National and Kapodistrian University of Athens

$\mu \prod \lambda \forall$

Graduate Program in Logic, Algorithms and Computation

M.Sc Thesis

Computability and Complexity of Two-Way Finite Automata

Aristotelis Misios

Supervisors

Christos A. Kapoutsis

Panos Rondogiannis

Thesis Committee

Panos Rondogiannis

Stathis Zachos

Nikolaos S. Papaspyrou

March 2013

Ευχαριστίες

Έχω δει σε πολλές εργασίες να αναφέρουν στην αρχή κάποιο απόφθεγμα, κάποιας σημαντικής προσωπικότητας.

"*Contrariwise, if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic.*" -Tweedledee

Επειδή μερικοί μπορεί να μην αναγνωρίζουν τον Tweedledee ως αυθεντία, παραθέτω ένα εναλλακτικό απόφθεγμα.

"*Listen, once you figure out what a joke everything is, being the comedian 's the only thing that makes sense.*" -The Comedian

Θα ήθελα να ευχαριστήσω πρώτα τον Χρήστο Καπούτση, για την επίβλεψη, τον χρόνο του και τον κόπο του, και τον Πάνο Ροντογιάννη για το εξαιρετικό μάθημα στην σημασιολογία γλωσσών προγραμματισμού και την ατελείωτη υπομονή του. Επίσης θέλω να ευχαριστήσω και τους κυρίους Ευστάθιο Ζάχο και Νικόλαο Παπασπύρου για την συμμετοχή τους στην τριμελή επιτροπή και όλους τους καθηγητές του ΜΠΛΑ, ειδικά, τον Κωνταντίνο Δημητρακόπουλο που παρά τα γραφειοκρατικά εμπόδια, κατάφερε να κρατήσει το μεταπτυχιακό λειτουργικό. Επίσης ευχαριστώ τους καθηγητές που με ενέπνευσαν να ασχοληθώ περαιτέρω με τα μαθηματικά, και ειδικά τον Απόστολο Θωμά. Νοιώθω την υποχρέωση να αναφέρω τις υπαλλήλους της γραμματείας του ΜΠΛΑ, για την εξαιρετική εξυπηρέτη (στα όρια της ειδικής μεταχείρησης) και την προθυμία τους κάθε φορά που χρειαζόμουν την βοήθεια τους, (σε χρονολογική σειρά) Άννα Βασιλάκη, Χρυσαφίνα Χόνδρου και Ελένη Κλη. Χαιρετίσματα σε όλους τους, ξενιτεμένους και μη, αποφοίτους του ΜΠΛΑ, τους ευχαριστω για την παρέα τους. Ειδική μνεία στον Αλέξανδρο Παλιουδάκη διότι δίχως την δική του συμβολή, δεν θα ειχε πραγματοποιηθεί αυτή η διπλωματική εργασία. Κλεινοντας προσθέτω πως ο Θάνος Τσουάνας δεν θα βρει ούτε μια λέξη για τυπικές γραμματικές σε αυτή την εργασία, για να μάθει κάποια έστω λίγα βασικά. Δυστυχώς θα χρειαστει να εγγραφεί σε κάποιο πλήρες μάθημα περί των τυπικών γραμματικών, κάποια στιγμή στην ακαδημαική του καριέρα.

Η εργασία αυτή είναι αφιερωμένη στις γιαγιάδες μου, Θέτις, Εριφύλη και Μελπομένη.

Σε αυτό το σημείο οφείλω να αναφέρω πως όλα τα λάθη, λογικά, τυπογραφικά, μαθηματικά, συντακτικά και λοιπών ειδών, βαραίνουν αποκλειστικά και μόνον εμένα. Α.Μ.

Contents

Chapter 1

Computability

1.1 Preliminaries

Definition 1.1 (Alphabets, words, and Languages)**.** An *alphabet* Σ is a finite, nonempty set of symbols. A *word* or a *string* is an n-tuple of symbols of Σ. Instead of writing $(a_1, a_2, \ldots, a_n)$ we simply write $a_1a_2\cdots a_n$. If $u = a_1a_2\cdots a_n$, then we say that n is the length of u, $|u| = n$. There is a unique word of length 0 and that is the empty string ϵ. We denote by Σ^n the set of words over Σ, that are of length n. $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$, where $*$ is kleene star, is the set of all words. If $L \subseteq \Sigma^*$ we call it a language over Σ. Therefore $\mathcal{P}(\Sigma^*)$ is the set of all languages over Σ. Finally there is a binary operation over Σ^* called concatenation. Given $u_1 = a_1a_2\cdots a_n$ and $u_2 = b_1b_2\cdots b_m$ we define $u_1 \cdot u_2 = a_1a_2\cdots a_nb_1b_2\cdots b_m$. From now on we will write u_1u_2 meant as the concatenation of u_1 and u_2. This operation can be expanded to be used on sets of strings. Let A and B be sets of strings. We define $A \cdot B = AB = \{w|\exists x \in A, y \in B : w = xy\}$.

Languages represent problems. Every language L represents the question of deciding whether a given string w is a member of a L. Does $w \in L$? Can this question be answered algorithmically? If so, then there is a machine for that language, a machine that solves that problem. A machine that, given string w, can answer yes or no.

There are many types of machines. The most complex machine is the Turing machine, which, according to the Church-Turning thesis, can compute anything that is computable, but our interest will be focused in much simpler models of computation. Models that can solve only some computable problems. One such model is the one-way deterministic finite automaton (1DFA).

1.2 One-way finite automata

1.2.1 One-way deterministic finite automaton

Definition 1.2. A *one-way deterministic finite automaton* (1DFA) is a five-tuple,

$$D = (Q, \Sigma, q_1, F, \delta),$$

where

- Q is a finite set of states $\{q_1, q_2, \ldots, q_k\}$
- Σ is a finite input alphabet
- $q_1 \in Q$ is the start state

- $F \subseteq Q$ is the set of final states
- $\delta : Q \times \Sigma \to Q$ is the transition function.

The input is written on a tape of cells. In each cell there is one symbol written. The machine has a head reading one cell, starting from the leftmost cell and moving one cell at a time. The automaton switches between states, depending on the current state and the current symbol being read, according to the transition function. The machine halts when the head reaches the end of the input, or the automaton reaches a state from which it cannot move to any other. Obviously the machine can make at most $|w|$ moves before halting. We say that D accepts w iff when the head of D reads the last symbol of the input, it moves to a final state. The set of all strings accepted by the automaton D is called the language of D.

We can also extend δ to $Q \times \Sigma^*$. $\delta' : Q \times \Sigma^* \to Q$ and define it the following way:

$$\delta'(q, a) = \delta(q, a)$$
$$\delta'(q, aw) = \delta'(\delta(q, a), w),$$

where $a \in \Sigma, w \in \Sigma^*$ and $q \in Q$. So now we can say that D accepts w if $\delta'(q_1, w) \in F$.

Example 1. Let $\Sigma = \{0, 1\}$ be the binary alphabet. For reasons of convenience we will be using this alphabet for most examples. In case we need to use another one, it will be stated clearly. We will define a 1DFA for the language

$$L = \{w | w \in \Sigma^* \text{ such that the last two symbols of the string are 01 in that order }\}.$$

$D = (Q, \Sigma, q_{11}, \{q_{01}\}, \delta)$, where $Q = \{q_{00}, q_{11}, q_{10}, q_{01}\}$ and δ transitions are according to the table below.

δ	0	1
q_{00}	q_{00}	q_{01}
q_{01}	q_{10}	q_{11}
q_{10}	q_{00}	q_{01}
q_{11}	q_{10}	q_{11}

Another way to describe it is the following: This 1DFA we defined recognises language L by reading every string and keeping in its memory (according to the state it is in) the last 2 symbols. Those 2 last symbols are the indices of the states.

1.2.2 One-way nondeterministic finite automaton

Another model of computation, similar to 1DFA, is the *one-way non-deterministic finite automaton* (1NFA). The concept of nondeterminism is essential to understand the difference between 1DFAs and 1NFAs. In a 1DFA the computation can be described as a path in the state diagram of the machine, since for every state and input there is only one state the machine can move to. This is a result of δ being a function, so for every given $(q, a) \in Q \times \Sigma$ there is a unique state $\delta(q, a) \in Q$. Nondeterminism works differently in that aspect, since there may be more than one path, and at least one of those paths need to lead to a final state for the whole computation to accept. Intuitively, the machine has the ability to "guess" the correct path, or rather, run all the paths at once.

