details and computer at hand.

2 Time Complexity

The time requirements of an algorithm are expressed in terms of the "size" of a problem's instance upon which the algorithm is applied. The size reflects the amount of data needed to describe the problem instance. After all, we would expect that the relative difficulty of solving a problem will increase with its input size, e.g. it takes more time to sort 100000 numbers rather than 10 numbers. Nonetheless, until we fix the encoding scheme, a procedure used to map problem instances into strings describing them, and the computer or model being used for determining execution time, the notion of "size" will remain ambiguous. However, we will see that these particular nuances will have no effect on the distinctions made in the theory of NP-completeness.

Since some algorithms perform better on particular instances of the same problem, e.g. sorting an array that is already almost sorted, we consider the *worst-case* time complexity to minimize the dependency on the specific instance.

Definition 1 (Time complexity). *Time complexity is a function $T : \mathbb{N} \rightarrow \mathbb{N}$, where $T(n)$ is the **maximum** number (worse case) of time units that an algorithm requires in order to solve a problem instance of input size- n .*

This definition introduces another term that needs clarification-*time units*. A measure of time complexity should be based on a unit of time that is independent of the specific CPU's clock rate. Such time units are the number of elementary operations an algorithm executes, where elementary operations are considered to be simple operations such as adding or comparing two integers. Thus, measuring time units amounts to counting the number of elementary

operations. This number in turn strongly depends on the algorithm's implementation details. Therefore rather than measuring the exact number $T(n)$ of elementary operations, we consider the asymptotic behavior of $T(n)$ as n gets very large. We say that $T(n)$ is of the order of $g(n)$ or that $g(n)$ is an **upper bound** for $T(n)$ and write $T(n) = \mathcal{O}(g(n))$ if there exists positive constants c and n_0 such that for all $n \geq n_0$, $T(n) \leq cg(n)$. Intuitively, $T(n) = \mathcal{O}(g(n))$ means that T is less than or equal to g if we discard differences up to a constant factor, i.e. \mathcal{O} represents a suppressed constant.

3 Tractability and Polynomial Time Complexity

One way of discriminating problems is by classifying them as *tractable* or *intractable*. In computational complexity problems that are solvable by a **polynomial algorithm**, an algorithm that has time complexity $\mathcal{O}(n^k)$ for some constant k are called tractable, i.e. an algorithm that requires time bounded by a polynomial in the length of the input n . Problems with non-polynomial time complexity, such as $\mathcal{O}(k^n)$ i.e. exponential algorithms or $\mathcal{O}(n!)$, are referred to as intractable problems. In essence the notion of tractability is used to separate problems that in practice have feasible solutions from those that don't.

This system of classification may be challenged on several grounds. One can argue that based on this system a problem with time complexity $\mathcal{O}(n^{100})$ should be classified as tractable whereas it is much more reasonable to regard it as intractable, $n = 10$ will already result a number that is comparable with the number of molecules in the universe or the number of nanoseconds since the "big bang". A similar argument can be raised for problems with time complexity $\mathcal{O}(n^{\log \log n})$. A further argument is that this system classifies problems based on a worst-case measure, such that a problem may have exponential time complexity but most of its instances require far less time, thus an average-case analysis (average time it takes to solve a problem over all possible instances) seems to be more appropriate and a better predictor of the practical utility of an algorithm.

Despite these reservations this method of classification has many appealing practical and theoretical advantages. Although it is reasonable to regard a problem that has time complexity $\mathcal{O}(n^{100})$ as intractable, or $\mathcal{O}(n^{\log \log n})$ as tractable, there are far and few practical problems that have this time complexity, most practical problems tend to have polynomial time complexity with polynomial coefficients equal to 2 or 3. In support of the worst-case criterion it is argued that no methods are known for predicting in advance if an exponential time algorithm will run faster in practice, i.e. predicting the portion of

instances that will require more time to be solved. Another contention is that utilizing average-case analysis requires us to determine which distributions of inputs to use for our analysis, a question with no absolute answer.

