CHAPTER -1

Introduction to Computational Complexity Theory

2.1 Introduction

Complexity Theory is a field of study shared by *mathematics* and *computer science* that is concerned with how the computational complexity of problems increases as the number of cases involved increases and with the classification of the problems according to whether a solution can be found in *polynomial time* and to *the algorithms* required for such a solution.

Computationai complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. A computational problem is understood to be a task that is in principle amenable to being *solved by a computer*, which is equivalent to stating that the problem may be *solved by mechanical application of mathematical steps, such as an algorithm.*

A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as *time* and *storage* Other complexity measures are also used, such as the amount of communication (used in *communication complexity*), the number of gates in a circuit (used in *circuit complexity*) and the number of processors (used in *parallel computing*). One of the roles of *computational complexity theory* is to determine the *practical limits* on what computers can and cannot do.

Closely related fields in theoretical computer science are *analysis of algorithms* and *computability theory*. A key distinction between analysis of algorithms and computational complexity theory is that the former is devoted to analyzing the *amount of resources* needed by a particular algorithm to solve a problem, whereas the latter asks a more general question about all possible algorithms that could be used to solve the same problem. More precisely, it tries to classify problems that can or cannot be solved with appropriately restricted resources. In turn, imposing restrictions on the available resources is what distinguishes computational complexity from computability theory: the latter theory asks what kind of problems can, in principle, be solved algorithmically.

CamScanner

2.2 Computational Problems

In theoretical computer science, a *computational problem* is a mathematical object representing a collection of questions that *computers* might want to solve.

For example, the problem of *factoring*:

> "Given a positive integer n, find a nontrivial prime factor of n." is a computational problem.

Computational problems are one of the main objects of study in theoretical computer science. The field of *algorithms* studies methods of solving computational problems efficiently. The complementary field of *computational complexity* attempts to explain why certain computational problems are intractable for computers.

2.2.1 Problem Instances

A *computational problem* can be viewed as an infinite collection of *instances* together with a *solution* for every instance. The input string for a computational problem is referred to as a *problem instance*, and should not be confused with the problem itself. In computational complexity theory, a *problem* refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem. For example, consider the problem of *primality testing*. The instance is a number (e.g. *15*) and the solution is "*yes*" if the number is prime and "*no*" otherwise (in this case "*no*"). Stated another way, the *instance* is a particular input to the problem, and the *solution* is the output corresponding to the given input.

To further highlight the difference between a problem and an instance, consider the following instance of the decision version of the *traveling salesman problem*: Is there a route of at most 2000 kilometers passing through all of Ethiopia's 15 largest cities? The quantitative answer to this particular problem instance is of little use for solving other instances of the problem, such as asking for a round trip through all sites in *Addis Ababa* whose total length is at most 30 km. For this reason, complexity theory addresses computational problems and not particular problem instances.

CamScanner

2.2.2 Representing Problem Instances

When considering computational problems, a problem instance is a *string* over an *alphabet*. Usually, the alphabet is taken to be the binary alphabet (i.e., the set {0, 1}), and thus the strings are *bit strings*. As in a real-world computer, mathematical objects other than bit strings must be suitably encoded. For example, *integers* can be represented in *binary notation*, and *graphs* can be encoded directly via their *adjacency matrices*, or by encoding their *adjacency lists* in binary.

Even though some proofs of complexity-theoretic theorems regularly assume some concrete choice of input encoding, one tries to keep the discussion abstract enough to be independent of the choice of encoding. This can be achieved by ensuring that different representations can be transformed into each other efficiently.

2.2.3 Types of Computational Problems

1. Decision Problem

A decision problem is a computational problem where the answer for every instance is either "yes" or "no". An example of a decision problem is *primality testing*:

"Given a positive integer n, determine if n is prime."

A decision problem is typically represented as the set of all instances for which the answer is *yes*. For example, primality testing can be represented as the infinite set

$L = \{2, 3, 5, 7, 11, \ldots.\}$

2. Search Problem

In a search problem, the answers can be *arbitrary strings*. For example, factoring is a search problem where the instances are (string representations of) positive integers and the solutions are (string representations of) collections of primes.

