Complexity Theory

Disclaimer

This follows the lessons from Mr. Berrachedi given in course, major additions, changes or rearrangements can be applied on the course for the sake of clarity, better explainations or just subjective opinions, written by HADIOUCHE Azouaou.

> The course will be heavily changed due to the fact that the needs the computational theory. To seperate the contents of the course to actual additions or out of context information, I will use the black band that is next to the one englobing this comment for out of course information.

USE THIS AT YOUR OWN RISK.

Chapter 1

Introduction

Complexity theory revolves around the idea of measuring execution time of an algorithm giving a measure of the execution time given some taken parameters about the inputs.

To introduce the concept of computation complexity, we start by considering the calculation of a determinant of a square matrix, we can calculate it with two different ways, first by the definition of a determinant, the second with LU decomposition.

- We start by determining how many steps needed for calculating the determinant of an $n \times n$ matrix by definition. Let $C_1(n)$ be the number of steps for evaluating the determinant of an $n \times n$ matrix $A = \left(A_i^j\right)$. By definition we have that

$$\det A = \sum_{\sigma \in S_n} \operatorname{sgn} \sigma \prod_{i=1}^{n} A_i^{\sigma(i)}.$$

  We have $\# S_n = n!$, if we suppose that $\operatorname{sgn}\sigma, \sigma(i)$ can be calculated in a constant time, then the remaining product has n steps to evaluate thus we have that $C_1(n) = n \cdot n!$

- Now for the calculation of determinant using the LU method, denote $C_2(n)$ the number of steps needed for this calculation, if we denote $T(n)$ be the amount of steps needed to do the LU decomposition, its easy to calculate that $T(n) = ((n-2)(n-1)(2n-3))/3$ and then we get that $A = LU$, with L, U triangular, and $\det L = 1$, then $\det(A) = \det(LU) = \det L \cdot \det U = \det U = \prod_{i=1}^{n} u_{ii}$ thus $C_2(n) = T(n) + n$.

Now that we calculated the amount of steps needed for each one, we will assume that each step takes a second and that we want to calculate the determinant of 100×100 matrix. Using the first algorithm, we get that it will take $C_1(100) = 100 \cdot 100!$ seconds which is approximately $3 \cdot 10^{150}$ years, for comparision, we have that the life span of the universe is approximately 10^{10} years, if we use the second method, it will take $C_2(100) = T(100) + 100$ which takes approximately a week and a day to calculate.

For our interest, we usually do not check exactly how the algorithm behaves at each point, but how it behaves asymptotically, thus we created our comparision notations.

Definition 1.1 (Comparision Notations): *Let $f, g : \mathbb{N} \to \mathbb{R}^+$, we define the following notations:*
- $f = o(g) \Leftrightarrow \forall \varepsilon > 0, \exists N \in \mathbb{N}, \forall n \geq N, f(n) \leq \varepsilon \cdot g(n)$.
- $f = O(g) \Leftrightarrow \exists c > 0, \exists N \in \mathbb{N}, \forall n \geq N, f(n) \leq c \cdot g(n)$.
- $f = \Theta(g) \Leftrightarrow f = O(g)$ *and* $g = O(f)$.

Example:
- if we consider our previous problem then $C_1(n) = O(n!)$ and $C_2(n) = O(n^3)$ which makes it really easy for us to compare the asymptotic behavior.
- if $f(n) = 100n, g(n) = n^2$ then $f = O(g)$ and $f = o(g)$.
- if $f(n) = n^2, g(n) = 100n^2 + 2n + 1$ then $f = \Theta(g)$.

Proposition 1.2: *Let $f, g : \mathbb{N} \to \mathbb{R}^+$, suppose that, then*
- *if* $\exists N \in \mathbb{N}, \forall n > N, g(n) \neq 0$ *then* $f = o(g) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.
- $f = O(g) \Leftrightarrow \exists M > 0, \exists N \in \mathbb{N}, \forall n > N, \frac{f(n)}{g(n)} < M$.

notice that n usually represents some variable of the quantity of data given by the algorithm. Depending on the context, we may take it to be the amount of elements in a list, the number of digits in a number and multiple other things.