Among the advantages is what is commonly referred to as the *robustness* of P . The complexity class P has not been defined yet however the underlying idea is that a polynomial time complexity measure will be invariant under the change of model of computation we use. That is, if a problem is solved in polynomial time in one model it can be solved in polynomial time in another model, under the restriction that we are using a "reasonable" model, such as Turing machines, multi-tape Turing machines, random access machines (RAMs). Reasonable model in this context means that there is a polynomial bound on the amount of work that can be done in single unit of time.

Another advantage is that problems with polynomial time complexity are also invariant under different encoding schemes. Suppose an instance of a problem requires us to encode a graph, such an instance might be encoded by simply listing all vertices and edges, by an adjacency matrix or by an adjacency list, each producing different input length. However different encodings of the same problem producing different input length, will differ from one another by at most a polynomial amount. This again is true under the restriction that we are using a "reasonable" encoding scheme. In this context "reasonable" means that the encoding should be concise and not "padded" with unnecessary information (allowing us to convert a exponential time algorithm to a polynomial time one), and that we are using any base other than the expensive unary base to encode numbers. The reason we don't use unary is that unary encodings differ by an exponential factor from other base encodings, resulting unrealistically good bounds. Suppose for example we want to determine whether a number n is a prime number or not, a naive solution is to go through all numbers between 2 to \sqrt{n} and check if any of them are factors of n . If the problem is encoded in unary we will have to make $\mathcal{O}(\sqrt{n})$ comparisons since n is the input size, however if the problem is encoded in binary the input size is $k = \lceil \log n \rceil$ but we still have to make $\mathcal{O}(\sqrt{n})$ comparisons, therefore in terms of the input size k we will have to make $\mathcal{O}(2^{k/2})$ comparisons, thus turning the problem into an intractable one.

In summary we see that polynomial time complexity is a useful measure in discriminating problems as tractable/intractable while maintaining a notion that is independent of implementation details and computer at hand.

4 Decision Problems

Recall that an ordered pair is a set of a pair of objects with an order associated with them, and a binary relation R from a set A to a set B is a subset of all possible ordered pairs $\langle a, b \rangle$ such that $a \in A$, $b \in B$, i.e. $R \subseteq A \times B$. A problem Π may be depicted abstractly as a binary relation R_Π on the set of problem instances I and the set of problem solutions S (assuming each instance has a solution). Solving problem Π corresponds to searching over the binary relation R_Π , where the input is some $i \in I$ and the task is to find some $s \in S$ such that $\langle i, s \rangle \in R_\Pi$. Consider the *traveling salesman problem* (TSP), where a traveling salesman has a number of cities to visit, called a tour, where every city is visited only once except that he returns to the city from which he starts. The goal is to find a tour that minimizes the total distance the salesman has to travel among all possible tours. The set of problem instances for TSP are pairs each consisting of a set of cities to be visited and a distance matrix, the set of solutions are sequences of cities visited. The problem of finding shortest tour can be viewed as the relation that associates each pair (instance) with a sequence (solution), and formally may be written as $R_{TSP} = \{\langle i, s \rangle \mid i \in I, s \in S, s \text{ is a shortest tour for } i\}$.

Problems generally come in three different flavors: the *search* problem, that is finding a feasible solution (most of which are optimization problems such as TSP, the *decision* problem, that is determining whether a feasible solution exists, and the *verification* problem, i.e. deciding whether a given solution is correct. Much of complexity theory however, and the theory of *NP-completeness* in particular deals with decision problems, which are problems whose solutions are either *yes* or *no*, i.e. the solution set is $\{1, 0\}$. In this case a decision problem can be viewed as a function mapping instances into the set $\{1, 0\}$. The reason decision problems are preferred is due to their simplicity and their very natural formal counterpart- a *language*. By investigating simpler problems whose solutions are recognizable without comparisons to all feasible solutions like with optimization problems such as TSP, we hope to learn more about the barrier that separates tractable from intractable problems. Although most problems are not decision problems but rather optimization problems, they can be recast into decision versions by imposing a bound on the value to be optimized. As an example the problem TSP can be recast into TSP(D): given a graph, cost matrix and a bound B , does the graph contain a TSP tour of cost less than B . Although this process yields a different problem than the original optimization problem, we can still gain insight to the tractability of the optimization problem. If the optimization problem can be solved in polynomial time so can the decision version of it, simply by comparing the solution obtained with the bound. Taking the contrapositive which is more relevant to