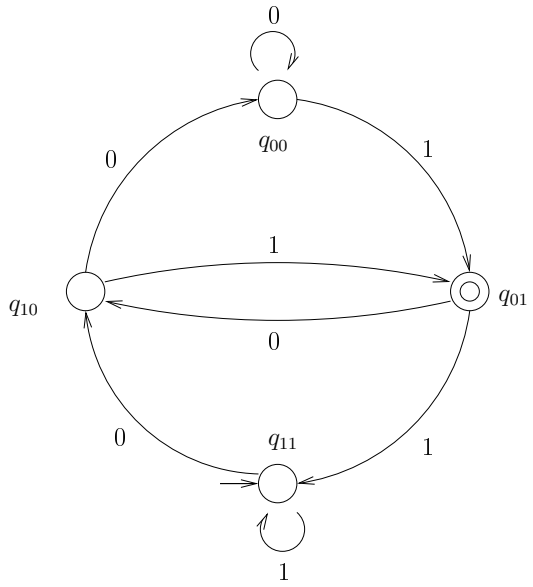

Figure 1.1: State diagram of the 1DFA D.

Definition 1.3. A *one-way nondeterministic finite automaton* (1NFA) is a five-tuple,

$$N = (Q, \Sigma, q_1, F, \delta),$$

where Q, Σ, q_1, F are as in the definition on the 1DFA and $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function.

Similarly to 1DFAs, for the machine to accept string w, there needs to be a path from the starting configuration (state q_1 and the head at the first symbol of w) to a final configuration (some $q \in F$ and the head at the last symbol of w). The main difference this time is that for any pair (q, a), where $q \in Q$, $a \in \Sigma$, the automaton may transition to more than one state (while definitely moving the head to the next symbol). $\delta(q, a)$ is a set of states, the set of all states the automaton may transition to. So, if $q_1 \in \delta(q, a)$ and $q_2 \in \delta(q, a)$ then the automaton may move to any of these two states. In order for the automaton to accept w, there needs to be at least one path (among all possible paths) to an accepting state. A note on non determinism: it doesnt matter which state, the automaton, decides to transition to. What is essential to the computation is the possibility of having a successful computation. In that sense we may say that the automaton, in every step, splits into many incarnations and, if one accepts, then the automaton accepts. Or we may suppose the automaton guesses a successful path, if one exists, and follows it.

Function δ has been changed, in order to be kept a function. We could also define δ as a relation $\delta \subseteq Q \times \Sigma \times Q$.Continuing in the same fashion as before, we can define an extension of δ, $\delta' : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$

$$\delta'(\{q\}, a) = \delta(q, a)$$
$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a),$$

where $a \in \Sigma, q \in Q$ and $S \subseteq Q$. We can then expand δ' in $\delta'' : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ as such:

$$\delta''(S, a) = \delta'(S, a)$$
$$\delta''(S, aw) = \delta''(\delta'(S, a), w),$$

where $a \in \Sigma, q \in Q, w \in \Sigma^*$, and $S \subseteq Q$.

So now we can say that N accepts w if $\delta''(q_1, w) \cap F \neq \emptyset$. From now on, we will be referring to δ' and δ'' as δ. That means N accepts w if there exists a state $q_f \in F$ such that $q_f \in \delta(q_1, w)$. And that means that, given w, there is a computation path from q_0 to q_f.

Example 2. We will construct a 1NFA that recognises the language

$$L = \{w | w \in \Sigma^* \text{such that the fourth symbol from the end of } w \text{ is the symbol 1}\}.$$

$N = (Q, \Sigma, q_0, \{q_4\}, \delta)$, where $Q = \{q_0, q_1, q_2, q_3, q_4\}$ and δ is described in the table below.

δ	0	1
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\{q_3\}$	$\{q_3\}$
q_3	$\{q_4\}$	$\{q_4\}$
q_4	$\emptyset$	$\emptyset$

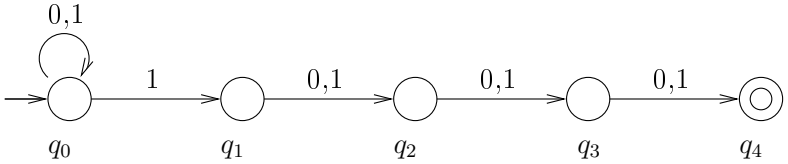

Figure 1.2: The state diagram of the 1NFA N.

Figure 1.2 repserents the states and transitions of the 1NFA. As we can see, while reading the input string, the machine stays on the starting state. At any point, where it reads the symbol 1, it may move to state q_1, and then it will count 3 more symbols before halting. If there are exactly 3 more symbols, the 1NFA accepts. We may say that N guesses the 1 that is fourth from the end. If the string has that property then there is a computation path through which N halts at the accepting state.

1.2.3 Configuration

Before we continue with the definition of another type of automaton, we introduce a concept that will help us understand computation even further.

Definition 1.4. A *configuration* is a string of elements from Σ and Q that can define clearly and fully any snapshot of the machine during its computation. A simple way to describe a configuration would be, given machine M and input w, $(q, i) \in Q \times \{1, 2, \ldots, |w|\}$, which means the machine is in state q over the i-th symbol of w. An other way to define configurations is with a string $c = uqaw$, where $u, w \in \Sigma^*$, $q \in Q$ and $a \in \Sigma$, and this means that the automaton, given the input string uaw, is at state $q$and the head is currently reading symbol a between strings u and w. From now on we will be using this way to describe configurations. Keep in mind that there are many possible configurations for a given input, namely $|Q| \times$ (length of input + 1), but not all of these configurations are reachable from the starting configuration, meaning some of these configurations might be impossible for the machine to be in.

From the definition of the automata, for input $w \in \Sigma^*$ the starting configuration is $c_0 = q_1 w$. Let $w = vau$, $c = vqau$ and $c' = vapu$ where $a \in \Sigma$ and $v, u \in \Sigma^*$. Then if $p \in \delta(q, a)$, we write $c \overset{M}{\to} c'$, and say that configuration c yields configuration c' in one step. A configuration that does not yield any other configuration in one step is a halting configuration. A halting configuration can be either an accepting configuration or a rejecting configuration. An accepting configuration has to be in this form: $c = wq$ where $q \in F$. The set of all configurations of machine M on input w, is denoted as $C_{M,w}$. This set can be seen as a graph, where the elements of $C_{M,w}$ are the nodes and the possible transitions between them ,$c \overset{M}{\to} c'$, are the edges. We call it the graph of configurations. The number of possible configurations is $(|w|+1)|Q|$, where $|w|+1$ is the possible positions of the head and $|Q|$ is the possible states the machince could be in. It is obvious that in order for string w to be accepted by the machine, there has to be a path from the starting configuration c_0 to an accepting configuration, in the graph of configurations.

Finally we define the set of functions $(C_{M,w} \to \mathbb{B})$, where $\mathbb{B} = \{0, 1\}$ is the set of boolean values true and false, represented as 1 and 0 respectively. A function $f \in (C_{M,w} \to \mathbb{B})$ assigns a boolean values to every configuration in set $C_{M,w}$. But what does 1 and 0 mean assigned to the configurations of $C_{M,w}$. The idea behind this to trace the property of a configuration being on an accepting path in a reverse way. We start on the final configurations $c_i = wq_{c_i}$ and we assign 1 to the accepting ones (those where $q_{c_i} \in F$) and 0 to the non accepting ones. Any other configuration is assigned the value 1 if it can inherit the property from its yeilding configurations, or the value 0 if it inherits the negation of the property. This means that the starting configuration will be assigned a boolean value as well. If the starting configuration is assigned the boolean value 1 then that means that the starting configuration meets the accepting conditions. So in that case M accepts w. Otherwise, if it is assigned the value 0 and M does not accept w.

This type of functions will help us define acceptance for the following automaton.

1.2.4 One-way alternating finite automaton

The next generalization of non determinism is alternation.

Definition 1.5. A *one-way alternating finite automaton* (1AFA) is a six-tuple,

$$A = (Q, \Sigma, q_1, F, U, \delta),$$

where $Q, \Sigma, q_1, F, \delta$ are as in the definition of 1NFA and $U \subseteq Q$ is the set of universal states.

The key difference between this definition and the one of the 1NFA is the set of universal states. We have two types of states, the existential ones and the universal ones. Every state must be of the one or the other type. Existential states, are states similar to the states of the 1NFAs. They can transition to more than one state, given the same symbol of the alphabet, and they are on an accepting path, if at least one of their transitions is. Universal states need all their transitions to be on an accepting path. Whatever we say about the states, can be interpreted equivalently for the configurations and the assignment of truth values to them. Configurations on existential states may yield more than one configuration, but need only one yelding configuration to be true, in order for them to be true. Universal configurations need all their yeilding configurations to be true, in order for them to be true. So in order for the machine to accept there needs to be a tree of configurations, where the leaves are accepting configurations.