Example: Consider the following linear programming problem with $A \in \mathbb{D}^{n \times m}$ and $b \in \mathbb{D}^m$, here $\mathbb{D}$ is the set of numbers that can be represented in the decimal base with finite digits.

$$(I) \quad \begin{cases} \max\limits_{c \in \mathbb{R}^n} c^t x \\ Ax = b \end{cases}$$

we can calculate the amount of bits needed to some $\alpha \in \mathbb{D}$ with this formula $\lceil \log_2(|\alpha|+1) \rceil$ and thus we get that the amount of bits needed to represent the problem (I) is $T = \lceil \log_2(|\alpha_1|+1) \rceil nm + \lceil \log_2(|\alpha_2|+1) \rceil m$ where the first term is to calculate how much the matrix A needs in bits and the second is for the vector b.

1.1. Formalism Of Computation

It can be noticed that in the prior definitions we did not formalize properly what is a "step" or an "algorithm", this is the goal of this section. For computations, we have a whole theory revolving around abstract computers called Computability Theory. For brievity, we will describe Turing machines directly and the concept of universality of computing machines.

We assume that the machines will just work with data written in $\{0,1\}$ alphabets, denote $\{0,1\}^* = \cup_{n \geq 0} \{0,1\}^n$ be the set of all possible binary strings, and we assume that any mathematical object x will be represented in these machines using some convention or cannonical representation denoted $\overline{x}$ or directly x is the context is clear.

Definition 1.1.3 (Turing Machine): *Let $M = (\Gamma, Q, \delta)$, M is said to be a TM (Turing Machine) if it satisfies*

1. $\Gamma = \{0, 1, \triangleright, \square\}$ *the set of alphabet of M used on the work tape.*
   - $\triangleright$ *is the start pointer.*
   - $\square$ *is the blank symbol.*
   - $0, 1$ *are symbols for intermediate calculations.*
2. $Q = \{q_{\text{start}}, q_{\text{halt}}, ...\}$ *is the set of states that M can be in.*
   - q_{start} *is the state that the machine starts with.*
   - q_{halt} *is the state that if the machine is in, ends the workflow.*
3. $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}^n$ *a transition function.*
   - L *is moving the tape pointer left.*
   - R *is moving the tape pointer right.*
   - S *is not moving the pointer.*

A turing machine can be thought of as a person given a problem P that is subdivided to subproblems $P_1, ..., P_n$, and we give him infinite paper (scratchpad) to do the intermediate solutions, Γ represents the possible symbols he can use on the scratchpad, for example the "English" language with basic mathematical notations, the states represent what subproblem P_i he is solving, and δ is the work he is doing, from writing the intermediate calculations and not changing the paper, to changing papers to do different subproblems.

Definition 1.1.4 (Running Time): *Let $f : \{0,1\}^* \to \{0,1\}^*$ and $T : \mathbb{N} \to \mathbb{N}$ be two maps and M a Turing machine. We say that M computes f in $T(n)$-time if for every $x \in \{0,1\}^*$, if we set the input of M as x then after at most $T(|x|)$ transition, $f(x)$ is written on the output and M halts.*

thus this measure is independent of the time that each transition goes, and using the definitions we have earlier, we can give upperbounds for the growth of an algorithm depending on its input's size. The most important result is the following theorem which allows us to simulate turing machines inside of others in an efficent manner.

Theorem 1.1.5 (Efficent Universal Turing Machine): *There exists a Turing machine $\mathcal{M}$ such that for every $x, \alpha \in \{0,1\}^*$, $\mathcal{M}(x, \alpha) = M_\alpha(x)$ where M_α is the Turing machine represented with α.*

α here represents a "program", and thus we can program any Turing machine behavior inside of the universal Turing machine $\mathcal{M}$.