the theory of *NP-completeness*, if we can provide evidence that the decision version is hard to solve or intractable then we have also shown that its related optimization problem is also hard to solve. A question that naturally arises is whether the converse is true, i.e. does an efficient solution for a decision problem guarantee an efficient one for its optimization counterpart. The answer is not known to be true in general, but can be shown to be true for all *NP-complete* problems, via self reducible relations (beyond the scope of this survey).

5 Formal Languages, and Turing Machines

Computational complexity can be discussed informally in terms problems and algorithms for solving them, and formally in terms of languages and Turing machines, where the languages correspond to problems, and the Turing machines to algorithms. As mentioned earlier the main reason for focusing on decision problems is that they have a very natural, formal counterpart, a formal-language. By using tools from formal-languages we will be able to express the relation between a decision problem and the algorithm that solves it in a mathematically rigorous way. Lets first review some of the basic definitions of formal language theory.

An *alphabet* Σ is a finite set of symbols, e.g. $\Sigma = \{0, 1\}$. We denote by Σ^* the set of all strings of symbols (words) from Σ , e.g. $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$ where ϵ is the empty string. A *language* L over Σ is a subset of Σ^* and the *empty language* is denoted by \emptyset .

The correspondence between languages and decision problems is the following: an encoding scheme (assuming a "reasonable" one as discussed earlier) of a problem Π may be thought of as a way of describing each instance I of Π by a string w of symbols of some fixed alphabet Σ_Π . This in turn induces a partition of Σ_Π^* into three classes of strings; those that encode instances of Π for which the answer is "yes" Y_Π , those for which the answer is "no" N_Π , and those that are not encodings of instances of Π . The language L_Π associated with decision problem Π is defined as Y_Π , i.e. $L_\Pi = Y_\Pi$, and the problem of determining whether the solution to an instance I of Π is "yes" or "no", can be cast into the formal-language *membership* problem of determining the membership of w in L_Π , i.e does $w \in L_\Pi$?

This correspondence also allows us to define complexity classes as classes of languages instead of problems, where, informally each class is classified by the amount of computational resources needed to solve the membership problem of each of its members (languages).

In order to define the notion of a complexity measure precisely we also need to formalize the notion of an algorithm, initially by fixing a particular model of computation. The standard model in computability theory is the Turing machine, introduced by Alan Turing in 1936. A *deterministic one-tape Turing machine* (DTM) consists of a finite state control (i.e. a finite program) attached to read/write head moving on an infinite tape. The tape is divided into squares, each capable of storing one symbol from a finite alphabet Γ which includes the blank symbol b . Each DTM has a specified input alphabet Σ , which is a subset of Γ , not including the blank symbol 'b'. At each step in a computation the DTM is in some state q in a specified finite set Q of possible states. Initially a finite input string over Σ is written on adjacent squares of the tape, all other squares are blank (contain 'b'), the head scans the leftmost symbol of the input string, and the DTM is in the initial state q_0 .

Formally a DTM is a tuple $\langle \Sigma, \Gamma, Q, \delta \rangle$ where Σ, Γ, Q are finite nonempty sets such that $\Sigma \subseteq \Gamma$, $b \in \Gamma - \Sigma$, and the state set Q contains three special states $q_0, q_{accept}, q_{reject}$. The transition function δ is

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 1\}$$

where $\delta(q, s) = (q', s', h)$ is to be interpreted as when the DTM is in state q scanning the symbol s then q' will be the new state, s' is the symbol printed, and the tape head moves right or left one square depending on whether h is 1 or -1 .