Acccoding to the above, we can recursively define $P_{A,w} : C_{A,w} \to B$.

$$P_{A,w}(c) = \begin{cases} 1 & \text{if } c \text{ is accepting} \\ 0 & \text{if } c \text{ is rejecting} \\ \vee\{P_{A,w}(d) | c \xrightarrow{A} d\} & \text{if } q_c \notin U \text{ and } c \text{ is not halting} \\ \wedge\{P_{A,w}(d) | c \xrightarrow{A} d\} & \text{if } q_c \in U \text{ and } c \text{ is not halting} \end{cases}$$

where max is the arithmetic equivalent of the logical operator OR, and min is the eqivalent of AND. We say that D accepts w if $P_{A,w}(c_0) = 1$, where c_0 is the starting computation. Alternatively we can say that, for D to accept w, there has to be a tree with root the starting configuration, and all leafs accepting configurations, where each existential node has one child (at most), and each universal node has all possible yielding configurations as children.

Example 3. This time we will use a different alphabet $\Sigma = \mathcal{P}(\{1, 2, 3, 4\})$. The language we will be using is

$$L = \{ab | a, b \in \Sigma \text{ such that } a \subseteq b\ \}.$$

We define $A = (Q, \Sigma, q_0, \{q_f\}, \{q_0\}, \delta)$, where $Q = \{q_0, q_e, q_1, q_2, q_3, q_4, q_f\}$ and δ is defined by the transitions below.

$\delta(q_e, a) = \{q_f\}$

$$\delta(q_0, a) = \begin{cases} \{q_e\} & \text{if } a = \emptyset \\ \{q_i | i \in a\} & \text{if } a \neq \emptyset \end{cases}$$

$$\delta(q_i, a) = \begin{cases} \{q_f\} & \text{if } i \in a \\ \emptyset & \text{if } i \notin a \end{cases}$$

where $i \in \{1, 2, 3, 4\}$.

There are no other transitions than the ones described above. The only universal state is the starting one. How this machine works: On the first symbol it reads, the machine makes a universal move to the states that represent the elements of that first symbol (the symbol is a set in $\mathcal{P}(\{1, 2, 3, 4\})$. Then the machine varifies for each element that it is contained in the second symbol. If an element of the first set is in the second set, then the machine moves to the accepting state. If all elements of the first set are included in the second, then all paths lead to the accepting state, and the machine accepts. In the special case where the first symbol is the empty set, the machine just verifies that the input consists of two symbols. The state diagram is in Figure 1.3, where a_i is any set such that $i \in a_i$ and accordingly, b_i is any set that contains i. b is any set. Formally $\{i\} \subseteq a_i, b_i$.

A note on P and δ: these functions work in a reverse way to each other (even thought the first one is over configurations and the second one over states and symbols). P assigns the truth to the halting configurations and spreads backwards towards the starting configuration while δ is applied from the starting configuration (meaning starting state and first symbol), and finds its way through an accepting configuration (meaning a final state, while the head has run through the whole word w).

1.2.5 The grid of configurations

Let's take a moment now to consider a different view on the set of configurations $C_{M,w}$. As mentioned above, the set $C_{A,w}$ has $|Q| \times (|w| + 1)$ distinct elements. We can describe each configuration as an element of $Q \times \{1, 2, \ldots, |w| + 1\}$. Figure 1.3 represents a way we can describe this set.

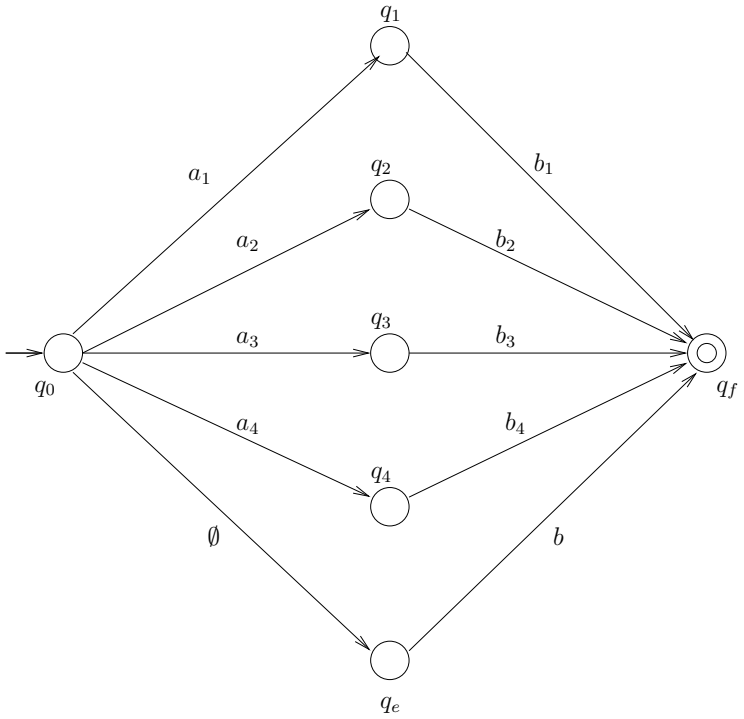

Figure 1.3: The state diagram of the 1AFA A.

We can clearly identify the starting configuration on the top left corner and the halting configurations (the rightmost column) some of them accepting, some of them rejecting. As long as we are looking into one-way machines, we know that each node on the grid can only relate to the elements of the next row. This is essential to understand the accepting process for our next machine, the one-way boolean finite automaton.

1.2.6 One-way boolean finite automaton

A boolean automaton is a generalization of all previous models.

Definition 1.6. A *one-way boolean finite automaton* is a five-tuple,

$$B = (Q, \Sigma, q_1, F, g)\,,$$

where Q, Σ, q_1, F are as usual and $g : (Q \times \Sigma) \to (\mathbb{B}^k \to \mathbb{B})$ is a function. $\mathbb{B} = \{0, 1\}$ is the set of boolean values true and false.

Function g assigns a boolean function $g(\cdot, \cdot) : \mathbb{B}^k \to \mathbb{B}$, to every combination of state and symbol $(Q \times \Sigma)$. As mentioned earlier, every node on the grid of configurations, is a configuration, having a state and a symbol that is being read. Those two define, through g, a boolean function. So we can assign on every element of the grid a boolean function over $\mathbb{B}^k$.

For any given configuration $c = uqav$ we define recursively $P_{B,w} : C_{B,w} \to \mathbb{B}$ as follows:

$$P_{B,w}(c) = \begin{cases} 0 & \text{if } c \text{ is rejecting} \\ 1 & \text{if } c \text{ is accepting} \\ g(q,a)(P_{B,w}(d_1), P_{B,w}(d_2), \ldots, P_{B,w}(d_k)) & \text{if } c \text{ is not halting} \end{cases}$$

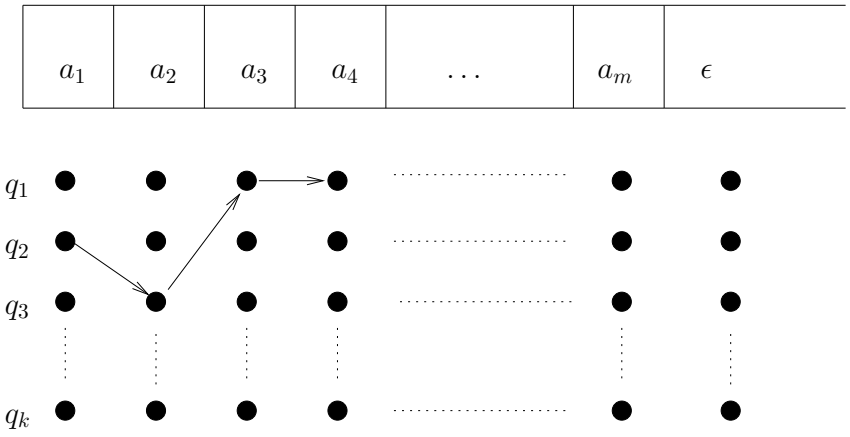

Figure 1.4: The grid of configurations.

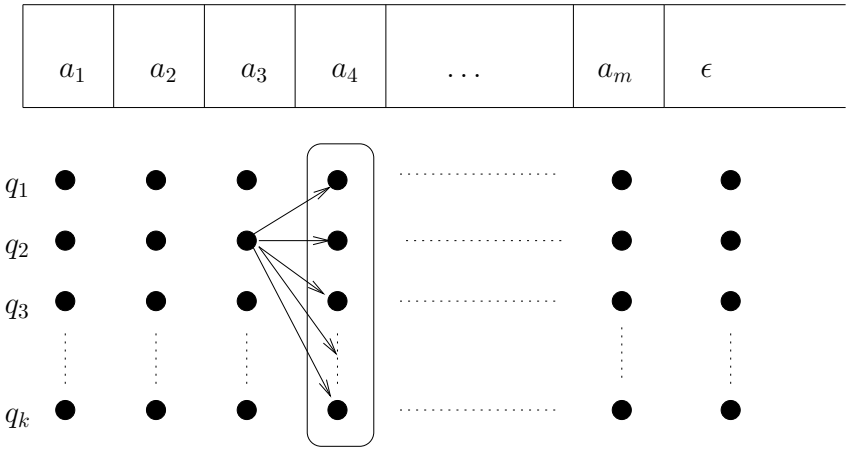

Figure 1.5: The viariables of the boolean function $g(q_2, a_3) : \mathbb{B}^k \to \mathbb{B}$.

where $d_i = uaq_iv$, meaning all k configurations of B, on input w, that are on the column to the right of c, in the configuration grid. In other words, the function assigns a boolean value to each node on the configuration grid, accordring to the values it assigned on the next column.

