4CS4-6: Theory of Computation (Computational Complexity)

Lecture 25: April, 19, 2019

Prof. K.R. Chowdhary *: Professor of CS*

25.1 Introduction

When mathematical notion of algorithm is adopted, it opens up the possibility that certain computational problems cannot be solved by any algorithm. We have discussed in the beginning chapters, that given a set of alphabet there are only countable number of strings, where as there are uncountable number of languages. The FA, and PDA are all finite objects and these can be used to specify the languages that can be described by strings. Therefore, there are only countable many recursive and recursively enumerable languages over the alphabet. However, there still exist the languages that can be partially decided or not decided by the Turing machines.

According to the Church-Turnings thesis, computational task which cannot be carried out by Turing machine are impossible, hopeless and undecidable.

A problem whose language is *recursive* is said to be decidable otherwise the problem is undecidable. In other words, a problem is undecidable if there is no algorithm that takes an input an instance of the problem and determines whether the answer to that is "Yes" or "No".

25.2 Computational Complexity measures

Due to use of computer in everyday life, the attention rose to the questions like–how to measure the effectiveness of computer programs (algorithms), how to compare the effectiveness of two algorithms, and how to measure the computational difficulty of computing problems. The theory of complexity is concerned with measuring the difficulty of computations. To do this, we must discuss what is meant by a complexity measure.

We are concerned with the computational complexity measures which are defined for all possible computations, i.e., for all partial recursive functions mapping the integers into the integers. Therefore, to define a complexity measure, we require an effective way of specifying all possible computations or algorithms (for the computati on of these functions). The complexity measure will then show how many "steps" it takes to evaluate any one of these algorithms on any specific argument. For example, our list of algorithms or computing devices could be a standard enumeration of all one-tape Turing machines (which we know

are capable of computing all partial recursive functions), and the complexity measure of a given machine M_i (or algorithm) working on argument n could be the number of operations performed by M_i before halting on input n.

It should be noted that these complexity measures are associated with the algorithms and not directly with the functions they compute. The reason for this is that in computations we usually deal with algorithms which specify functions, and for each computable function there are infintely many algorithms which compute it. Furthermore, as we will discuss later, there exists functions which have no "best" algorithm, and thus we cannot talk of complexity of function as that of its best algorithm [hartmanhopcp].

Time and Space Complexity Dealing with these questions two fundamental complexity measures, *time* and *space*, were introduced. Both of them have been considered as functions of input. Informally, the *time complexity* of an algorithm working on an input is the number of "elementary" operations executed by the algorithm processing the given input. In other words, it is the amount of work done to arrive from input to the corresponding output. The *space complexity* of an algorithm is the number of "elementary" cells of the memory used in the computing process. Obviously, what does "elementary" means, depends on the formal model of algorithms one chooses (*machine models, axiomatic models, programming languages*, etc). Since a *theory* always tries to establish results (*assertions*) which are independent of the formalism used and have general validity, this dependence on the measurement on the model does not seem a welcome! Fortunately, the reasonable computing models used are equivalent, in the sense that the differences in the complexity measurement are negligible for the main concepts and statements of the complexity theory.

Here, we use Turing machine (TM) as the standard computing model to define the complexity measures. From the several versions of TM we consider the off-line multitape Turing machine (MTM) consisting one two-way read-only input tape, and finite number of working tapes each having its read-write head. A computing step of a MTM is considered to be an elementary operation of this model. In one step, a MTM M reads one symbol from each of these tapes and depending on them and current state of the M, M possibly changes its state, rewrites the symbols read from the working tapes and moves the heads at most one position to left or right. A *configuration* is set of global state of the machine, current contents of the tapes and the position of heads on tapes. A *computation* of M is sequence of configurations C_1, C_2, ..., C_m such that $C_i \to C_{i+1}$ (C_{i+1} is reached from C_i in one step).

Let M be a MTM recognizing a language $L(M)$ and let $w \in \Sigma^*$, where Σ is the input alphabet of M. If $C = C_1, C_2, ..., C_k$ is finite computation of M on input w, then the *time complexity* of the computation M (say T_M) on w is,

$$T_M(w) = k - 1.$$

If the computation of M on w is infinite, then

$$T_M(w) = \infty.$$

The *time complexity* of M is the partial function from $\mathbb{N}$ to $\mathbb{N}$,

$$T_M(n) = max\{T_M(w) \mid w \in \Sigma^n \cap L(M)\}. \tag{25.1}$$

The *space complexity* of one configuration C of M, $S_M(C)$, is the length of the longest word over the working alphabet stored on the working tapes in C. The space complexity of a computation $D = C_1, C_2, ..., C_m$ is,

$$S_M(D) = max\{S_M(C_i) \mid i = 1, ..., m\}. \tag{25.2}$$

The space complexity of M on a word $w \in L(M)$ is $S_M(w) = S_M(D_w)$, where D_w is the computation of M on w.

The space complexity of M is the partial function from $\mathbb{N}$ to $\mathbb{N}$,

$$S_M(n) = max\{S_M(w) \mid w \in \Sigma^n \cap L(M)\}. \tag{25.3}$$

25.3 Time and Space bounded Turing Machines

Let M be a Turing machine that halts on all input $x \in \Sigma^*$. Let $t(n)$ (t is $t : \mathbb{N} \to \mathbb{N}$) be the *maximum* number of steps taken by M on different inputs of length n.

Definition 25.1 Time bounded. *The Turing machine M is said to be $t(n)$ time bounded. The running time of machine is $O(t(n))$.*

Definition 25.2 time constructible. *A function $t : \mathbb{N} \to \mathbb{N}$ is time constructible if $t(n) \geq n$ and there is a Turing machine M_t that computes the function $n \mapsto t(n)$ in time $t(n)$.*

We shall only consider time-constructible functions as *time-bound*. Note that n, $\log n$, 2^n etc, are all *time-constructible*.