A search problem is represented as a relation over consisting of all the instance-solution pairs, called a *search relation*. For example, factoring can be represented as the relation

$R = \{(4, 2), (6, 2), (6, 3), (8, 2), (9, 3), (10, 2), (10, 5)...\}$

which consist of all pairs of numbers (n, p), where p is a nontrivial prime factor of n.

CamScanner

3. Counting Problem

A counting problem asks for the *number* of solutions to a given search problem. For example, a counting problem associated with factoring is:

"Given a positive integer n, count the number of nontrivial prime factors of n."

A counting problem can be represented by a function f from $\{0, 1\}^*$ to the non-negative integers. For a search relation R, the counting problem associated to R is the function:

$f_R(x) = |\{y: (x, y) \in R\}|$.

4. Optimization Problem

An optimization problem is the problem of finding the "*best solution*" from all feasible solutions to a search problem. One example is the *maximum independent set* problem:

"Given a graph G, find an independent set of G of maximum size."

Optimization problems can be represented by their search relations.

Optimization problems can be divided into two categories depending on whether the *variables* are *continuous* or *discrete*.

a) Combinatorial optimization problem

An optimization problem with discrete variables is known as a *combinatorial optimization problem*. In a combinatorial optimization problem, we are looking for an object such as an *integer*, *permutation* or *graph* from a finite (or possibly countable infinite) set.

Formally, a combinatorial optimization problem A is a quadruple (I, f, m, g), where

- I is a set of instances;
- Given an instance $x \in I$, $f(x)$ is the set of feasible solutions;
- Given an instance x and a feasible solution y of x, $m(x, y)$ denotes the measure of y, which is usually a positive real.
- g is the goal function, and is either $\min$ or $\max$.

The goal is then to find for some instance x an *optimal solution*, that is, a feasible solution y with:

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\}.$$

CamScanner

For each combinatorial optimization problem, there is a corresponding *decision problem* that asks whether there is a feasible solution for some particular measure m_0.

b) Continuous Optimization Problem

The *standard form* of a (continuous) optimization problem is:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p \end{aligned}$$

where

- $f(x) : \mathbb{R}^n \to \mathbb{R}$ is the *objective function* to be minimized over the variable x,
- $g_i(x) \leq 0$ are called *inequality constraints*, and
- $h_i(x) = 0$ are called *equality constraints*.

By convention, the standard form defines a *minimization problem*. A *maximization problem* can be treated by *negating* the objective function.

2.2.4 Decision Problems as Formal Languages

Decision problems are one of the central objects of study in computational complexity theory. A decision problem is a special type of computational problem whose answer is *yes* or *no*, or alternately either *1* or *0*. A decision problem can be viewed as a *formal language*, where the members of the language are instances whose output is yes, and the non-members are those instances whose output is no. The objective is to decide, with the aid of an algorithm, whether a given input *string* is a member of the *formal language* under consideration. If the algorithm deciding this problem returns the answer *yes*, the algorithm is said to accept the input string, otherwise it is said to reject the input.

An example of a decision problem is the following. The input is an arbitrary *graph*. The problem consists in deciding whether the given graph is *connected*, or not. The formal language associated with this decision problem is then the set of all connected graphs—of course, to obtain a precise definition of this language, one has to decide how graphs are encoded as binary strings.

CamScanner

A decision problem has only two possible outputs, *yes* or *no* (or alternately *1* or *0*) on any input

Decision problems are closely related to function problems, which can have answers that are more complex than a simple 'yes' or 'no'. A corresponding function problem is "given two numbers x and y, what is x divided by y?". They are also related to optimization problems, which are concerned with finding the *best* answer to a particular problem.

A method for solving a decision problem given in the form of an algorithm is called a *decision procedure* for that problem. A decision procedure for the decision problem "given two numbers x and y, does x evenly divide y?" would give the steps for determining whether x evenly divides y, given x and y. One such algorithm is long division, taught to many school children. If the remainder is zero the answer produced is 'yes', otherwise it is 'no'. A decision problem which can be solved by an algorithm is called ***decidable***.

