Chapter 6:

Complexity Theory

Computational complexity theory is branch of theory of computation in computer science and mathematics that focus on classifying computational problems according to their inherent difficulties or complexity. It includes

- the efficiency of algorithms
- the inherent difficulty of problems of practical and or theoretical importance

In other words, Computational complexity theory is a part of theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are

- time (how many steps it takes to solve a problem)
- space(how much memory it takes)

There are two kinds of measures with Complexity Theory: (a) time and (b) space.

(i) ***Time Complexity***: It is a measure of how long a computation takes to execute. As far as Turing machine is concerned, this could be measured as the number of moves which are required to perform a computation. In the case of a digital computer, this could be measured as the number of machine cycles which are required for the computation.

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

(ii) ***Space Complexity***: It is a measure of how much storage is required for a computation. In the case of a Turing machine, the obvious measure is the number of tape squares used, for a digital computer, the number of bytes used.

Let M be a deterministic Turing machine that halts on all inputs. The ***space complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$.

If M is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

It is to be noted that both these measures functions of a single input parameter, viz., "size of the input", which is defined in terms of squares or bytes. For any given input size, different inputs require different amounts of space and time. Thus it will be easier to discuss about the "average case" or for the "worst case". It is usually interesting to look at the worst-case complexity because

(a) It may be difficult to define an "average case"

(b) Usually easier to compute worst-case complexity.

Computability theory deals only with whether a problem can be solved at all , regardless of the resources required .

Much of complexity theory deals with decision problems A decision problem is a problem where the answer is always yes /on

Polynomial Time

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is O(n^k) for some nonnegative integer k, where n is the complexity of the input. Polynomial-time algorithms are said to be "fast." Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time. Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

Class P and Class NP

The Class P

Computing practice reveals that many problems which are solvable in principle can not be solved in practice due to the excessive time requirement.

For example,

In a Travelling sales man problem if there are n cities to visit, it requires (n-1)! itineraries (record of root of journey) If there are 10 cities , it requires 9!= 362880 itineraries .However if there are 40 cities it requires greater than 10^{45} itineraries Hence we can say that the problems which are theoretically compostable are not practically feasible at all .

In order to be a practically feasible algorithm, we must limit our computational devices to only run for a number of steps that is bounded by a **polynomial** in the length of inputs.

We can define solvable problems as:

When the space and time required for implementing the steps of the particular algorithm are reasonable, then we can say that problem is solvable or tractable in practice.

Polynomially bounded Turing Machine

A Turing machine M= (K,Σ, δ, s ,H) is said to be polynomially bounded Turing machine if there is a polynomial P(n) such the following is true For any input x, there is no configuration C such that

$$(s, \triangleright \underline{\sqcup} x) \vdash_M^{p(|x|)+1} C.$$

In other words , machine always halts after P(n) steps, where n is the length of the input.

A language is called polynomially decidable if there is a polynomially bounded Turing Machine that decides it. The class of all polynomially decidable languages are denoted by P.

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

The class P plays a central role in our theory and is important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

Theorem : P is closed under complement

Proof: If a language L is decidable by a polynomial bounded Turing machine M then its complement is decidable by the version of Turing machine that inverts yes and no. Obviously the polynomial bound is unaffected.

Examples of Problems in class P

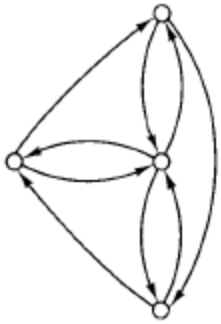
There are several examples of problems in class P such as:

1. Eulerian and Hamiltaon graph Problem
2. Optimization problems
3. Integer Partition Problems
4. Equivalence of Finite automata
5. Kruskals algorithm : minimum weight spanning tree.

For more detail on this read from book

Eulerian Graph

A graph G is *Eulerian if and only if it has the following two properties:*
(a) For any pair of nodes u, v Σ *V neither of whieh* is *isolated, there* is *a path from u to v; and*
(b) All nodes have equal numbers of incoming and outgoing edges.



The Class NP

A non deterministic Turing machine M = = (K,Σ, δ, s ,H) is said to be polynomially bounded if there is a polynomial P(n) such that for any input x , there is no configuration C of M with

$$(s, \triangleright \sqcup x) \vdash_M^{p(|x|)+1} C$$

That is , no computation of this machine continuous for more than polynomially many steps.
Now we can define NP as
We can define NP (for non deterministic polynomial) to be the class of all languages that are decidable by a polynomially bounded Turing Machine.

