

# Computation

ian.mcloughlin@gmit.ie

**Abstract**—The following notes are about computation. We consider the limits and efficiency of computation. We are largely only interested in syntax.

## I. LANGUAGES

Start with any set, call it an alphabet and denote it by  $A$ . For example, take the set  $A = \{0, 1\}$  is an alphabet. We define strings of length  $i$  over  $A$  recursively:

$$A^0 = \{\epsilon\}; A^1 = A; A^i = \{as \mid a \in A, s \in A^{i-1}\}$$

The  $\epsilon$  stands for the empty string<sup>1</sup>.

The Kleene star  $A^*$  of the alphabet  $A$  is then union of all  $A^i$  for  $i \in \mathbb{N}_0$ . Here  $\mathbb{N}_0$  is the set of natural numbers including zero:  $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ .

$$A^* = \bigcup_{i \in \mathbb{N}_0} A^i$$

A language  $L$  over an alphabet  $A$  is a subset of  $A^*$ :  $L \subseteq A^*$ . Note that the strings in a language all have finite length.

## II. TURING MACHINES

Turing machines encapsulate the concept of a computer. They are simple machines, consisting of a single read/write head and an infinite tape of cells, all but a finite number of which are blank. The head is over a single cell of the tape and is in any one of a finite number of states at a given point in time. Turing machines perform one small step at a time, which involves reading the symbol in the current cell, overwriting it with a symbol, moving to the cell to the left or right, and changing state.

We define a (deterministic<sup>2</sup>) Turing machine  $M$  by a 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_f)$$

where:  $Q$  is a set of states;  $\Sigma$  is the input alphabet, not containing the blank symbol  $\sqcup$ ;  $\Gamma$  is the tape alphabet, a superset of  $\Sigma$  containing  $\sqcup$ ;  $\delta$  is a map:

$$\delta : (Q \setminus \{q_a, q_f\}) \times \Gamma \rightarrow \Gamma \times \{L, R\} \times Q;$$

$q_0$  is the initial states;  $q_a$  is the accept state, which is a terminal state;  $q_f$  is the reject state, also a terminal state.

Before the Turing machines begins its operation, a finite number of consecutive non-blank cells contain symbols and the head is over the left-most of these. These non-empty cells form the Turing machine's input and can be viewed as a string.

When the Turing machine operates on an input, there are three possibilities. The first two are similar: the machine can accept the input by ending in the accept state or the machine can reject the input by ending in the reject state. In these cases the machine stops upon entering these states, which is usually called halting. The third possibility is that the machine might never stop operating.

The subset of  $\Sigma^*$  (the Kleene star of the input alphabet) containing all elements that are accepted by the Turing machine  $M$  is called the language of the Turing machine  $L(M)$ :

$$L(M) \subseteq \Sigma^*$$

A Turing machine that halts on all inputs is called a decider and is said to decide its language. Note it's possible for two distinct Turing machines  $M_1$  and  $M_2$  to accept or decide the same language:  $L(M_1) = L(M_2)$ .

## III. STATE TABLES

A Turing machine can be summarised in five-columned table called a state table. Each row of the table describes how to perform the transition function  $\delta$  for a given state and tape symbol. Table I gives an example of a state table for a Turing machine that accepts all strings over the alphabet  $\{0, 1\}$  that contain an even number of 1's, including zero as an even number.

State	Input	Write	Move	Next
$q_0$	0	0	R	$q_0$
$q_0$	1	1	R	$q_1$
$q_0$	$\sqcup$	$\sqcup$	L	$q_a$
$q_1$	0	0	R	$q_1$
$q_1$	1	1	R	$q_0$
$q_1$	$\sqcup$	$\sqcup$	L	$q_f$

TABLE I: Turing machine state table

Provided we adopt some conventions, the state table can be used to describe the whole Turing machine. First, the initial state is the first one listed in the table. Second, the tape alphabet is given by the distinct symbols in the second column. Finally, the only difference between the input and tape alphabets is the blank symbol. Note that there is single row for each combination of state and tape symbol.