At each step of the computation the DTM is in some state q and the head is scanning a tape square containing some tape symbol s , and the action performed depends on the pair (q, s) and is specified by the DTM's transition function (or program) δ . The action consists of printing a symbol on the scanned square, moving the head left or right one square, and assuming a new state. If during the computation one of the states q_{accept} or q_{reject} are reached then the computation halts with either "yes" or "no" respectively. We say that a DTM program M with input alphabet Σ **accepts** $w \in \Sigma^*$ iff M halts in state q_{accept} . We denote the language **recognized** (or accepted) by M as

$$L_M = \{w \in \Sigma^* | M \text{ accepts } w\}$$

Note that if $w \in \Sigma^* - L_M$ then the computation may halt in state q_{reject} or never halt, i.e. loop forever. However, for a DTM program M to correspond to our notion of an algorithm, it must halt on all possible strings of its input alphabet. We say that a DTM program M **decides** $w \in \Sigma^*$ iff it either accepts or rejects (halts in state q_{reject}) w .

We can now state the correspondence between "recognizing" languages and solving decision problems. We say that a DTM program M solves the decision problem Π if M decides its input and $L_M = L_\Pi$.

6 Time Complexity Classes

6.1 Complexity Class \mathcal{P}

We are now ready for a formal definition of *time complexity* and the two most important time complexity classes, \mathcal{P} and \mathcal{NP} . The time used in the computation of a *DTM* program M on input w , denoted by $t_M(w)$, is the number of steps occurring in that computation until a halt state is reached. The time complexity of M is a function $T_M : \mathbb{N} \rightarrow \mathbb{N}$ given by

$$T_M(|w|) = \max\{t_m(w) \mid w \in \Sigma^{|w|}\}$$

where $|w|$ is the length of string w . A program M is called a **polynomial time** DTM program if there exists a constant k such that for any $n \in \mathbb{N}$, $T_M(n) = \mathcal{O}(n^k)$.

Definition 2 (Complexity Class \mathcal{P}).

$$\mathcal{P} = \{L \mid \text{there exists a polynomial time DTM program } M \text{ for which } L = L_M\}$$

We say that a decision problem Π belongs to \mathcal{P} , that is solvable in polynomial time, if $L_\Pi \in \mathcal{P}$.

6.2 Complexity Class \mathcal{NP}

Originally the class \mathcal{NP} was defined in terms of polynomial time nondeterministic Turing machines (*NDTM*), which are similar to *DTMs* except that the transition function δ now takes the form

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \rightarrow \Delta \subseteq (Q \times \Gamma \times \{-1, 1\})$$

that is, a machine that at any point in the computation has the choice to proceed according to more than one unique possibility for a given state and symbol scanned. One way of viewing the computation of a *NDTM* is as a tree whose branches correspond to parallel processes, one for each of the possible elements of Δ , thus a nondeterministic computation can potentially perform an exponential number of computations in polynomial time. Another common view is that the nondeterministic computation consists of two separate stages, guessing and checking. Given an instance I the first stage merely "guesses" a solution, which together with I is then fed into the checking stage that checks in a normal deterministic manner whether or not the guessed solution is a solution to the problem. We say that a nondeterministic algorithm solves a decision problem Π if the following properties for all $I \in \Pi$ hold:

- if $I \in Y_{\Pi}$, then there exists at least one guess that will lead the checking stage to respond "yes", or equivalently at least one branch of the computation tree that leads to a "yes" answer.
- if $I \notin Y_{\Pi}$, then there is no guess that will cause the checking stage to respond "yes", or equivalently all branches of the computation tree lead to a "no" answer.