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k).$$

Example of Class NP

1. **Travelling Sales man Problem**

NON-DETERMINISTIC POLYNOMIAL (NP) TIME ALGORITHMS

A nondeterministic computation is viewed as:

(i) when a choice point is reached, an infallible oracle can be consulted to determine the right option.

(ii) When a choice point is reached, all choices are made and computation can proceed simultaneously.

A Non-deterministic Polynomial Time Algorithm is one that can be executed in polynomial time on a nondeterministic machine. The machine can either consult an oracle in constant time, or it can spawn an arbitrarily large number of parallel processes, which is obviously a nice machine to have.

ADDITIONAL NP PROBLEMS

The following problems have a polynomial-time solution, but an exponential-time solution on a deterministic machine. There are literally hundreds of additional examples.

(a) ***The Travelling Salesman Problem (TSP):*** A salesman starting in Texas, wants to visit every capital city in the United States, returning to Texas as his last stop. In what order should he visit the capital cities so as to minimize the total distance travelled?

(b) ***The Hamiltonian Circuit Problem:*** Every capital city has direct air flights to at least some capital cities. Our intrepid salesman wants to visit all the capitals, and return to his starting point, taking only direct air flights. Can he find a path that lets him do this?

(c) ***Linear Programming:*** We have on hand X amount of butter, Y amount of flour, Z eggs etc. We have cookie recipies that use varying amounts of these ingredients. Different kinds of cookies bring different prices. What mix of cookies should we make in order to maximize profits?

NP-COMPLETE PROBLEMS

All the known NP problems have a remarkable characteristic: They are all reducible to one another. What this means is that, given any two NP problems X and Y,

(a) There exists a polynomial-time algorithm to restate a problem of type X as a problem of type Y, and

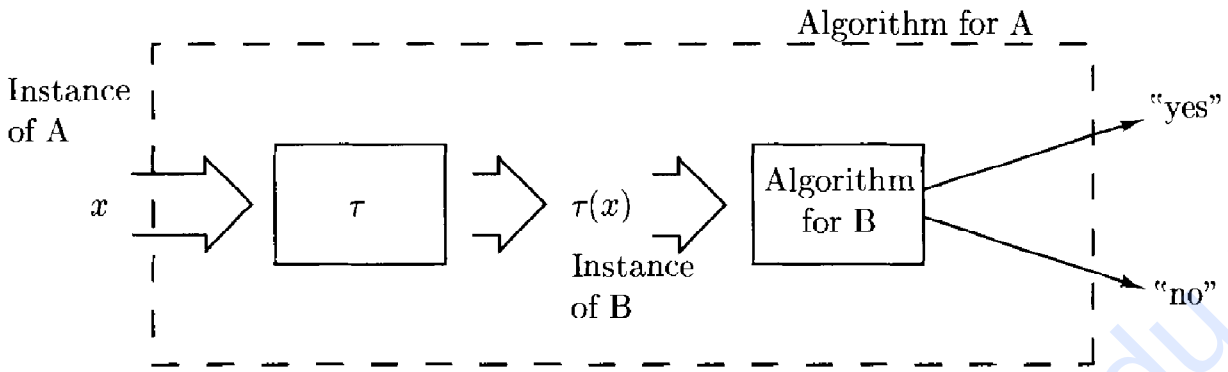
(b) There exists a polynomial-time algorithm to translate a solution to a type Y problem back into a solution for the type X problem.

POLYNOMIAL-TIME REDUCTIONS

Definition 7.1.1: A function $f : \Sigma^* \mapsto \Sigma^*$ is said to be **polynomial-time computable** if there is a polynomially bounded Turing machine M that computes it.

Let now $L_1, L_2 \subseteq \Sigma^*$ be languages. A polynomial-time computable function $\tau : \Sigma^* \mapsto \Sigma^*$ is called a **polynomial reduction from L_1 to L_2** if for each $x \in \Sigma^*$ the following holds: $x \in L_1$ if and only if $\tau(x) \in L_2$.

In this sense we can say that *T* is a polynomial reduction from Problem A to Problem B if it is a polynomial reduction between the corresponding languages. That is, *T* transforms in polynomial time instances of Problem A to instances of Problem B in such a way that *x* is a "yes" instance of Problem A if and only if *T(X)* is a "yes" instance of Problem B.