Example 25.3 *Time-bound of Regular language.*

Let L_1 be a regular language.The state transitions of a Turing machine mimics the state transitions of the corresponding DFA:

1. an input symbol is read;
2. the same symbol is written on the tape;
3. the state transition follows the transition of the DFA and the head moves towards right.
4. At the end of input, the state may be either *Accept* or *Reject* indicating acceptance or rejection of input.

It is clear that the running time of such a machine is bounded by $O(n)$, so the language L is also decided in time $O(n)$.

25.4 Time Complexity of arithmetic operations

Let us consider computations steps for some standard computations.

Adding two n-digit numbers: It takes usually $n+1$ steps. But if we look at minor steps then $5n+1$ steps (n additions of digits, n additions of carry, n comparisons if sum of two digits and carry is greater than or equal to 10, n steps to print lower digit, n steps to save carry). The last step is for carry save from last sum.

Even further smaller steps are taken, it comes out to be $an+b$, where a, b are constants, not dependent on n. Thus time complexity of add, is $\theta(n)$.

Multiplication: To multiply x and y, one approach is add x to 0, y times. If both numbers are n digit long, then complexity of computation is $\theta(n.10^n)$.

Other method is $\theta(n^2)$ complexity, as for adding single n-digit number it is $\theta(n)$, and doing this n times, is $\theta(n^2)$. The best known algorithm for multiplication is $\theta(n^{1.1})$.

Factoring: Let us try to find out factoring of an n-digit number. Some times the factoring is not well defined, for example, $1001 = 77 \times 13$ or it is 91×11. In this, since factors re different, the complexity of each computation is different. To factor Z we need to divide it by a range 2 to $Z-1$. If $|Z| = n$, complexity is $O(10^n)$. No solution like, $\theta(n)$ or $\theta(n^c)$, where c is constant, is available for factoring.

25.5 Complexity Terminology

Following are some standard terms of complexities.

Definition 25.4 *$T(n)$.(Deterministic TM) It is* Time complexity *of standard Turing Machine. The function $T(n)$ is called* time-constructible *if there exists a time-bound Deterministic TM that with input $|w| = n$.*

Definition 25.5 Time-constructible function *is a function f from natural numbers to natural numbers ($\mathbb{N} \to \mathbb{N}$), with the property that $f(n)$ can be constructed from $|w| = n$ (also called 1^n) by a Turing machine in time of the order of $f(n)$.*

Definition 25.6 *$T(n)$.(Nondeterministic TM) Time complexity of NDTM is also defined in the same way as for deterministic Turing Machine.*

Definition 25.7 *$S(n)$. It is* Space complexity *of standard Turing Machine. The function $S(n)$ is called* space-constructible*, if there exists a space-bound standard TM, that for each input of length n, requires exactly $S(n)$ space.*

Definition 25.8 Space Constructible. *A function f is space constructible if there exists a positive integer number and a TM M which, given a string 1^n halts after exactly $f(n)$ cells for all $n \geq n_0$, where n_0 is a positive integer.*

Examples of commonly used functions $f(n)$, such as n, n^k, 2^n are time and space constructible, as long as $f(n)$ is at least cn for a constant $c > 0$.

Definition 25.9 *$DTIME(T(n))$. It is class of all the languages that have deterministic time complexity of $O(T(n))$.*

Definition 25.10 *$NTIME(T(n))$. It is class of languages that have nondeterministic time complexity of $O(T(n))$.*

25.6 The class P

Let a language $L \subseteq \Sigma^*$, and L is *polynomial* if membership $w \in L$ can be determined in polynomial function of n, where $|w| = n$. A Polynomial time is defined in terms of number of transitions in TM M, where $L = L(M)$. The L is decidable in polynomial time if standard standard TM M can decide L in time $tc_M \in O(n^r)$, where r is natural number, not related to n. Then the family of L is called **Class-P**.

A language accepted by multi-tape TM in time $O(n^r)$ is accepted by standard TM in time complexity $O(n^{2r})$, which is also polynomial. This invariant property shows the robustness of TM.

Definition 25.11 **P** : *Class of membership problems for the languages in*

$$\mathbf{P} = \bigcup_{P(n)} DTIME(P(n)), \tag{25.4}$$

where $P(n)$ is polynomial in n.

Following are the examples of problems in class **P**:

1. *Acceptance of palindromes.* The output of this problem is Yes if $w \in \Sigma^*$ is a palindrome, else No. The Complexity class of this problem is **P**, as complexity is $O(n^2)$ for standard Turing machine.

2. *Path problem in directed graphs.* Input is $G = (V, E)$, and output is Yes if for $v_i, v_j \in V$, there is a path from v_i to v_j in the graph, else No. The complexity class is **P**, as time complexity of this problem is $O(n^2)$ due to Dijkstra's shortest path algorithm.

3. *Deriveability in CNF.* Input is Chomsky Normal Form grammar G, and sentence w, output Yes, if $S \Rightarrow^* w$ else No. Complexity for this derivation is $O(n^3)$.

We can also define $DTIME$ for function computation in the following.

Definition 25.12 *$DTIME(f(n))$ is class of languages that have deterministic time complexity of $f(n)$.*

Definition 25.13 *$EXPTIME$ is class of membership problems which can be computed by deterministic TM in exponential time, expressed as,*

$$EXPTIME = \bigcup_{P(n)} DTIME(2^{P(n)}). \tag{25.5}$$