B accepts w if $P_{B,w}(c_0) = 1$, where c_0 the starting configuration.

1.2.7 Computability equivalence

Theorem 1 (computability equivalence of one-way finite automata [9])**.** *For every k-state 1BFA there is a 2^{2^k}-state 1DFA that recognises the same language.*

Proof. Let $M = (Q, \Sigma, q_1, F, g)$ be a 1BFA. We will create a 1DFA $D = (Q_D, \Sigma, q_0, F_D, \delta_D)$, that accepts the language of M. Let $|Q| = k$ and u denote a k-tuple of boolean values $u = (u_1, u_2, \ldots, u_k) \in \mathbb{B}^k$. Let π_i be the i-th projection, $\pi_i(u) = u_i$. Let f denote the characteristic vector of F,

$$\pi_i(f) = \begin{cases} 1 & \text{, if } q_i \in F \\ 0 & \text{, if } q_i \notin F. \end{cases}$$

Now we can define D:

- $Q_D = (\mathbb{B}^k \to \mathbb{B})$, the set of all functions from $\mathbb{B}^k$ to $\mathbb{B}$.
- $q_0 = \pi_1$ is the starting state.
- $F_D = \{h | h \in (\mathbb{B}^k \to \mathbb{B}) : h(f) = 1\}$, meaning that of all functions-states in $(\mathbb{B}^k \to \mathbb{B})$, final are considered the ones that give the boolean value 1 to vector f.
- Finally the transition function is $\delta_D : Q_D \times \Sigma \to Q_D$. For every $h \in Q_D = \mathbb{B}^k \to \mathbb{B}$ and $a \in \Sigma$, we have:

$$\delta_D(h, a) = h \circ (g(q_1, a), g(q_2, a), \ldots, g(q_k, a)), \forall a \in \Sigma.$$

Let $t_a : \mathbb{B}^k \to \mathbb{B}^k$, such that $t_a = (g(q_1, a), g(q_2, a), \ldots, g(q_k, a))$, for $a \in \Sigma$. Figure 1.6 explains the moves of the transition function. The grid can be seen as a sequence of functions of vectors (of functions). The only important configuration from the first column is that of the starting state, so for input $w = a_1 \cdots a_m$, there is a function $g(q_1, a_1) : \mathbb{B}^k \to \mathbb{B}$ that relates the value of that node to the values of the nodes of next column. From there on the values of each column, are assigned by applying function $t_a : \mathbb{B}^k \to \mathbb{B}^k$, on the values of the nodes of the next column. But D does not have to keep in its memory this function because it can compose it with the current state-function. The composition of $h : \mathbb{B}^k \to \mathbb{B}$ with $t_a : \mathbb{B}^k \to \mathbb{B}^k$ is $h \circ t_a : \mathbb{B}^k \to \mathbb{B}$.

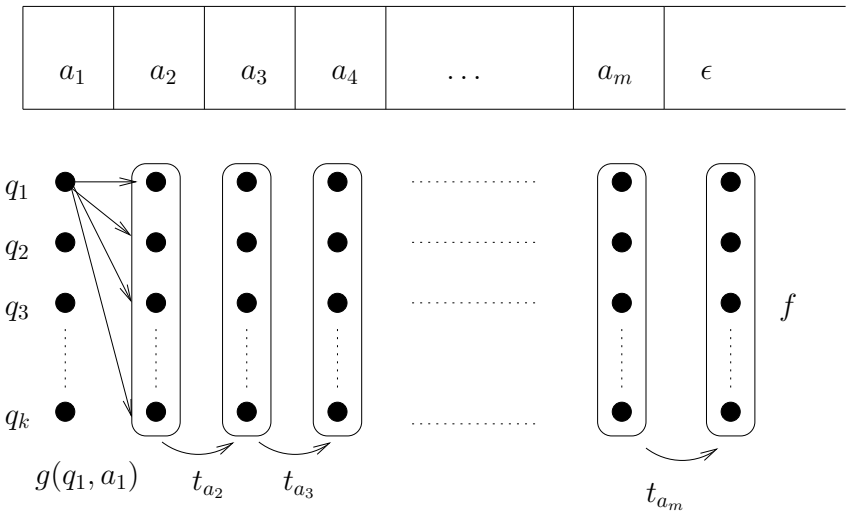

Figure 1.6: The functions of the vectors on the grid.

$\delta_D(\pi_1, a_1) = g(q_1, a_1)$,
$\delta_D(\pi_1, a_1 a_2) = g(q_1, a_1) \circ t_{a_2}$,
$\delta_D(\pi_1, a_1 a_2 a_3) = g(q_1, a_1) \circ t_{a_2} \circ t_{a_3}$ and so forth.
When the machine reaches the final state-function, it uses vector f to decide if M accepts of rejects (that is the way we have defined set F_D). □

Corollary 1. *All one-way finite automata we defined so far are equivalent.*

Proof. It is immediate from Theorem 1, since 1BFA is a generalization of all the one-way finite automata we defined. □

1.3 Two-way automata

The main difference between two-way automata and one-way automata is the movement of the head that reads the input. In this case, the head can move left or move right or remain stationary. This translates into three types of transitions in the transition function. The computation may continue by moving the head to the left or to the right, or by remaining at the current cell, reading the same symbol. This creates two types of problems that need to be dealt with. The first one is quite typical. There is the possibility of the head being driven out of the input tape. From now on we will be using two new symbols,$\vdash$ and $\dashv$, that cannot be part of our alphabet. The input w will be given in the form $\vdash w \dashv$. This will help the automata, whenever they read an end-marker, to never move any further than that. The second problem we encounter is the possibility of the automaton entering an infinite loop, never completing its computation. This means that the automaton may be unable to answer for some inputs. We will consider a string w to be accepted, only if the computation on w halts and returns a positive answer.

1.3.1 Two-way deterministic finite automaton

Definition 1.7. A *two-way deterministic finite automaton* (2DFA) is a five-tuple,

$$D = (Q, \Sigma, q_1, F, \delta),$$

where Q, Σ, q_1, F are as usual and $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to Q \times \{L, S, R\}$ is the transition function.

Before we define acceptance there are a few things that need to be stated about δ. The transition function works pretty much the same way in every automaton, whether it is one-way or two-way. The differences are the following: A value in $\{L, S, R\}$ defines whether the head of the machine will move to the left, stay at the same position, or move to the right, respectively. There is also the restriction that whenever $\delta(q, \vdash) = (p, x)$, x can only be R or S, while whenever $\delta(q, \dashv) = (p, R)$, $p = q$ and q can only be final. The automaton D, given input w, accepts if it reaches configuration a final configuration $c_f = \vdash w \dashv q_f$, where $q_f \in F$. So this means that the computation starts from the starting configuration and ends in an accepting configuration. There are two basic differences with the one-way model. The first one is that the head is not always moving torwards the halting configurations. Actually, a configuration could be yeilding, in one step, any configuration on the same column, or on the two adjacent columns to its left and to its right. The second difference is the possibility of the computation entering an infinite loop. We can see that the set of configurations is finite. This means that in order for the computation to never end, the path needs to repeat a set of configurations. This can be detected if the computation path grows longer than the number of possible configurations (by the pigeonhole principle).

The partial function $P_{D,w} : C_{D,w} \to B$ is defined as follows:

$$P_{D,w}(c) = \begin{cases} 1 & \text{if } c \text{ is accepting} \\ 0 & \text{if } c \text{ is rejecting} \\ P_{D,w}(d) & \text{if } c \text{ is not halting and } c \xrightarrow{D} d. \end{cases}$$

Some configurations may stay undefined (all the configurations that have no path to halting configurations, to be exact), so this function is partial, since it can assign boolean values only to some elements of $C_{D,w}$. D accepts w if $P_{D,w}$ is defined on c_0 and $P_{D,w}(c_0) = 1$.