Examples

A classic example of a decidable decision problem is the set of prime numbers. It is possible to effectively decide whether a given natural number is *prime* by testing every possible nontrivial factor. Although much more efficient methods of *primality testing* are known, the existence of any effective method is enough to establish *decidability*.

Decidability

- A decision problem A is called *decidable* or *effectively solvable* if A is a *recursive set*.
- A problem is called *partially decidable*, *semi decidable*, *solvable*, or *provable* if A is a *recursively enumerable set*.

Problems that are not decidable are called ***undecidable***. The *halting problem* is an important undecidable decision problem.

Recursive Set

A subset S of the *natural numbers* is called **recursive** if there exist a *total computable function* f such that:

$f(x) = 1$ if $x \in S$ and

CamScanner

$f(x) = 0$ if $x \notin S$.

In other words, the set S is recursive if and only if the indicator function f is *computable*.

Examples:

- Every finite or *cofinite* subset of the natural numbers is computable. This includes these special cases:
  - The *empty set* is computable.
  - The entire set of natural numbers is computable.
  - Each natural number (as defined in standard set theory) is computable; that is, the set of natural numbers less than a given natural number is computable.
- The set of prime numbers is computable.
- A recursive language is a recursive subset of a formal language.

Properties of Recursive Set

- If A is a recursive set then the complement of A is a recursive set.
- If A and B are recursive sets, then $A \cap B$, $A \cup B$ and $A \times B$ are recursive sets.
- Set A is a recursive set if and only if A and the complement of A are both recursively enumerable sets.
- Set A is recursive if and only if it is either the range of a non decreasing total computable function or the empty set.

Recursively enumerable set

A set S of natural numbers is called *recursively enumerable* if there is a *partial recursive* function (synonymously, a partial computable function) whose *domain* is exactly S, meaning that the function is defined if and only if its input is a member of S.

The following are all *equivalent properties* of a set S of natural numbers:

- ***Semi-decidability***:
  - The set S is recursively enumerable. That is, S is the domain (co-range) of a partial recursive function.
  - There is a partial recursive function f such that:

$$f(x) = \begin{cases} 0 & \text{if } x \in S \\ \text{undefined/does not halt} & \text{if } x \notin S \end{cases}$$

CamScanner

- ***Enumerability***:
  - The set S is the range of a partial recursive function.
  - The set S is the range of a total recursive function or empty. If S is infinite, the function can be chosen to be *injective*.
  - The set S is the range of a *primitive recursive function* or *empty*. Even if S is infinite, repetition of values may be necessary in this case.
- ***Diophantine***:
  - There is a polynomial p with integer coefficients and variables $x, a, b, c, d, e, f, g, h, i$ ranging over the natural numbers such that:
    $$x \in S \Leftrightarrow \exists a, b, c, d, e, f, g, h, i \ (p(x, a, b, c, d, e, f, g, h, i) = 0).$$
  - There is a polynomial from the integers to the integers such that the set S contains exactly the non-negative numbers in its range.

The equivalence of *semi-decidability* and *enumerability* can be obtained by the technique of ***dovetailing***.

Examples

- Every recursive set is recursively enumerable, but it is not true that every recursively enumerable set is recursive.
- A recursively enumerable language is a recursively enumerable subset of a formal language.
- The simple sets are recursively enumerable but not recursive.
- The creative sets are recursively enumerable but not recursive.
- Any productive set is **not** recursively enumerable.

Properties recursively enumerable sets

- If A and B are recursively enumerable sets then $A \cap B$, $A \cup B$ and $A \times B$ are recursively enumerable sets.
- A set is recursively enumerable if and only if it is at level Σ_1^0 of the arithmetical hierarchy.

CamScanner

- A set T is called **co-recursively enumerable** or **co-r.e.** if its complement $\mathbb{N} \setminus T$ is recursively enumerable.
- A set A is recursive (synonym: computable) if and only if both A and the complement of A are recursively enumerable.
- A set is recursive if and only if it is either the range of an increasing total recursive function or finite.
- Some pairs of recursively enumerable sets are effectively separable and some are not.

