Contents

Part I

Computability Theory

Chapter 1

Lecture Notes

1.1 Computation

Abstractly, a *computation* is the "work done" by a *computation machine*. For example, we can consider our brains as computation machines and whatever they do is considered a computation. A computation machine M can be viewed as a black box that takes an input w and acts on the input in a step by a step manner. Suppose a brain is tasked with calculating a formula. This formula is the input and at each step the brain may calculate or simplify a certain part of the formula.

A *configuration* is a description of the internals and the state of the machine. Note that the state of the machine may be described in many ways, however, we usually fix a method of describing so that the resulting configurations are unique. A configuration of a brain computing a formula might be the state of neurons in the brain and the steps of the calculation as written in a piece of paper.

In each unit of time the configuration of the machine changes based on its *configuration graph*. This is like saying that in each time unit the brain does an algebraic step such as addition or multiplication and then changes the content of the paper to the new intermediate result. This graph is a directed graph with all the possible configurations of M as nodes. There is an edges from configuration C to C' if the machine go from C to C' in one unit of time. In our case, the brain can go from C to C' if the corresponding formula of C' can be arrived by doing one algebraic step from the corresponding formula of C. This graph might be infinite, and in fact, finite configuration graphs can be dealt with tools from graph theory and thus do not provide a rich theory of computation.

In this way, we can formalize the notion of computation machines as a discrete dynamical system.

Definition: A discrete dynamical system is a system whose state varies with a discrete variable, time, according to a directed graph $G = (\mathcal{V}, \mathcal{E})$ such that $\mathcal{V}$, the set of states of the system, is a discrete set. The set $\mathcal{E}$ determines how the state of the system varies with time. Let $V_t \in \mathcal{V}$ be the state of the system at time t, then the state of the system a unit of time later, V_{t+1}, must be a neighbor of V_t. That is,

$$V_{t+1} \in N(V_t) = \{V \in \mathcal{V} \,|\, (V_t, V) \in \mathcal{E}\}$$

Remark 1. In general, $N(V_t)$ might have zero, one, or more than one states. When $N(V_t)$ then the system can not progess and it stops. When V_t has exactly one neighbouring state, the next state is uniquely determined. However, when it has more than one neighbouring

states, then the next state is selected non-determinstically. Such selections are not based on some probability distribution or any other rules.

Definition: A **computation machine** is a discrete dynamical system $M = (\mathcal{C}, \mathcal{E})$. The state $C \in \mathcal{C}$ is called a **configuration** of the machine and $E \in \mathcal{E}$ is called a **transition**. On an input w, the machine will be placed in an *initial configuration* $C_0^w \in \mathcal{C}$. At each unit of time, a neighbouring configuration is chosen if it exists. If no neighbouring configurations exists, then the machine stops. A computation machine M is deterministic for any $C \in \mathcal{C}$ there is at most one neighbour.

The possible configurations that may be attained by an input w, form a tree. If the computation is deterministic, then the tree is a path.

Remark 2. In computation, we usually distinguish between halting and crashing. Loosely defined, halting is an expected stopping and crashing is an unexpected stopping. To distinguish between these two scenarios, we may consider $\mathcal{C}_h \subset \mathcal{C}$ as the set of halting configuration and add it to the description of M. Therefore, if a non-halting configuration has no neighbour, the computation machine crahses.

When viewing our current definition of the computation machines as computer programs, we see certain similarities. For example, a computer program is a set of variables and instructions. At each step, an instruction is executed which will change the value of the some the variables– e.g. the instruction pointer. The input of the program is considered as one of the variables. Then, the values of the variables is the configuration of the program. The instruction are the transition between different configurations.

However, there is a major difference between computer programs and computation machines. Computer programs are finite, however, computation machines are not. As stated above, the set of configurations of any useful machine is infinite. As a result, the set of transitions is also infinite. On the other hand, a practical theory of computation requires a certain finite limitation. This finite limitation is placed on the description of the computation machine. That is, there exists a finite set symbols such that the computation machines can be described by a finite string of those symbols.

Therefore, if we can describe a machine by a string, it makes sense to assume that its configurations and transitions correspond to some strings. As a result, a computation machine can also be viewed as a string processor. We want our string processor to have a finite description.