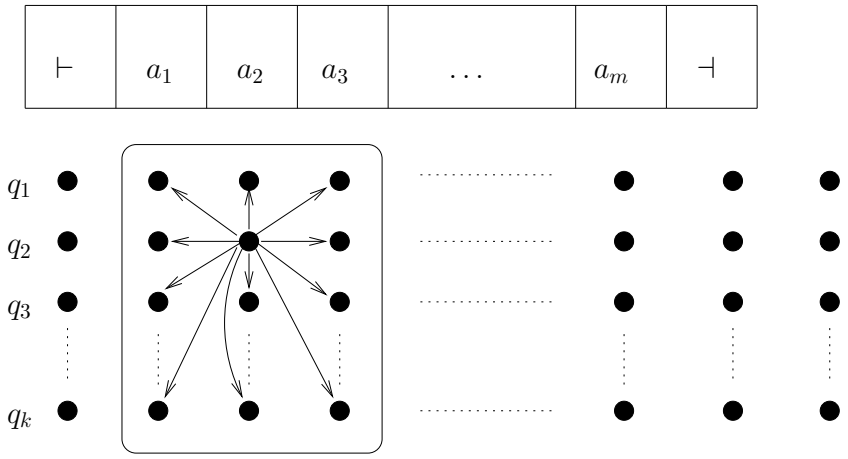

Figure 1.7: The possible transitions out of configuration $\vdash a_1 q_2 a_2 \cdots a_m \dashv$ are in the frame.

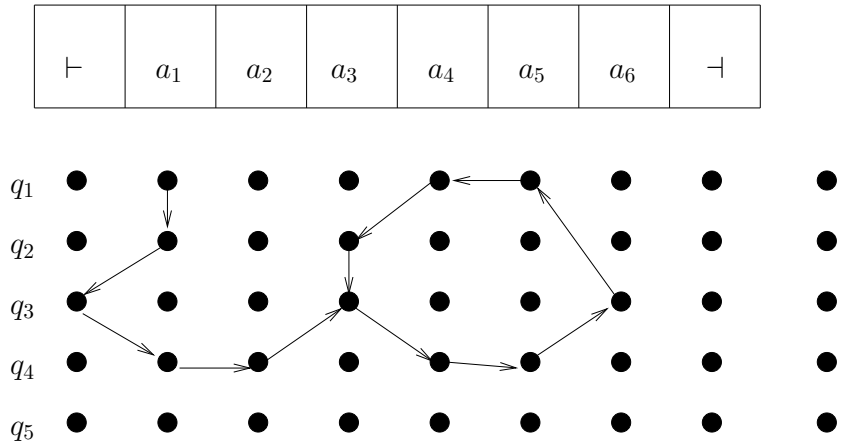

Figure 1.8: An example of a non ending computation (loop), on the configuration grid.

Example 4. We will define a 2DFA that recognises the language of example 3. We define the 2DFA $D = (Q, \mathcal{P}(\{1,2,3,4\}), q_1, \{q_f\}, \delta)$, where $Q = \{q_1, q_1', q_2, q_2', q_3, q_3', q_4, q_4', q_5, q_f\}$ and δ is defined as follows:

$$\text{For } i \in \{1,2,3\},\ \delta(q_i, a) = \begin{cases} (q_i', R) & \text{if } i \in a \\ (q_{i+1}, S) & \text{if } i \notin a \end{cases}$$

$$\text{For } i \in \{1,2,3\},\ \delta(q_i', a) = \begin{cases} (q_{i+1}, L) & \text{if } i \in a \\ undefined & \text{if } i \notin a \end{cases}$$

$$\delta(q_4, a) = \begin{cases} (q_4', R) & \text{if } i \in a \\ (q_5, R) & \text{if } i \notin a \end{cases}$$

$$\delta(q_4', a) = \begin{cases} (q_5, S) & \text{if } i \in a \\ undefined & \text{if } i \notin a \end{cases}$$

$$\delta(q_5, a) = (q_f, R)$$

$$\delta(q_f, \dashv) = (q_f, R)$$

There are no transitions other than the ones described. This concludes the description of the machine. The machine moves back and forth checking one by one each element. If an element is included in the first set, then the head moves to the right and verifies that same element is included in the second set. Then it continues to the next elment. When it checks the last one, returns to the first symbol and counts two symbols, before moving

to the accepting state. The automaton is also described in Figure 1.9, where a_i stands for any set that $q_i \in a_i$ and a'_i stands for any set that $q_i \notin a'_i$. Same goes for b_i, while b stands for any set.

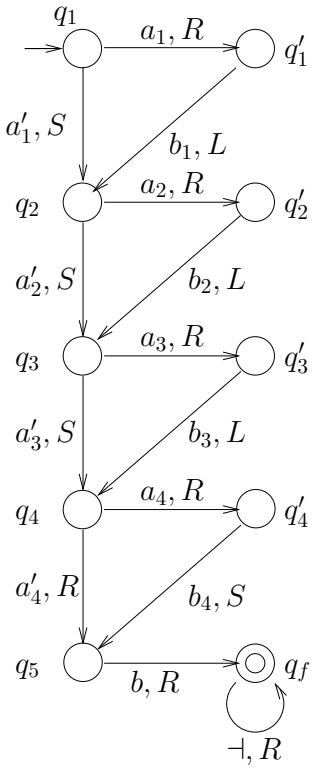

Figure 1.9: The state diagram of D.

1.3.2 Two-way nondeterministic finite automaton

Definition 1.8. A *two-way nondeterministic finite automaton* (2NFA) is a five-tuple,

$$N = (Q, \Sigma, q_1, F, \delta)\,,$$

where Q, Σ, q_1, F are as usual and $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to \mathcal{P}(Q \times \{L, S, R\})$ is the transition function.

Function δ has the same behavior conserning $\vdash$ and $\dashv$, as the function of a 2DFA: if $(p, x) \in \delta(q, \vdash)$ then $x \in \{S, R\}$ and if $(p, R) \in \delta(q, \dashv)$ then $p = q \in F$. Similarly to the 1NFA, we are looking for a path among all possible computation paths, which will lead us to an accepting configuration. This time, if there exists a path between the starting configuration and an accepting configuration then there is an equivalent path (same start and same end) with length no greater than the total number of configurations. This means that, similarly to 2DFAs, if there is no path of at most that length, then there is no path at all.

Similarly to 2DFAs, a partial function $P_{N,w} : C_{N,w} \to \mathbb{B}$ is defined as follows:

$$P_{N,w}(c) = \begin{cases} 1 & \text{if } c \text{ is accepting} \\ 0 & \text{if } c \text{ is rejecting} \\ 1 & \text{if } c \text{ is not halting and } P_{N,w}(d) = 1 \text{ for some } d \text{ such that } c \xrightarrow{N} d \\ 0 & \text{if } c \text{ is not halting and } P_{N,w}(d) = 0 \text{ for all } d \text{ such that } c \xrightarrow{N} d \end{cases}$$

As in the 2DFA model, a path from the starting configuration c_0, to an accepting configuration exists iff $P_{N,w}$ is defined on c_0 and $P_{N,w}(c_0) = 1$.

1.3.3 Partially ordered sets

As we saw on the two previous models, there is something special about our P functions, they are partial funtions. To tackle this problem we will extend set $\mathbb{B}$. We now define set $\mathbb{B}^+ = \mathbb{B} \cup \{\bot\}$. The $\bot$ element is called bottom element and is assigned to all the configurations whose values are undefined. Let $(\mathbb{B}^+, \sqsubseteq)$ be a partial order over $\mathbb{B}^+$, such that $\bot \sqsubset 1$, $\bot \sqsubset 0$, and 0 and 1 are not comparable. In the same way, we define the

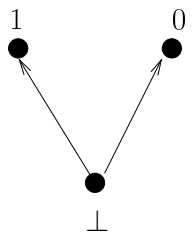

Figure 1.10: The partial order *sqsubseteq* on $\mathbb{B}^+$.

partially ordered set $((\mathbb{B}^+)^k, \sqsubseteq)$ where for all a,b in $(\mathbb{B}^+)^k$, we have $a \sqsubseteq b \iff (\forall i)(a_i \sqsubseteq b_i)$, where a_i and b_i are the i-th coodrinates of a and b, respectively. This partial order considers the defined value greater than the undefined value. So, if a vector a is greater than a vector b, then a is "more defined" than b i.e. has as many or fewer $\bot$ elements in the same coordinates as b. The addition of an extra symbol to the set of boolean values creates a problem in the way we handled values so far. We need to extend the boolean functions $f' : (\mathbb{B}^+)^k \to \mathbb{B}^+$ so that they agree with their restrictions over $\mathbb{B}^k$, and at the same time they do not assign arbitrary values. The rule is simple: For $x \in (\mathbb{B}^+)^k \setminus \mathbb{B}^k$, if all $y \in \mathbb{B}^k$ such that $x \sqsubseteq y$ agree on $f(y)$, then $f'(x) = f(y)$, otherwise $f'(x) = \bot$. This simple rule can be justified as follows: If all $y \in \mathbb{B}^k$ such that $x \sqsubseteq y$ share the same $f(y)$ value, then the undefined coordinates of x do not affect the value of $f(x)$ (they could be set to either true or false, and the result would stay the same), otherwise, the undefined coordinates do affect the resulting value, so $f(x)$ cannot be defined either.

