AN INTRODUCTION TO COMPLEXITY CLASSES

KSHEMAAHNA NAGI

1. A Bit of Background

We refer to functions as functions whose input and output is a string of bits. Recall that a string of bits is a finite sequence of 0's and 1's. The set of all strings of bits of length n is denoted by $\{0,1\}^n$. The set of all strings of bits (the set of all strings of bits of all possible lengths) is denoted by $\{0,1\}^*$ which is the union of $\{0,1\}^n$ for all n≥0. Or,

$$\{0,1\}^* = \cup_{n\geq 0} \{0,1\}^n$$

. Simple encodings can be used to represent objects such as integers, letters, graphs and matrices as string of bits. Henceforth, we refer to objects as strings of bits.

Definition 1.1 (Boolean function)**.** A Boolean function f is a function which maps strings to strings, returning a single bit as an output, identified as the set $L_f = \{x : f(x) = 1\}$.

The sets described in the above definition are referred to as languages or decision problems. Thus, the problem of computing f can be stated as identifying x, such that $x \in L_f$.

2. Computational Models

(Church-Turing Thesis) The Church-Turing thesis states that any real-world computation can be translated into an equivalent computation involving a Turing machine.

The original formulation, known simply as Church's Thesis is as follows:

(Church Thesis) Real-world calculation can be done using the lambda calculus, which is equivalent to using general recursive functions.

These statements are important as they point to the fact that all computational models, be it the Turing Machine, lambda calculus, programming languages like JAVA and Python or even Conway's Game of Life, are equivalent and universal i.e. any algorithm that can be run on one model can be simulated on any other.

3. Turing Machines

We begin our discussion on k-tape Turing Machines. A tape is an infinite one-directional line of cells, each cell capable of holding at most one symbol. A symbol already present in a filled cell can be read by a read type tape head and a symbol can be written in an empty cell by a write type tape head. Note that a read/write type tape head might be present. The tape head can only move one cell left or right in a single step, dividing the computation of the Turing Machine into discrete time steps.

A Turing Machine can have k such tapes each with their own tape head. The first tape is designated as the input tape is equipped with a read-only tape head (this just means the input tape is prepared beforehand). The other k-1 tapes are called work tapes and the last

Date: August, 2024.

work tape is called the output tape. The work tapes are equipped with a read/write tape head.

The Turing Machine has a state at every discrete time step. Each state is the set of symbols of the cells being read/written at that step in all k tapes. The i^{th} state is denoted by q_i and the collection of all states of the machine is denoted by Q. The machine also contains a "register" that stores the state of the machine at each discrete time step.

The symbols allowed to be ascribed to a cell come from a finite set called the alphabet, denoted by Γ. The alphabet is a set consisting of a designated start symbol, a designated blank symbol, 0 and 1.

The state of the machine at the i^{th} discrete time step determines the action of the machine (or the collective actions taken by the tape heads) at the next $(i+1)^{th}$ time step, thereby determining the state at the next $(i+1)^{th}$ time step.

Each tape head is allowed to perform the following actions:

(1) Read the symbol in the cell directly under it

(2) Replace an already written symbol in a cell with a new one or re-write the old symbol which is equivalent to not replacing it (this is applicable to read/write tape heads).

(3) change the register to contain a new state from the set Q

(4) Move one cell to the left or right

The set of actions performed to go from the state at the i^{th} discrete time step to the state at the $(i+1)^{th}$ discrete time step is called the transition function. The transition function is a function $\delta : \mathrm{Q} \times \Gamma^k \to \mathrm{Q} \times \Gamma^{k-1} \times \{L, S, R\}$ describing the rule the Turing Machine, say M, uses in performing each step. Here L,S and R denote Left, Right and Stay respectively.

The Turing Machine halts when the state is one of the designated final states of the Turing Machine M.

Definition 3.1 (Computing A Function and Running Time)**.** Let $f : \{0,1\}^*$ ß $\{0,1\}^*$ and let T: N $\to$ N be some functions, and let M be a Turing machine. We say that M computes f in T(n)-time if for every $x \in \{0,1\}^*$,if M is initialized to the start configuration on input x, then after at most T(—x—) steps it halts with f(x) written on its output tape.We say that M computes f if it computes f in T(n) time for some function T:$\mathbb{N} \to \mathbb{N}$.

Definition 3.2. We say that a function T: $\mathbb{N} \to \mathbb{N}$ is time constructible if T(n) $\geq$ n and there is a Turing Machine M that computes the function $x \mapsto \langle T(|x|) \rangle$ in time T(n).

3.1. **Universal Turing Machine.** Alan Turing first showed that there exists a universal Turing Machine that could simulate every Turing Machine M given M's description as its input. It feels intuitive that any function should be computable given enough time. However this is provably false: there exist some functions which cannot be computed by a Turing Machine, i.e. cannot be computed in a finite number of steps. One such function is HALT (the Halting Problem). The function HALT takes as input a pair α, x and outputs 1 if and only if the the Turing Machine M_α represented by α halts on input x within a finite number of steps.