A string processor's configurations are strings and its transition describe how these strings change. There are infinitely many possible strings. Thus, the challange is to give the correct result/appropriate action for infinitely many input by a finitely describable entity. Hence, finite description is the same as saying that configuration graph can be constructed by finitely many rules. Finitely many rules implies that configurations change locally. Local property is necessary for finite description. Basic operations are changes in bounded number of memory.

1.1.1 Problems

A problem is a query on some given data and constants. To answer the query by a computation machine, we need to encode the data of the problem into some string. This string is then given to the computation machine. The computation machine takes into consideration the constants and query of the problem so that on any valid input it returns the answer to the query.

The coding of the problem must not have any pre-processing. We enforce this by making the coding essentially trivial. The reason being that different coding may result in better or faster computation.

1.2 Formal Languages

- Suppose we want to compute a function $f : \Sigma^* \to \Pi^*$ with yes/no answer. That is, $\forall w \in \Sigma^*,\ f(w) \in \{0, 1\}$. Then, computing $f(w)$ is equivalent to determining $w \in L_f$ where $L_f = \{w \mid f(w) = 1\}$.
- Generally a problem type is described by a triple of constants, given data, and a query. Yes/no problems have queries which are either true or false.
- To have a well-defined yes/no problem, the set L must have a finite description.

1.2.1 Finite description of formal languages

1. Logical: $L = \{x \mid P(x)\}$ for some proposition P on strings.
2. Grammar: $L = \langle A, \tau_1, \ldots, \tau_n \rangle$ where A is a set of axioms and τ_i are the rules that allow us to combine the axoim, inference rules.
3. Computation Machine: $L =$ is the language of a computation machine, a program in a compiler.
4. Enumerator: $L =$ is the language of a an enumerator, a generator in a compiler.
5. Formal Game: $L =$.?
6. Transducer?

In general, the classes of languages that can be described by several of the above's methods are interesting.

A definition is given by specifying a type, the data, and the properties.

1.3 Computation Machines

Computation is a finite description for the formal language and an evaluation process. Types of machines:

- Deterministic vs Nondetermninistic. Technically, any deterministic machine is also non-determninistic.

Machines may have some halting configurations. Based on that we can convene the certain halting configuration are accepting configuration and others are rejecting configuration. Based on that we can define certain languages for the machines, for example, the input that are accepted. Then, we have two further classifications.

- Acceptors vs Deciders.

Therefore, a computation is described by its finite description and its standard language. Deciders are closed under inversion of output.

The main problem in theory of computation is determining the relationships of the subclasses of a model. For example, whether DetDec are a proper subset of DetAcc or are they equal.

1.3.1 Finite automaton

1.3.2 Pushdown automaton

1.3.3 Turing machine

1.3.4 Enumerators

1.4 Complexity of the languages

We want to measure the complexity of the problems wrt some model. One way is to reduce the problems to each other.

many-to-one reduction. B must be serializable. hardness and completeness. isomorphism conjecture. Berman-Hartmanis conjecture

1.5 Gap problems

Gap problems between NATM and DATM: No. Gap problems between DATM and DDTM: No. Gap problems between NATM and DDTM: Yes. Gödel's theorem.

Halting problem. Complete problem.

1. Fix a coding for a TMs in $\{0, 1\}$.
2. It is like numbering them.
3. Determining if a number is TM are not is a simple syntax checking.
4. Construct a language that does not have a DTM. Done by the Cantor's diagonalization.
5. If lucky, the language has a acceptor.

1.6 Constraining Turing machines

1.6.1 Time constraints

redefine the TM as binary and single taped. define a the TIME as the length of the computation. Define $f(n)$-boundedness of a machine. Now limit by choosing f.

Write the Gödel's proof for P vs NP.

$NP \subset DDTM$.

Combination of two poly TM is another poly TM.

NP is like having a poly certifier.

1.6.2 Space constraints

redefine the TM as 3-taped: input RO, worktap RW, output WO. Argue that this does not affect our computation capability and results in the same class of languages (almost).

LBAs and the context sensitive grammers.