## IV. DECISION PROBLEMS

A decider makes a binary choice for a given input: accept or reject. The Turing machine  $M$  in this case performs a map:

$$f : \Sigma^* \rightarrow \{q_a, q_f\}$$

<sup>1</sup>It's notoriously difficult to depict.

<sup>2</sup>All Turing machines will be assumed to be deterministic until we get to section VII.

Sometimes this map is called an indicator function, as it indicates which elements of  $\Sigma^*$  are in  $M$ . In general, a map from a set to any set with two elements is called a decision problem. So, a decider performs a decision problem.

## V. COMPLEXITY

A decider takes a finite number of steps before halting on a given input. We are typically interested in knowing the maximum number of steps  $f(n)$  the decider takes on all inputs of length  $n$ . We use the number of steps as a proxy for time — it's common to call the number of steps the *time* the Turing machine takes. We can often express  $f(n)$  using a simple formula. For example, the Turing machine in Table I takes  $f(n) = n + 1$  steps on every input — reading every symbol of the input and the blank after it before halting.

We are interested in knowing how  $f(n)$  grows in relation to  $n$ . For this, we use big-O notation:

**Definition.** A function  $f(n)$  is said to be  $O(g(n))$  if there exist two positive integers  $c$  and  $n_0$  such that  $cg(n) \geq f(n)$  for all  $n \geq n_0$ .

In the previous example,  $f(n) = 2n + 2$  is  $O(n)$  which can be seen by setting  $g(n) = n$ ,  $c = 3$  and  $n_0 = 2$  as per Fig. 1.

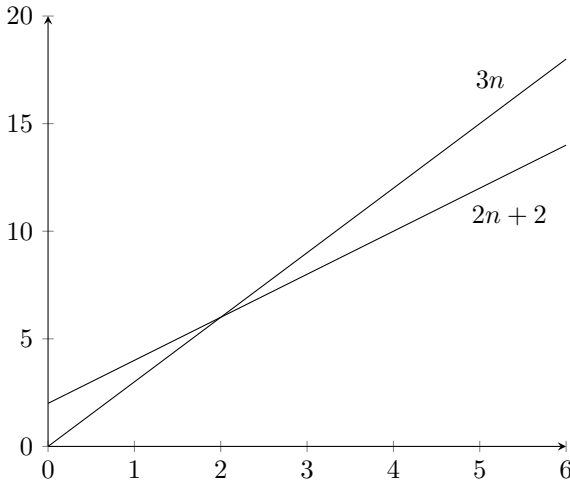


Fig. 1:  $2n + 2$  is  $O(n)$

We usually define the complexity class  $\mathbf{NTIME}(f(n))$  as the set of all languages for which there is a decider that takes  $O(f(n))$  steps.

## VI. P

We denote by  $\mathbf{P}$  the set of languages that are each decidable by some Turing machine in  $O(n^k)$  steps for some  $n \in \mathbb{N}$  where  $\mathbb{N}$  is the set of natural numbers without zero,  $\{1, 2, 3, \dots\}$ . Using the  $\mathbf{NTIME}(f(n))$  definition above,  $\mathbf{P}$  is equivalently defined as the union of  $\mathbf{NTIME}(n^k)$  for all  $k \in \mathbb{N}$ .

The  $\mathbf{P}$  is short for polynomial. Here, we define a polynomial is an expression in a variable  $n$  made up of constants (elements on  $\mathbb{N}$ ), powers of  $n$  to a constant, addition, subtraction

and multiplication. For example,  $n^5 + 2n + 4$  is a polynomial whereas  $2^n + n^2 + 5$  is not because it is made up of something other than those elements listed above, namely  $2^n$ .

## VII. NON-DETERMINISM

We consider now a modification to Turing machines: we allow the Turing machine to transition to zero or more states simultaneously from a given state when reading a given symbol. This requires a modification to the transition function as follows:

$$\delta : Q \times \Gamma \rightarrow \Gamma \times \{L, R\} \times \mathcal{P}(Q)$$

where  $\mathcal{P}(Q)$  is the powerset of  $Q$  — the set of all subsets of  $Q$ . From the state table point of view, the effect is that there may be zero, one, or more rows for each combination of state and tape symbol. In practice, we can view this as the Turing machine being able to fork, at will, into two or more branches of computation. Note that every deterministic Turing machine is basically a non-deterministic Turing machine<sup>3</sup>.