This partial order can help us create another partial order on $(C \to \mathbb{B}^+)$. Let $P_1, P_2 \in (C \to \mathbb{B}^+)$. We say that $P_1 \sqsubseteq_* P_2$ if $P_1(x) \sqsubseteq P_2(x)$, for all $x \in C$. This relation can be described as: P_2 is more defined than P_1. There is a bottom element in this poset, $\bot_*$, which is the function not defined anywhere: $\bot_*(x) = \bot$, for all $x \in C$. By definition $\bot_* \sqsubseteq P$, for all $P \in (C \to \mathbb{B}^+)$.

1.3.4 Two-way alternating finite automaton

Definition 1.9. A *two-way alternating finite automaton* (2AFA) is a six-tuple,

$$A = (Q, \Sigma, q_1, F, U, \delta),$$

where Q, Σ, q_1, F, U are the same as in the one-way model and $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to \mathcal{P}(Q \times \{L, S, R\})$ is the transition function.

The same restrictions for δ over the end-markers apply and the set of states U plays the same role as in the one-way model. We define $P_{A,w} : C_{A,w} \to \mathbb{B}$ as follows:

$$P_{A,w}(c) = \begin{cases} 1 & \text{if } c \text{ is accepting} \\ 0 & \text{if } c \text{ is rejecting} \\ \vee\{P_{A,w}(d) | c \xrightarrow{A} d\} & \text{if } c \text{ is not halting and } q_c \notin U \\ \wedge\{P_{A,w}(d) | c \xrightarrow{A} d\} & \text{if } c \text{ is not halting and } q_c \in U \end{cases}$$

We may now extend $P_{A,w}(c)$ to $P'_{A,w} : C_{A,w} \to \mathbb{B}^+$. This function helps up define functional $\tau : (C_{A,w} \to \mathbb{B}^+) \to (C_{A,w} \to \mathbb{B}^+)$ the following way:

$$\tau(P)(c) = \begin{cases} 1 & \text{if } c \text{ is accepting} \\ 0 & \text{if } c \text{ is rejecting} \\ \vee\{P(d) | c \xrightarrow{A} d\} & \text{if } c \text{ is not halting and } q_c \notin U \\ \wedge\{P(d) | c \xrightarrow{A} d\} & \text{if } c \text{ is not halting and } q_c \in U \text{ and } c \end{cases}$$

for all $P \in (C_{A,w} \to \mathbb{B}^+)$. By $\perp_*$, the bottom element of $(C_{A,w} \to \mathbb{B}^+)$, where all configurations are valued undefined, and this functional, we create a non-decreasing sequence of functions in $(C_{A,w} \to \mathbb{B}^+)$.

$$P_{A,w,0} = \perp_*$$
$$P_{A,w,i} = \tau(P_{A,w,i-1})$$

The set $(C_{A,w} \to \mathbb{B}^+)$ is finite (due to $C_{A,w}$ and $\mathbb{B}^+$ being finite), therefore this non-decreasing sequence (in the sence of more or equally defined) has a least fixed point. We denote this least fixed point $P_{A,w,\tau}$, and by definition $\tau(P_{A,w,\tau}) = P_{A,w,\tau}$, which means that this function contains the minimum amount of information that can be derived from the rules of $P_{A,w}$. Therefore we say that A accepts w when $P_{A,w,\tau}(c_0) = 1$, where c_0 is the starting configuration.

1.3.5 Two-way boolean finite automaton

Definition 1.10. A *two-way boolean finite automaton* (2BFA) is a five-tuple,

$$B = (Q, \Sigma, q_1, F, g)\,,$$

where Q, Σ, q_1, F are as in the one-way model and $g : (Q \times (\Sigma \cup \{\vdash, \dashv\})) \to (\mathbb{B}^{3k} \to \mathbb{B})$ is a function.

Function g, as in 1BFAs, assigns a boolean function $g(\cdot, \cdot) : \mathbb{B}^{3k} \to \mathbb{B}$, to every combination of state and symbol $(Q \times \Sigma)$. As mentioned earlier, every node on the grid of configurations, is a configuration, having a state and a symbol that is being read. Those two define, through g, a boolean function. So we can assign on every element of the grid a boolean function over $\mathbb{B}^{3k}$. Similarly to the above, we define $P_{B,w} : C_{B,w} \to \mathbb{B}$ as follows:

$$P_{B,w}(c) = \begin{cases} 1 & \text{if } g(q_c, a_c) \text{ is the constant function 1} \\ 0 & \text{if } g(q_c, a_c) \text{ is the constant function 0} \\ g(q_c, a_c)(P_{B,w}(d_1), P_{B,w}(d_2), \ldots, P_{B,w}(d_{3k})) & \text{if } g(q_c, a_c) \text{ is not a constant function} \end{cases}$$

We also define functional $\tau : (C_{B,w} \to \mathbb{B}^+) \to (C_{B,w} \to \mathbb{B}^+)$ as follows:

$$\tau(P)(c) = \begin{cases} 1 & \text{if } g(q_c, a_c) \text{ is the constant function 1} \\ 0 & \text{if } g(q_c, a_c) \text{ is the constant function 0} \\ g(q_c, a_c)(P(d_1), P(d_2), \ldots, P(d_{3k})) & \text{if } g(q_c, a_c) \text{ is not a constant function} \end{cases}$$

for all $P \in (C_{B,w} \to \mathbb{B}^+)$, where $g(q, a)$ is a boolean function dependent on function g (given in the definition of B), with domain $(\mathbb{B}^+)^{3k}$. The first k configurations, d_1,d_2,...,d_k, are the configurations yielded by c when the head moves to the left. The next k configurations $d_{k+1}, d_{k+2} \ldots d_{2k}$ are the configurations yielded by c when the head remains stationary. Finally, $d_{2k+1}, d_{2k+2} \ldots d_{3k}$ are the last k configurations yielded by c with head position to the right of the cell of c. All of these are the configurations that c may yield in one step. If $q \in F$ then $g(q, \dashv) = 1$ the constant function of $\mathbb{B}^3 \to \mathbb{B}$, but there might be other constant functions $g(q, a)$ as well. Similarly to 1AFAs, we define the following sequence:

$$P_{B,w,0} = \bot_*$$
$$P_{B,w,i} = \tau(P_{B,w,i-1})$$

We denote this least fixed point $P_{B,w,\tau}$, and by definition $\tau(P_{B,w,\tau}) = P_{B,w,\tau}$, which means that this function contains the minimum amount of information that can be derived from the rules of $P_{B,w}$. Acceptance is defined the same way it was defined for the 2AFA: B accepts w when $P_{B,w,\tau}(c_0) = 1$ where c_0 is the starting configuration.

1.3.6 Computability equivalence

Theorem 2 (computability equivalence [10])**.** *All the two-way finite automata defined in this chapter are equivalent and they are equivalent with the one-way finite automata as well.*

Proof. It is clear from Lemma 1 and Lemma 2 below that for every 2BFA there is an equivalent 1DFA. All other two-way models are restrictions of the 2BFA, therefore all two-way models are equivalent to each other and equivalent to 1DFAs. □

We will introduce a restriction of the 2AFA model, named deterministic-movement 2AFA. This model with serve as an intermediate model, for prooving the equivalence of 2BFAs and 1DFAs.

Definition 1.11. A *deterministic-movement two-way alternating finite automaton* (DM-2AFA) is a six-tuple,

$$A = (Q, \Sigma, q_1, F, U, \delta),$$

where $Q, \Sigma, q_1, F, U, \delta$ are as in the 2AFA model. Beyond the standard restrictions of δ over the two end-markers, whenever the head moves to the left or right, the machine can only make a deterministic transition to another state. Formally, δ is allowed to have only 3 types of moves: $\delta(q, a) = \{(p, L)\}$, $\delta(q, a) = \{(p, R)\}$ and $\delta(q, a) = \{(p_1, S), (p_2, S), \ldots, (p_m, S)\}$. Hence every step is repsectively deterministic with head movement to the left or deterministic with head movement to the right, or existential or universal with no head movement. Finally, the DM-2AFA ends all its computations moving right after the right end-marker, meaning, the machine cannot halt while the head is not positioned over the right end-marker.