We say that nondeterministic algorithm solves decision problem Π in polynomial time if the checking stage returns an answer in time bounded by a polynomial, which in turn imposes a polynomial bound on the length of the guess, or equivalently the length of any branch in the computation tree is bounded by a polynomial. The class \mathcal{NP} can now be defined informally as the class of all decision problems that can be solved by polynomial time nondeterministic algorithms.

Both these Traditional views come to show that the power of nondeterminism lies in the idea that it allows the exhaustive enumeration of an exponentially large number of candidate solutions in polynomial time, and if the evaluation of each candidate solution can be done in polynomial time then the total time for solving a problem is polynomial.

A more recent approach better suited for proving membership in \mathcal{NP} , and better captures the conceptual contents of the class, is the notion of polynomial time "verifiability" that the class \mathcal{NP} intends to isolate. Informally, we can view the class \mathcal{NP} as the class of languages that admit short *certificates* for membership in the language. Given this certificate, called *witness*, membership in the language can be verified in polynomial time. This certificate can be thought of as the string corresponding to the guess made by the *NDTM* mentioned earlier, and as a result must be succinct, that is its length must be bounded by a polynomial, and checkable in polynomial time.

We formalize this notion using a *checking/verifying relation* which is simply a binary relation $R \subseteq \Sigma_x^* \times \Sigma_w^*$ for some finite alphabets Σ_x and Σ_w . The corresponding language associated with R and defined over $\Sigma_x \cup \Sigma_w \cup \{\#\}$ is

$$L_R = \{x\#w \mid \langle x, w \rangle \in R\}$$

where $\#$ used to separate the input x from the witness w and therefore is not included in either Σ_x or Σ_w .

Definition 3 (Complexity Class \mathcal{NP}). *The complexity class \mathcal{NP} is the set of all languages L , each over some Σ_x , that satisfy the following conditions.*

1. *there exists a checking relation R for L where $L_R \in \mathcal{P}$*

2. $\forall x \in \Sigma_x^*, \exists k \in \mathbb{N} (x \in L \Leftrightarrow \exists w (|w| \leq |x|^k \text{ and } \langle x, w \rangle \in R))$

The first condition states that the certificate must be checkable in polynomial time, while the second condition ensures that every "yes" instance of a problem must have a certificate, which furthermore is succinct, and for "no" instances there is no such certificate.

Using this definition of \mathcal{NP} it can easily be shown that $\mathcal{P} \subseteq \mathcal{NP}$, by showing that for any language L defined over Σ , if $L \in \mathcal{P}$ then $L \in \mathcal{NP}$. Let the checking relation for L be $R = \{\langle x, y \rangle \mid x \in L, y \in \Sigma^*, |y| \leq |x|^k\}$ (y is some arbitrary bounded string). $L \in \mathcal{P}$ entails that there exists a polynomial time *DTM* program M that recognizes L , using this M we can create a new program M' that simply ignores a portion of the input (the portion to the right of #), and proceeds to compute according to M . Thus we have showed that $L_R \in \mathcal{P}$, fulfilling the first requirement of the definition of \mathcal{NP} , where the second requirement holds by definition of R .

In this view of the complexity class \mathcal{NP} , the most famous and important question in computer science, $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ can be formulated as the question whether the existence of a succinct witness verifiable in polynomial time (as implied by membership in \mathcal{NP}) necessarily brings about an efficient algorithm for finding it (as required for membership in \mathcal{P}), i.e. $\mathcal{NP} \subseteq \mathcal{P}$?

7 Reducibility and \mathcal{NP} -completeness

A Reduction is a mathematical tool with which the relative complexity of problems are compared. The idea behind reductions is the transformation of one problem Π_1 to another Π_2 in such a way that if Π_2 is known to be easy, so is Π_1 , and vice versa, if Π_1 is known to be hard so is Π_2 . In practical terms, the notion of a reduction from Π_1 to Π_2 is the ability to use an algorithm for solving Π_2 as a subroutine for solving Π_1 . We will see that utilizing this idea leads to the result that there are some intractable problems that are complexity wise equivalent to each other. These problems are the so called \mathcal{NP} -complete problems, which seem to embody the secret of intractability in a way that an efficient algorithm for solving any one of them will immediately imply the tractability of all problems in \mathcal{NP} .