We say an input is accepted by a non-deterministic Turing machine if any of its branches end in the accept state. It may be that other branches end in the fail state or compute forever. If all branches halt for all inputs, we say the non-deterministic Turing machine decides its language.

How does non-determinism affect the computations Turing machines can perform? The bad news is that this doesn't give a Turing machine any extra power. Turing proved in his 1936 paper *On computable numbers with an application to the encheidungproblem* that some languages are not the decidable by any Turing machine. There are even languages that cannot be accepted, let alone be decided. Allowing non-determinism does not change the languages that can be allowed or decided.

However, it is not known whether computations can be done more efficiently on non-deterministic Turing machines. Examples abound of efficient non-deterministic Turing machines that accept or decide languages for which no known efficient deterministic Turing machine is known. It is unknown whether such deterministic Turing machines exist. A special case of this question is the  $\mathbf{P}$  versus  $\mathbf{NP}$  problem.

## VIII. NP

The  $\mathbf{NP}$  complexity class is the non-deterministic equivalent of the deterministic  $\mathbf{P}$  complexity class. Define  $\mathbf{NTIME}(f(n))$  as the set of languages decidable by a non-deterministic Turing machine in  $O(f(n))$ . Then,  $\mathbf{NP}$  is the set of all languages decidable by non-deterministic Turing machine in  $O(f(n))$  steps.

Given that non-deterministic Turing machines may fork into many paths, how should we count the steps they take? The common way to count this is to consider the longest path taken, irrespective of whether it accepts or rejects. Thus a non-deterministic Turing machine takes  $O(f(n))$  steps if the

<sup>3</sup>You do have to change the transition function to map to  $\{q\}$  rather than  $q$ , but that's a trivial change.

maximum number of steps in its longest path on an input of length  $n$  is  $O(f(n))$ .

Note, because every deterministic Turing machine is also a non-deterministic Turing machine, every language in **P** is in **NP**. So,  $\mathbf{P} \subseteq \mathbf{NP}$ . The **P** versus **NP** problem is to determine whether  $\mathbf{P} = \mathbf{NP}$  or  $\mathbf{P} \subset \mathbf{NP}$ .

## IX. CHURCH-TURING THESIS

The ideas surrounding Turing machines arose from Turing's investigations into computability. Alonzo Church was Turing's PhD supervisor and developed an equivalent system called lambda calculus. By equivalent, we mean that any computation that can be done on a Turing machine can be done using lambda calculus and vice versa. Lambda calculus is the basis for functional programming languages, and the difference between functional and imperative programming languages is mirrored in that between lambda calculus and Turing machines.

Turing and Church were working on ideas posed by Gödel and Hilbert which concerned the foundations of mathematics. Hilbert asked if mathematics was algorithmic in the sense that true mathematical statements could be deduced by some procedure from axioms. Gödel had shown that there must be some statements in mathematics that are true but cannot be proved. Church and Turing proved, while developing their computational systems, that their systems could not compute everything. What has become known as the Church-Turing thesis is the idea that what we as humans think of as computation is perfectly encapsulated by Turing machines and so the limits on computation are real — they are not just features of Turing machines.

## X. REDUCTIONS

Ultimately the output of a Turing machine is a binary accept/reject decision. However, Turing machines also have another type of output: what is left on the tape once the machine halts. We might care about this, for instance, if we are interested in running two Turing machines sequentially on the one input. The first Turing machine might manipulate the input in order that the second Turing machine receives that input altered in some way.

As an example, consider the language **EVENPARITY** containing all strings over the alphabet  $A = \{0, 1\}$  that contain an even number of 1's, including zero as an even number. Thus,  $\mathbf{EVENPARITY} = \{\epsilon, 0, 00, 11, 110, 101, 011, \dots\}$ . A simple Turing machine decides this language, as per Table I.

We can use this Turing machine to decide the language **ODDPARITY**, containing all strings over  $A$  that contain an odd number of 1's, provided we use the machine given in Table II on the input first.