Lemma 1 (2BFA to DM-2AFA)**.** *For every k-state 2BFA there is a $2(3k + 2^{6k})$-state DM-2AFA that recognises the same language.*

Proof. Let $M = (Q, \Sigma, q_1, F, g)$ be a 2BFA. We will construct a DM-2AFA $A = (Q_A, \Sigma, q_1, F_A, U_A, \delta)$ such that they accept the same language. We will need to substitute the boolean functions $g(q, a) : B^{3k} \to B$, with existential and universal moves. Every $g(q, a) : B^{3k} \to B$ can be written in conjunctive normal form. This is essential to the construction of the DM-2AFA. The description of A is as follows:

- Defining Q_A: We define the set $Q^- = \{q^- | q \in Q\}$, which consists of one negative state for each original state in $Q = Q^+$.

  We also define $Q_L^+ = \{q_L | q \in Q\}$ and $Q_R^+ = \{q_R | q \in Q\}$, that are just sets similar to Q^+, only tagged with an L or an R, for left and right. Similarly, we define Q_L^- and Q_R^-.

  Let $Q_{lit}^+ = Q^+ \cup Q_L^+ \cup Q_R^+$ and $Q_{lit}^- = Q^- \cup Q_L^- \cup Q_R^-$, and $Q_{lit} = Q^+ \cup Q^- \cup Q_L^+ \cup Q_R^+ \cup Q_L^- \cup Q_R^-$. It is immediate that $|Q_{lit}| = 6k$.

  In a similar fashion, we call Q_c^+ the set of all clauses that can be made using the elements of Q_{lit} as variables. We can easily describe Q_c^+ as the powerset of Q_{lit}, $Q_c^+ = \mathcal{P}(Q_{lit})$. That way, every subset of Q_{lit} denotes the clause of the elements of that subset. We also define $Q_c^- = \{q^- | q \in Q_c^+\}$.

  We name the set $Q_A^+ = Q^+ \cup Q_c^+ \cup Q_L^+ \cup Q_R^+$, the set of positive states.

  Finally, we define the set $Q_A^- = \{q^- | q \in Q_A^+\}$, the mirror set of Q_A^+, which is identical but with a tag of the minus symbol on each element, the set of negative states. So obviously $Q_A^- = Q^- \cup Q_c^- \cup Q_L^- \cup Q_R^-$.

  All preparation is over. We finally define $Q_A = Q_A^+ \cup Q_A^-$. If $|Q^+| = k$ then we can see that $|Q_A| = 2(3k + 2^{6k})$.

- F_A is the set of accepting states. Let $F^+ = F$ and $F^- = \{q^- | q \in F^+\}$. Then $F_A = F \cup (Q^- \setminus F^-)$.

- Defining U_A: $U_A = Q^+ \cup Q_c^-$. On a side note, the set of existential states is $Q_c^+ \cup Q^-$. The remaining states $Q_L^+, Q_R^+, Q_L^-, Q_R^-$ are states that move deterministically, so we may add them to any of the two sets.

- $\delta : (Q_A \times (\Sigma \cup \{\vdash, \dashv\})) \to \mathcal{P}(Q_A \times \{L, R, S\})$ is the transition function which will be described in detail below.

According to our restrictions, a state $q \in Q_A$ can be either universal, existential, or deterministic. So, it is immediate that $\delta(q, a) = \{(q_i, S), (q_j, S), \ldots, (q_m, S)\}$ in case q is existential or universal, and $\delta(q, a) = \{(q_i, L)\}$ or $\delta(q, a) = \{(q_i, R)\}$ in case q is deterministic.

Here is how δ works:

- $Q_L^+, Q_R^+, Q_L^-, Q_R^-$: The function is defined over these sets as follows.

  $$\delta(q_L, a) = \{(q, L)\}\,,\ \delta(q_R, a) = \{(q, R)\}\,,\ \delta(q_L^-, a) = \{(q^-, L)\}\,,\ \delta(q_R^-, a) = \{(q^-, R)\}$$

  for all $q \in Q^+$, and these are the only deterministic moves.

- Q_c^+ and Q_c^-: Each state in one of these sets represents a clause of a number of variables among Q^+, Q_L^+, Q_R^+. As described earlier $Q_c^+ = \mathcal{P}(Q_{lit})$, so that means $q_c \in Q_c^+$ is in fact $q_c \in \mathcal{P}(Q_{lit})$.

  So for $q_c \in Q_c^+$, $\delta(q_c, a) = \{(q, S) | q \in q_c\}$.

  And for $q_c^- \in Q_c^-$, $\delta(q_c^-, a) = \{(q^*, S) | q \in q_c\}$, where $q^* = \begin{cases} q^- & \text{if } q \in Q_{lit}^+ \\ p & \text{if } q \in Q_{lit}^- \\ & \text{and } q = p^-. \end{cases}$

- Q^+ and Q^-: We know that $g(q, a)$ is a boolean function in $(B^{3k} \to B)$. Every such function can be written in conjunctive normal form (CNF). CNF consists of a number of clauses under conjunction. Let $C_{g(q,a)} \subseteq Q_c^+$ be the set of clauses of the CNF form of $g(q, a)$. We have two sets that represent all clauses, Q_c^+ and Q_c^-.

  In case $q \in Q$ then $\delta(q, a) = \{(p, S) | p \in C_{g(q,a)}\}$.

  In case $q^- \in Q^-$ then $\delta(q^-, a) = \{(p^-, S) | p \in C_{g(q,a)}\}$.

This completes the description of δ. Machine A makes the same computation as machine M, but has many more states to eliminate existential and universal moves while the head moves left or right, and to replace boolean functions with CNFs. The elements of the sets Q_L^+ and Q_R^+ work as representatives of the states of the initial machine: Q_L^+ solves the problem of non-deterministic, or universal, movement to the left, and Q_R^+ the problem of non-deterministic, or universal, movement to the right. So, for example, if at any point the machine has to move left to state p, it just moves to p_L first, without moving its head, and then p_L moves the head deterministicaly to the left, while moving to state p. The other two methods used, the set of clauses and the duplication of the states (under the "minus" tag), are just tools for replacing the boolean functions in the computational process. What was done in one step in M, now takes three steps in A: For the boolean function $g(q, a)$, we construct the equivalent CNF. So q becomes a universal state, connecting under its universal branching the states for the appropriate clauses. Then the clause-states find their literals. If a literal has negation, then the rest of the computation moves to the mirror states, which have symmetrical transitions. All transitions stay within the mirror states, until a negative literal is found again. That is why the accepting and rejecting states have been reversed on the mirror machine. (Another way to interpret the duplication of the states is to consider it as method for counting modulo 2, the number of negations along each path.) □

Lemma 2 (DM-2AFA to 1DFA)**.** *For every k-state DM-2AFA there is a 2^{k2^k}-state 1DFA that recognises the same language.*

Proof. We will construct a 1DFA that recognises the same language as a given DM-2AFA. The DM-2AFA we will be using has one important characteristic: it starts its computation from the starting state at the far right of the input, $\dashv$. This model is equivalent to the original DM-2AFA, by the addition of a new starting state that moves the head to the other end of the input, berofe finally transitioning to the original starting state. Also, the input of the 1DFA is exactly the same as the input of the restricted 2AFA, including $\vdash$ and $\dashv$ at the start and end of the input, respectively.

Given a DM-2AFA $M = (Q, \Sigma, q_s, F, U, \delta)$, we construct a 1DFA $D = (Q_D, \Sigma, s_0, F_D, \delta_D)$. We also define $\bar{Q} = \{\bar{q} | q \in Q\}$.

- The set of states is $Q_D = \mathcal{P}(Q \times \mathcal{P}(\bar{Q}))$.

- The starting state is $s_0 = \delta_D(\vdash)$.
- $F_D = \{t \in Q_D | \exists (q_1, A) \in t : A = F\}$ is the set of accepting states.
- $\delta_D : Q_D \times (\Sigma \cup \{\vdash, \dashv\}) \to Q_D$ is the transition function which will be described in detail below.