Partial function

In mathematics, a **partial function** from X to Y (written as $f: X \nrightarrow Y$) is a function $f: X' \rightarrow Y$, where X' is a subset of X. It generalizes the concept of a function $f: X \rightarrow Y$ by not forcing f to map *every* element of X to an element of Y (only some subset X' of X). If $X' = X$, then f is called a **total function** and is equivalent to a function. Partial functions are often used when the exact domain, X', is not known (e.g. many functions in computability theory).

Specifically, we will say that for any $x \in X$, either:

$f(x) = y \in Y$ (it is defined as a single element in Y) or

$f(x)$ is undefined.

For example we can consider the square root function restricted to the integers

$$g: \mathbb{Z} \rightarrow \mathbb{Z}$$

$$g(n) = \sqrt{n}.$$

Thus $g(n)$ is only defined for n that are perfect squares (i.e. 0, 1, 4, 9, 16, ...). So, $g(25) = 5$, but $g(26)$ is undefined.

CamScanner

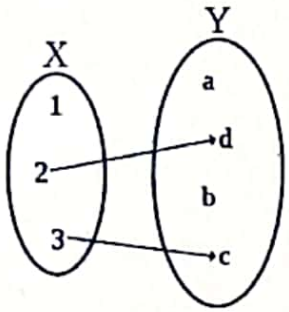

Fig . An example of partial function that is injective.

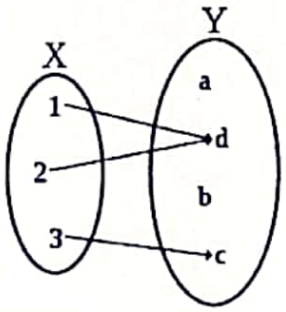

Fig. An example of total function that is not-injective.

A partial function is said to be *injective* or *subjective* when the total function given by the restriction of the partial function to its domain of definition is. A partial function may be both injective and subjective, but the term bijection generally only applies to total functions.

An injective partial function may be inverted to an injective partial function and a partial function which is both injective and subjective has an injective function as inverse.

CamScanner

Examples:

The first diagram above represents a partial function that is **not** a total function since the element 1 in the left-hand set is not associated with anything in the right-hand set.

Subtraction of natural numbers (non-negative integers) can be viewed as a partial function:

$$f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$f(x, y) = x - y.$$

It is only defined when $x \geq y$.

2.2.5 Function Problems

A *function problem* is a computational problem where a *single output* (of a total function) is expected for *every input*, but the output is more *complex* than that of a *decision problem*, that is, it isn't just *yes* or *no*. Notable examples include the *traveling salesman problem* and the *integer factorization* problem.

It is tempting to think that the notion of function problems is much richer than the notion of decision problems. However, this is not really the case, since function problems can be recast as decision problems. For example, the *multiplication* of two integers can be expressed as the set of triples *(a, b, c)* such that the relation $a \times b = c$ holds. Deciding whether a given triple is member of this set corresponds to solving the problem of multiplying two numbers. Similarly, finding the *minimum* value of a mathematical function $f(x)$ is equivalent to a search on k for the problem of determining whether a feasible point exists for $f(x) \leq k$.

2.2.6 Measuring the Size of an Instance

To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem. However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve. Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as function of the size of the instance. This is usually taken to be the size of the input in bits. Complexity theory is interested in how algorithms scale with an increase in the *input size*. For instance, in the problem of finding whether a graph is connected, how much more

CamScanner

time does it take to solve a problem for a graph with $2n$ vertices compared to the time taken for a graph with n vertices?

If the input size is n, the time taken can be expressed as a function of n. Since the time taken on different inputs of the same size can be different, the worst-case time complexity $T(n)$ is defined to be the maximum time taken over all inputs of size n. If $T(n)$ is a polynomial in n, then the algorithm is said to be a polynomial time algorithm. Cobham's thesis says that a problem can be solved with a feasible amount of resources if it admits a polynomial time algorithm.

CamScanner