3.2. **Non-deterministic Turing Machines.** A Non-Deterministic Turing Machine is a Turing Machine where there are more than one possible states that the machine can go to from a given state.

4. Lambda Calculus

Now we discuss untyped lambda calculus, a computational model that is equivalent to a Turing Machine.

Lambda functions are a feature in programming languages such as Python, C++ and Mathematica. Here is the syntax of a lambda function in python:

```
lambda <arguments> : <expressions>
```

It is clear that the lambda keyword is defining a function without naming it. Hence, lambda functions are also called anonymous functions in some programming languages. When the <arguments> and <expressions> involve some variable, the lambda keyword allows us to abstract over the variable with a value. The syntax for this is:

```
(lambda <arguments> : <expressions>) (variable value)
```

For example, the python syntax below returns the value 121.

```
(lambda x : x*x + 2*x + 1)(10)
```

In untyped lambda calculus, the lambda functions use the mathematical operator λ to behave like the lambda keyword but with a slightly different syntax. Henceforth, we refer to untyped lambda calculus as lambda calculus. Lambda calculus is a system of using functions as the basis of programming.

To understand the notation of lambda calculus, we start with the notation for a λ-term:

$$(\lambda x. < \text{some expression} >)\text{variable value}$$

Rewriting the above python example as a λ-term:

$$(\lambda x.x^2 + 2x + 1)10$$

.This λ-term in the example above is evaluated as follows:

$$(\lambda x.x^2 + 2 \cdot x + 1)10 \rhd 10^2 + 2 \cdot 10 + 1 = 121$$

The above example shows the central principal of λ-calculus called β-reduction also known as β-abstraction denoted by the $\rhd$ operator. The $\rhd$ operator helps us abstract the value over the variable. β-abstraction can be formally represented as:

$$\lambda[M](N) \rhd M[x := N]$$

where M is an expression involving some variable say, x and N is the value that we want to abstract over x by substituting x with a given value.

It is also worth noting the difference between free and bound variables in order to make correct substitutions. A variable is said to be free if:

- It is just a simple variable

- A variable say y is free in a lambda term if it is not to the left of the .
- A variable is free in AB where both A and B are lambda terms if it is free in at least one lambda term A or B. If it is free in one and bound in the other, then it is said to be free or bound with respect to the term.

Naturally, we might want to express expressions involving more than one variable as a λ-term. We can achieve this by nesting λ-terms:

$$\lambda x_1.\lambda x_2.\lambda x_3 \ldots .\lambda x_n. < \text{some expression involving } x_1, x_2, \ldots x_n >$$

For example, λ-term for the expression $x^2 + y^2 + z^2$ can be expressed as:

$$\lambda x.\lambda y.\lambda z.x^2 + y^2 + z^2$$

. This can also be written as

$$\lambda xyz.x^2 + y^2 + z^2$$

. We must be careful to abstract over the variables in order (here x, y and then z). As we keep abstracting over a variable, we can think of the resulting lambda term as a new one with one less parameter. For example, say we abstract over x in the above example with 4. The resulting new lambda term is

$$\lambda yz.16 + y^2 + z^2$$

.

There are in general the following types of lambda terms:

- Variables like x, y, z etc.
- Abstraction terms of the form $\lambda x.M$ where x is some variable and M is a lambda term. They are called abstraction terms as we are 'wrapping up' or abstracting over an expression.
- Application terms are of the form MN where M is either a lambda term or an expression and N is also either a lambda term or an expression. They are called application terms as we are 'unwrapping' or applying one term over another.

Note: Application takes place from the left. Say A, B, C, D . . . so forth are some expressions. ABCDE... is equivalent to (((AB)C)D)E... so forth.

Since lambda calculus is a computational model, it follows from the Church-Turing Thesis that we should be able to run all the algorithms we can on a Turing Machine using Lambda calculus. In other words, lambda calculus is Turing completer. To do this, we will need things like data types and logical operators. In the beginning we talked about how lambda calculus is the basis of functional programming. This statement can be extended to say lambda functions are the theoretical base of functional programming This is because all data types and operators can be represented by lambda functions and this is what helps make lambda calculus Turing complete. The encoded data and operators in lambda calculus are referred to as Church encoded numerals. Here is a guide [4].

5. Intuitive Equivalence of Lambda Calculus and the Turing Machine

An intuitive proof for the equivalence of the two computational models described can be to simulate the Turing Machine in a programming language say Python or Haskell and convert the program into an untyped lambda calculus version using Church-encoded numerals for data and operations.

6. Algorithms Analysis

A complexity class is a set of functions that can be computed within a given resource. When looking at complexity classes, we are concerned with asymptotic analysis or how the running time of an algorithm varies with increasing input size without bound. Worst case time complexity is done using the input for which the algorithm takes the maximum running time. Best case time complexity is done using the input for which the algorithm takes the minimum running time and average case analysis is generalised to any input.

7. Some Complexity Classes

Complexity classes, in essence, is a kind of taxonomy for computable problems that helps us determine how feasible it is to solve them with limited computational resources. Individual classes can be grouped by the type of computational resource we are placing a restriction on. When running time is limited, time complexity classes are created. When storage space is limited, space complexity classes are created.