DEFINITION OF NP-COMPLETENESS

Definition 7.1.2: A language $L \subseteq \Sigma^*$ is called $\mathcal{NP}$**-complete** if
(a) $L \in \mathcal{NP}$; and
(b) for every language $L' \in \mathcal{NP}$, there is a polynomial reduction from L' to L.

A language B is NP-complete if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

Summary to common time complexities:

Complexity	Verbal Description	Feasilality
$O(1)$	constant time	feasible
$O(\log n)$	log time	feasible
$O(n)$	linear time	feasible
$O(n \log n)$	log linear time	feasible
$O(n^2)$	quadratic time	sometimes feasible
$O(n^3)$	cubic time	less often feasible
$O(2^n)$	exponential time	rarely feasible

Example of Polynomial Time

- $O(n)$, $O(n^2 + n)$, $O(10^{100}n)$

Example of Non polynomial Time

- $O(e^{2n})$, $O(2^n)$, $O(e^{n/1000})$

NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A

problem *H* is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H (i.e., $L \leq_T H$).

NP-hard problems may be of any type: decision problems, search problems, or optimization problems.

As consequences of definition, we have (note that these are claims, not definitions):

- Problem *H* is at least as hard as L, because H can be used to solve *L*;
- Since *L* is NP-complete, and hence the hardest in class NP, also problem *H* is at least as hard as NP, but *H* does not have to be in NP and hence does not have to be a decision problem (even if it is a decision problem, it need not be in NP);
- Since NP-complete problems transform to each other by polynomial-time many-one reduction (also called polynomial transformation), all NP-complete problems can be solved in polynomial time by a reduction to *H*, thus all problems in NP reduce to *H*; note, however, that this involves combining two different transformations: from NP-complete decision problems to NP-complete problem *L* by polynomial transformation, and from *L* to *H* by polynomial Turing reduction;
- If there is a polynomial algorithm for any NP-hard problem, then there are polynomial algorithms for all problems in NP, and hence P = NP;
- If P ≠ NP, then NP-hard problems have no solutions in polynomial time, while P = NP does not resolve whether the NP-hard problems can be solved in polynomial time;
- If an optimization problem H has an NP-complete decision version *L*, then *H* is NP-hard.

A common mistake is to think that the *NP* in *NP-hard* stands for *non-polynomial*. Although it is widely suspected that there are no polynomial-time algorithms for NP-hard problems, this has never been proven. Moreover, the class NP also contains all problems which can be solved in polynomial time.

Examples: An example of an NP-hard problem is the decision subset sum problem, which is this: given a set of integers, does any non-empty subset of them add up to zero? That is a decision problem, and happens to be NP-complete. Another example of an NP-hard problem is the optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph. This is commonly known as the Traveling Salesman Problem.

There are decision problems that are NP-hard but not NP-complete, for example thehalting problem. This is the problem which asks "given a program and its input, will it run forever?" That's a *yes*/*no* question, so this is a decision problem. It is easy to prove that the halting problem is *NP-hard* but not *NP-complete*. For example, theBoolean satisfiability problem can be reduced to the halting problem by transforming it to the description of a Turing machine that tries all truth valueassignments and when it finds one that satisfies the formula it halts and 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 in a finite number of operations, while the halting problem, in general, is not. There are also NP-hard problems that are neither NP-complete nor undecidable. For instance, the language of True quantified Boolean formulas is decidable in polynomial space, but not non-deterministic polynomial time (unless NP = PSPACE).

NP-naming convention

NP-hard problems do not have to be elements of the complexity class NP, despite having *NP* as the prefix of their class name. The *NP*-naming system has some deeper sense, because the NP family is defined in relation to the class NP and the naming conventions in the Computational Complexity Theory:

NP

Class of computational problems for which solutions can be computed by a non-deterministic Turing machine in polynomial time (or less). Or, equivalently, those problems for which solutions can be checked in polynomial time by a deterministic Turing machine.

NP-hard

Class of problems which are at least as hard as the hardest problems in NP. Problems in NP-hard do not have to be elements of NP, indeed, they may not even be decision problems.

NP-complete

Class of problems which contains the hardest problems in NP. Each element of NP-complete has to be an element of NP.

NP-easy

At most as hard as NP, but not necessarily in NP, since they may not be decision problems.

NP-equivalent

Exactly as difficult as the hardest problems in NP, but not necessarily in NP.