There several types of reductions commonly known as *Turing*, *Cook*, *Karp*, *Levin*, and *log-space* reductions. The two most significant to the theory of \mathcal{NP} -completeness are the *Cook*, *Karp* reductions.

Definition 4 (Oracle Turing Machine). *An Oracle Turing Machine (OTM) denoted M^A is a Turing machine with an extra tape called the oracle tape for*

language A , and three extra states $q_?$, q_Y , q_N . The machine can write a string w on the oracle tape, and enter the query state $q_?$, then in one step transfer control to state q_Y or q_N depending whether $w \in A$ or $w \notin A$.

Informally, an oracle for language A can be viewed as a magical device that can answer membership problems for language A (decide A) in a single time unit (single step).

Definition 5 (Cook Reduction). *A Cook reduction from problem Π_1 to problem Π_2 denoted $\Pi_1 \leq_T^p \Pi_2$ is an OTM that in polynomial time solves problem Π_1 on input x while getting oracle answers for problem Π_2 .*

Definition 6 (Karp Reduction). *A Karp reduction (also called many to one reduction) of language $L_1 \subseteq \Sigma_1^*$ to language $L_2 \subseteq \Sigma_2^*$ denoted $L_1 \leq_m^p L_2$, is a polynomial time computable function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ such that $x \in L_1 \Leftrightarrow f(x) \in L_2$. [Kar72]*

The \leq symbol is used to emphasize that one problem is at least as hard as the other. The A Karp reduction is also called "many to one reduction" because the transformation function f is a many to one function, mapping potentially different instances of one problem to the same instance of another.

A Karp reduction can be viewed as a more realistic, constraint and special case of a Cook reduction where the oracle may be queried only once, as opposed to many times like with the Cook reduction. Formally this can be written as $\leq_m^p \subseteq \leq_T^p$, which seems to imply that a Cook reduction has more computing power than a Karp reduction. It is an open question however whether that is the case. Let $co\text{-}\mathcal{NP} = \{L | \bar{L} \in \mathcal{NP}\}$ denote the class of problem whose complement is in \mathcal{NP} , another open problem is $\mathcal{NP} \stackrel{?}{=} co\text{-}\mathcal{NP}$, it conjectured that they are not (discussion beyond the scope of this survey), however it is easy to see that if $\leq_m^p = \leq_T^p$ then $\mathcal{NP} = co\text{-}\mathcal{NP}$.

Lemma 1. *if $L_1 \leq_m^p L_2$ and $L_2 \in \mathcal{P}$ then $L_1 \in \mathcal{P}$.*

The proof is trivial and the idea is to use the DTM for deciding L_2 as a subroutine for deciding L_1 . Since deciding L_2 takes polynomial time and the transformation of a string from L_1 to a string from L_2 also takes polynomial time then the total time is also polynomial, thus deciding L_1 takes polynomial time. This result comes to show, as stated earlier that an easy solution for one problem can imply an easy solution for another.

Definition 7 (\mathcal{NP} -complete). *A language L is \mathcal{NP} -complete if and only if:*

1. $L \in \mathcal{NP}$

2. for every language $L' \in \mathcal{NP}$, $L' \leq_m^p L$

These languages are the hardest problems in \mathcal{NP} , in the sense that if we knew how to solve an \mathcal{NP} -complete problem efficiently we can efficiently solve any problem in \mathcal{NP} , which is merely an application of lemma 1. The following result shows the converse, informally the idea is that if one problem is known to be hard, and it reduces to another then so is the other problem hard. This result also shows how to prove that a problem is \mathcal{NP} -complete, by finding another problem known to be \mathcal{NP} -complete and a polynomial transformation of the known problem to the new problem.