7.1. **Time Related.**

7.1.1. ***DTIME***.

Definition 7.1 (**DTIME**)**.** Let $T : \mathbb{N} \to \mathbb{N}$ be some function. By definition, **DTIME**(T(n)) is the set of all Boolean functions that are computable by a Deterministic Turing Machine in time c · T(n) for some constant c>0.

It follows from this definition that for every Boolean function f ∈ **DTIME**(T(n)), it can be said that f(n)=O(T(n)). The class **DTIME** is used to construct the definition for some other (time) complexity classes.

7.1.2. ***P***.

Definition 7.2. The class **P** is a collection of Boolean functions f computable in polynomial time by a deterministic Turing Machine. Or,

$$\mathbf{P} = \cup_{c\geq 1}\mathbf{DTIME}(n^c)$$

Some examples of problems in the complexity class P are searching, sorting, Shortest Path Problem (finding the shortest path between 2 vertices of a graph) and the Minimum Spanning Tree (MST) Problem.

A spanning tree is a subset of a graph G such that all the vertices of the graph are connected by the minimum number of edges without any cycles. When each of the edges is assigned a weight or value, the minimum spanning tree is the spanning tree with the minimum possible total sum of edge weights. The Minimum Spanning Tree (MST) problem requires us to find a MST on the given graph.

One algorithm to achieve this is the Kruskal's algorithm. The algorithm works as follows:

(1) The edges of the given graph are sorted in ascending order of weights and added to a set.
(2) A forest of all vertices in the given graph is constructed but with each vertex isolated.
(3) The edge with least weight is taken from the set and added to the forest. The 2 vertices connected by the edge are added to a separate set of visited vertices.
(4) Step 3 is repeated, ensuring there are no cycles formed.
(5) The algorithm terminates when the set of vertices in the graph is equal to the set of vertices visited.

The worst case, average case and best case complexity of this algorithm are the same, O(E log E) or O(E log V) where E and V are the number of edges and vertices of the graph. It follows that this is a polynomial time algorithm.

7.1.3. ***NTIME***.

Definition 7.3. Let $T : \mathbb{N} \to \mathbb{N}$ be some function. By definition, **NTIME**(T(n)) is the set of all Boolean functions that are computable by a Non-Deterministic Turing Machine in time c · T(n) for some constant c>0.

7.1.4. ***NP***.

Definition 7.4. The class **NP** is a collection of Boolean functions f computable in polynomial time by a Non-deterministic Turing Machine. Or,

$$\mathbf{NP} = \cup_{c\geq 1}\mathbf{NTIME}(n^c)$$

This represents the class of problems verifiable in polynomial time but not solvable in polynomial time. The most famous open problem in complexity theory is whether **P=NP**. We believe it these two are not equal. However, if they were that means that we could find polynomial time algorithms to solve problems in the class **NP**, possibly breaking cryptography.

P and **NP** rely on worst case computation. However, there are also analogues, namely **distP** and **distNP** which are defined for average case computation.

7.1.5. ***EXPTIME***.

Definition 7.5. The class **EXPTIME** is a collection of Boolean functions f computable in at most time by a deterministic Turing Machine. Or, for some constant k,

$$\mathbf{EXPTIME} = \cup_{c\geq 1}\mathbf{DTIME}(2^{n^k})$$

7.1.6. ***NEXPTIME***.

Definition 7.6. The class **NEXPTIME** is a collection of Boolean functions f computable in at most time by a Non-deterministic Turing Machine. Or, for some constant k,

$$\mathbf{NEXPTIME} = \cup_{c\geq 1}\mathbf{NTIME}(2^{n^k})$$

7.2. **Space-Related. SPACE**(f(n)) is the class of languages that can be accepted by a Deterministic Turing machine that visits never more than f(n) cells of its work tape. **NSPACE**(f(n)) is defined likewise, using non-deterministic machines.

Definition 7.7. (**PSPACE**)

$$\mathbf{PSPACE} = \cup_{c\geq 1}\mathbf{SPACE}(n^c)$$

Definition 7.8. (**NPSPACE**)

$$\mathbf{NPSPACE} = \cup_{c \geq 1} \mathbf{NSPACE}(n^c)$$

Definition 7.9. (**L**)

$$\mathbf{L} = \mathbf{SPACE}(log(n))$$

Definition 7.10. (**NL**)

$$\mathbf{NL} = \mathbf{NSPACE}(log(n))$$

Theorem 7.11. *We have:*

$$\boldsymbol{L} \subseteq \boldsymbol{NL} \subseteq \boldsymbol{P} \subseteq \boldsymbol{NP} \subseteq \boldsymbol{PSPACE} = \boldsymbol{NPSPACE} \subseteq \boldsymbol{EXPTIME} \subseteq \boldsymbol{EXPSPACE}.$$

References

[1] The lambda calculus, Aug 2023.
[2] Programming with lambda calculus, Aug 2024.
[3] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2006.
[4] WIKIPEDIA. Church encoding, Aug 2024.

[3] [4] [2] [1]