State	Input	Write	Move	Next
$q_0$	0	0	R	$q_0$
$q_0$	1	1	R	$q_0$
$q_0$	$\square$	1	L	$q_1$
$q_1$	0	0	L	$q_1$
$q_1$	1	1	L	$q_1$
$q_1$	$\square$	$\square$	R	$q_a$

TABLE II: Append a 1

In state  $q_0$ , this machine scans from left to right across the input and puts a 1 in the blank cell at the end. Then it moves to state  $q_1$  where it moves back from right to left, until it overshoots the original starting point by one cell and then moves right one cell to that exact point. Note the machine always accepts, no matter the input. The net effect is that the head is back in the same starting point on the tape and a 1 has been appended to the original input on the tape. Giving this new input to the Turing machine in Table I will lead to an accept if the original input contained an odd number of ones and reject otherwise. We call this a reduction from **ODDPARITY** to **EVENPARITY**, denoted as  $\mathbf{ODDPARITY} \leq \mathbf{EVENPARITY}$ .

How many steps  $f(n)$  does the Turing machine in Table II take on an input of length  $n$ ? It scans across the input, then reads the blank at the end and moves left, back to the last symbol of the original input. That takes  $n + 1$  steps. It then scans to the start of the input and reads the blank just beyond that before accepting, taking another  $n + 1$  steps. It total, that's  $f(n) = 2n + 2$  steps — a polynomial.

What we have shown is that the **ODDPARITY** decision problem can be converted into **EVENPARITY** decision in polynomial time. By converted into, we mean that anything in **ODDPARITY** is converted into something in **EVENPARITY** and anything not in **ODDPARITY** is converted into something not in **EVENPARITY** so that the decider for **EVENPARITY** can decide **ODDPARITY**. The fact that the conversion happens in polynomial time is significant, as the sum of two polynomials is a polynomial. Thus if **EVENPARITY** is in **P** then so is **ODDPARITY**. This essentially means that **ODDPARITY** is no more difficult to decide than **EVENPARITY**.

Note a quirk in this situation: **EVENPARITY** can be reduced to **ODDPARITY** using the same set up. The two problems are considered equivalent. This is not always the case — sometimes we can only reduce in one direction.

## XI. COMPLETE

In 1971, Stephen Cook proved [1] an interesting result: there is a language to which all of the languages in **NP** can be reduced in polynomial time. Since then, a number of languages have been shown to have this property. They are called **NP-HARD** because they are at least as hard as each of the languages in **NP**. The language that Cook used is itself in **NP**. Languages that are both **NP-HARD** and in **NP** themselves are called **NP-COMPLETE**.

The specific language that Cook demonstrated was **NP-COMPLETE** is called **SAT**, which is short for *Boolean*

*satisfiability problem*. The language is made up of well-formed Boolean formulas. These are made up of a finite number of Boolean variables that can take either of the values `True` or `False`, the two binary operators `AND` and `OR`, and the unary operator `NOT`. The `AND` operator is denoted  $\wedge$  where, for two Boolean variables  $A$  and  $B$ ,  $A \wedge B$  is `True` if and only if both  $A$  and  $B$  are. The `OR` operator is denoted  $\vee$  where  $A \vee B$  is `True` if any of  $A$  and  $B$  are, including if both are. Finally, `NOT` is denoted  $\neg$  and  $\neg A$  is `True` when  $A$  is false and vice versa. We can define well-formed formulas by saying that variables and the above three operators as described are well-formed formulas, and then allowing the substitution of well-formed formulas for the  $A$ 's and  $B$ 's in the above using brackets as necessary to show precedence. A formula is said to be satisfiable if there is some setting of the variables within the formula that makes it `True`. The language `SAT` is then the set of all satisfiable well-formed formulas<sup>4</sup>.

#### REFERENCES

- [1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: ACM, 1971, pp. 151–158. [Online]. Available: <http://doi.acm.org/10.1145/800157.805047>

<sup>4</sup>There is a slight technicality here in that the number of variables in the language is possibly infinite. Alphabets must be finite, so the variables themselves cannot be in the alphabet. We can get around this by encoding the variables using a finite alphabet.