Lemma 2. *if L_1 is \mathcal{NP} -complete, and $L_2 \in \mathcal{NP}$, and $L_1 \leq_m^p L_2$ then L_2 is \mathcal{NP} -complete.*

This result follows by the transitivity of Karp reductions.

8 Cook's Theorem

The theory of \mathcal{NP} -completeness is primarily used to compare the relative hardness of problems, where lemma 1 and lemma 2 provide us with the mechanism of establishing such relations. However without identifying the first \mathcal{NP} -complete problem the whole theory of \mathcal{NP} -completeness is somewhat sterilized. Such a problem was provided by Cook's theorem.

Definition 8. *Let $X = x_1, x_2, \dots, x_n$ be a finite set of **Boolean variables**, and let $\bar{X} = \bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$ stand for the negations of x_1, x_2, \dots, x_n . we call the elements of $X \cup \bar{X}$ **literals**. We call a set of literals a **clause** C , and a set of clauses a **formula** ϕ . We say that a formula is in **conjunctive normal form** (CNF), if $\phi = \bigwedge_{i=1}^n C_i$, where $n \geq 1$, and each C_i is the disjunction of one or more literals.*

Definition 9. *A truth assignment T for ϕ is a mapping $T : X \rightarrow \{1, 0\}$. A literal u is true under T ($T \models u$) iff $T(u) = 1$. A truth assignment T for X satisfies a clause $C \in \phi$, where ϕ is a Boolean formula in CNF, iff at least one literal $u \in C$ is true under T . We say that T satisfies ϕ iff it satisfies every clause in ϕ .*

Definition 10 (SAT problem).

Instance: A Boolean formula ϕ in conjunctive normal form.

Question: Does there exist a satisfying truth assignment for ϕ ?

Theorem 1 (Cook's Theorem). *SAT $\in \mathcal{NP}$ -complete [Coo71]*

Proof (outline): We first show that $SAT \in \mathcal{NP}$. A nondeterministic algorithm for it will guess a truth assignment and check to see whether that assignment satisfies the formula which was the input to the algorithm. It is easy to see that this can be accomplished in polynomial time and that the length of the guess is less than the size of the input.

Showing that every \mathcal{NP} language reduces to the SAT language cannot be achieved by presenting a reduction for each of them, since there are infinitely many of them. However each one of them has a NDTM program that recognizes it, thus we construct a generic reduction that takes the input string and produces a Boolean formula that simulates the NDTM program that recognizes the \mathcal{NP} language with that input. If the machine accepts the formula produced will have a satisfying assignment that correspond to the accepting computation, and if the machine does not accept, no assignment will satisfy the formula produced. A tableau for NDTM program M on input w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of M on w . A tableau is accepting if any of its rows is an accepting configuration. The problem of determining whether M accepts w is thus equivalent to the problem of determining whether an accepting tableau for M on w exist. Therefore we generate a formula that simulates M on input w by generating a formula such that a satisfying assignment for it corresponds to an accepting tableau for M on w , and no satisfying assignment for it corresponds to the fact that M does not accept w . This formula in turn, is a conjunction of four smaller formulas that guarantee that certain conditions hold for the correspondence between an assignment and a tableau. The first guarantees that any satisfying assignment specifies one and only one symbol for each cell of the tableau. The second formula ensures that the first row of the tableau is a starting configuration. The third ensures that an accepting configuration occurs in the tableau. Finally, the fourth formula guarantees that each row of the tableau corresponds to a configuration that legally follows from M 's transition function, by constructing a formula for every 2×3 window of cells, and taking the conjunction of all such windows. Showing that the construction of the formula took polynomial time follows from the fact the the length of the formula is bounded by a polynomial in the length of w .

9 \mathcal{NP} -hard problems

Although the definition of \mathcal{NP} -complete seems to be fairly unified , the definition of \mathcal{NP} -hard is somewhat less so. One of the definitions states that \mathcal{NP} -hard is the set of languages that satisfy only (but not necessarily property 1) of definition 7 (\mathcal{NP} -complete), i.e. \mathcal{NP} -hard refers to the class of decision

problems that contains all problems L such that for all decision problems L' in NP there is a polynomial-time many-one reduction (Karp reduction) to L . Informally this class can be described as containing the decision problems that are at least as hard as any problem in NP. It is easy to see that according to this definition all \mathcal{NP} -complete problems are also \mathcal{NP} -hard, but there are not many problems that according to this definition are \mathcal{NP} -hard but not \mathcal{NP} -complete. One problem that has this property is the famous *halting problem*, given a program and its input, will it come to a halt or run forever? to show that it is \mathcal{NP} -hard we reduce *SAT* to it by transforming a *SAT* instance to the description of a Turing machine that tries all truth assignments for the *SAT* instance and when it finds one it halts, otherwise it goes into an infinite loop. It is also easy to see that the halting problem is not in NP since all problems in NP are decidable and the halting problem is not.

An alternative definition of \mathcal{NP} -hard which seems to have wider acceptance is one which extends the previous definition to also include search problems and not only decision problems, making it generally more useful. We say that a problem (search or decision) is \mathcal{NP} -hard if solving it in polynomial time would make it possible to solve all problems in class \mathcal{NP} in polynomial time. Since the definition does not require that such a polynomial time solution exist, it implicitly replaces Karp reductions with Cook/Turings reduction where the oracle can compute any function, not only functions mapping to $\{0,1\}$. Note however that under this definition all problems in $co\text{-}\mathcal{NP}$ are also \mathcal{NP} -hard.

Yet another definition, very similar to the previous says that a problem is \mathcal{NP} -hard if its decision version is known to belong to \mathcal{NP} -complete.

10 Historical Note

The existence of NP-complete problems was proved independently by Stephen Cook in the United States and Leonid Levin in the Soviet Union. Cook, then a graduate student at Harvard, first identified the languages which we now call P and NP, and showed that several natural problems, including Satisfiability, and subgraph isomorphism are NP-complete. Meanwhile, Levin, a student of Kolmogorov at Moscow State University, proved that a variant of the tiling problem is NP-complete. Cook used in his proofs a different notion of reducibility, generating a similar class of problems, albeit believed to be bigger than the class presently known as NP-complete. He used polynomial Turing reductions which he termed "P-reductions", and originally meant to show that all polynomial time nondeterministic computations can be Turing reducible to the problem of determining if a propositional formula given in disjunctive normal form is tautology, where satisfiability result result was an intermedi-

ate result. It is also for this reason that Cook and Turing reduction are often considered the same thing. It was Richard Karp in 1972, in a tremendously influential paper, that the theory of NP-completeness took on its present form. Karp introduced the terms P and NP, and showed that Cook's theorem would hold if Cook reductions are replaced by the simpler and more realistic many to one reductions, which we call Karp reductions. He also showed that eight central combinatorial problems, including clique, independent set, set cover, and the traveling salesman problems are NP-complete. The terminology used today, such NP-complete, and NP-hard are primarily due to Donald Knuth's efforts of standardizing these terms.[FH03]

References

- [Coo71] Stephen A. Cook, *The complexity of theorem-proving procedures*, In Conference Record of Third Annual ACM Symposium on Theory of Computing (1971), pages 151–158.
- [FH03] L. Fortnow and S. Homer, *A short history of computational complexity*, The History of Mathematical Logic (D. van Dalen, J. Dawson, and A. Kanamori, eds.), North-Holland, Amsterdam, 2003, To appear.
- [Kar72] Ricard M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (K.E. Miller and J.W. Thatcher, eds.), Plenum Press, New York, 1972, pp. 85–103.