The set of states of an automaton is its memory. In case of D, the set of states consists of sets of pairs (q, A), where $q \in Q$ and $A \subseteq \bar{Q}$. Let's suppose that D is in state $p \in Q_D$ and the head is over the a_i symbol of the input. Consider the grid of configurations in Figure 1.11. If $(q, A) \in p$ then there is a valid tree of computation of M, that corresponds

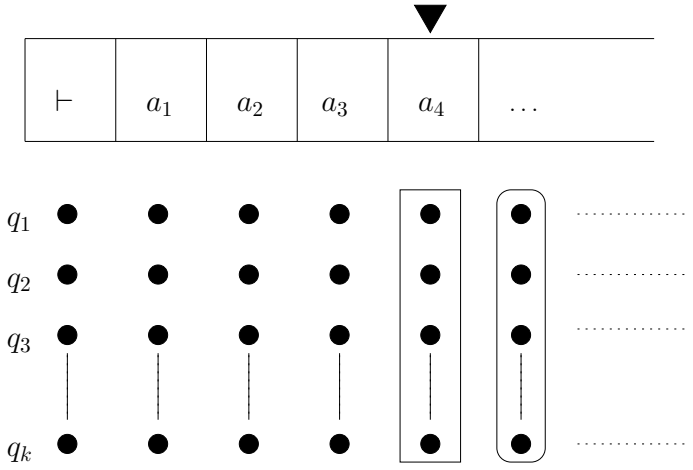

Figure 1.11: The first column is that of the roots and the second is that of the tagged leaves.

to a valid tree of configurations of M with the following charecteristics: The root of tree is the configuration of state q, while head is over symbol a_i. The leaves of the tree are the configurations with states those of set A, while the head reads symbol a_{i+1} Finally, all nodes, but the leaves, are on first $i + 1$ columns (since the machine has read only the first i symbols and has considered $\vdash$ as a first symbol without reading it). Keep in mind, it doesnt matter what a_{i+1} is, at this point. Also p does not know what the actual tree of configurations is, but knows the fact that a valid tree, with the above characteristics, exists. So every pair in p represents a valid tree of configuration, with root configuration in the first marked column, and leaves on the second marked column, in Figure 1.11. Let's name every pair a limb and $p \in Q_D$, a set of limbs.

So $\delta_D(p, a_{i+1}) = p'$, where p' is the set of limbs of D, while its head is over symbol a_{i+1}. The question that arises is the following: Knowing the limbs of D, while its head is over symbol a_i (this is p), and knowing the next symbol, a_{i+1}, can we deduce the limbs of D, while its head is over symbol a_{i+1} (this is p'). We will show how to calculate $\delta_D(q, a)$.

For this we will need to define the set $Q_D^* = \mathcal{P}(Q \times \mathcal{P}(\bar{Q} \cup Q))$, a superset of Q_D. Every element in $z \in Q_D^*$, is a set of pairs just like the elements of Q_D. The difference is that for $(q, A) \in z$, A has tagged and untagged elements. So $(q, A) \in z$ means that there is a tree of configurations just as above, with the difference that the leaves of the tree may be the states of configurations over symbol a_i as well. The states of these configurations are the untagged elements of A. We need the untaggeed elements to produce $\delta_D(p, a)$. Let's define the process that determines $\delta_D(p, a)$. We will define six functionals: $B_{p,i} : \mathcal{P}(Q \times \mathcal{P}(Q \cup \bar{Q})) \to \mathcal{P}(Q \times \mathcal{P}(Q \cup \bar{Q}))$, for $1 \leq i \leq 6$. For $X \in \mathcal{P}(Q \times \mathcal{P}(Q \cup \bar{Q}))$:

- $B_{p,1}(X) = \{(q, \{q\})|q \in Q\}$
- $B_{p,2}(X) = \{(q, C)|(q, A) \in X, A \subseteq C\} \cup \{(q, B \cup C)|(q, B \cup \{p\}), (p, C) \in X\}$.
- $B_{p,3}(X) = \{(q, \{q_1, q_2, \ldots, q_m\})|q$ is universal and $\delta(q, a) = \{(q_1, S), (q_2, S), \ldots, (q_m, S)\}\}$.
- $B_{p,4}(X) = \{(q, \{q'\})|q$ is existential and $(q', S) \in \delta(q, a)\}$.
- $B_{p,5}(X) = \{(q, \{\bar{q'}\})|\delta(q, a) = \{(q', R)\}\}$.
- $B_{p,6}(X) = \{(q, A)|\delta(q, a) = \{(q', L)\}, (q', \bar{A}) \in p, A \subseteq Q\}$.

The basic intuition behind this construction is this: With functionals $B_{p,1}, B_{p,3}, B_{p,4}$ we add to set X all the limbs of trivial configuration trees of machine M, that can be created while the head of the machine remains stationary over the symbol a. In more detail: $B_{p,1}$ adds the trivial limbs of trees that consist of one node, $B_{p,3}$ adds the limbs of trees that consist of only a root and the leaves, for every universal state and its transitions on a, and $B_{p,4}$ adds the limbs of trees that consist of only a root and a leave for every existential state and each of its transitions on a. Functional $B_{p,6}$ adds to set X all the limbs of trees of configurations of M that can be created if we combine the roots of the limbs of p with the nodes-configurations of M while the head is over symbol a. Functional $B_{p,2}$ adds to X all the limbs of trees that can be created by the combination of the already created trees of $B_{p,1}, B_{p,3}, B_{p,4}, B_{p,6}$ and $B_{p,2}$ it self, by replacing the leaves of limbs with roots of other limbs. It also adds new limbs with expanded set of leaves. Finally, functional $B_{p,5}$ is the one responsible for adding to X the limbs with tagged elements. For every tree in X it adds a tree with leaves configuration over the next symbol of a, if there is the equivalent transition on δ.

Now we can define $B_p(X) : \mathcal{P}(Q \times \mathcal{P}(Q \cup \bar{Q})) \to \mathcal{P}(Q \times \mathcal{P}(Q \cup \bar{Q}))$ such that:

$$B_p(X) = B_{p,1}(X) \cup B_{p,2}(X) \cup B_{p,3}(X) \cup B_{p,4}(X) \cup B_{p,5}(X) \cup B_{p,6}(X) \cup X$$

There is a sequence created by this function over sets: $\emptyset, B_t(\emptyset), B_p(B_p(\emptyset)), \ldots$ It is obvious that for every $X \in P(Q \times \mathcal{P}(Q \cup \bar{Q}))$, we have $X \subseteq B_p(X)$. It is also clear that $P(Q \times \mathcal{P}(Q \cup \bar{Q}))$ is a finite set. Therefore, the ascending sequence created by this function has a least fixed point X_{fx}, such that $X_{fx} = B_p(X_{fx})$. So, now we can define

$$\delta_D(p, a) = \{(q, A) \in X_{fx}|A \subseteq \bar{Q}\}.$$

In other words, we keep only the elements of the fixpoint that have the tagged leaves. There are two special cases: $\delta_D(\vdash) = \delta_D(\emptyset, \vdash)$ is the starting case. This is because we suppose the machine starts with the head over the first symbol of the output, to the right of $\vdash$. So the first state is all the possble trees of computation the machine can have when it was read only $\vdash$. This includes trivial trees of one state. $\delta_D(q, \dashv)$ is the same as any symbol, with the difference that this time $(q, A) \in \delta_D(q, \dashv)$ means that the set A is the set of leaves that go right after $\vdash$, where the computation halts. As we seen above, if $q = q_s$ and $A = F$ then the tree of configurations has root the starting configuration and leaves the accepting configuration. In that case the machine accepts the input. This concludes the description of δ_D.

Hence, D accepts a string w if $\delta_D(s_0, \vdash w \dashv)$, namely if there is a computation tree of M, with root the starting state q_1, on $\dashv$ and leaves the accepting states on $\dashv$. This is exactly the definition of acceptance for M. □

Chapter 2

Complexity

2.1 Problems

This chapter presents the computational complexity classes related to the computational models of the previous chapter. But what does complexity mean for a finite automaton? In a Turing machine, computational complexity measures the resources that the machine uses in solving a problem, mainly the time, meaning the number of steps the machine makes before halting, and space, namely the number of cells the machine uses on its tape. In a finite automaton the main resource of study is its size, namely its number of states. But in order to relate the number of states of an automaton to a problem, we need to redefine problems.

Definition 2.1. A *problem* is an infinite sequence of languages $(L_h)_{h\geq 1} = (L_1, L_2, L_3, \dots)$.

Example 5. In the problem RETROCOUNT = $(\text{RETROCOUNT}_h)_{h\geq 1}$, we are given a binary string of arbitrary length and we are asked to check that the h-th rightmost digit is 1. Formally, for every $h \geq 1$ the input alphabet is $\Sigma = \{0, 1\}$ and

$$\text{RETROCOUNT}_h = \{w \in \Sigma^* | \text{ the } h\text{-th symbol from the end of } w \text{ is the symbol } 1\}.$$

Example 6. In the problem INCLUSION = $(\text{INCLUSION}_h)_{h\geq 1}$, we are given two subsets of $\{1, 2, \dots, h\}$ and we are asked to check that the first one is included in the second. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \mathcal{P}(\{1, 2, \dots, h\})$ and

$$\text{INCLUSION}_h = \{w \in \Sigma_h^* | w = ab \text{ where } a, b \in \Sigma_h \text{ and } a \subseteq b\}.$$

Example 7. In the problem LENGTH = $(\text{LENGTH}_h)_{h\geq 1}$, we are given a unary string and we are asked to check that its length is h. Formally, for every $h \geq 1$ the input alphabet is $\Sigma = \{0\}$ and

$$\text{LENGTH}_h = \{0^h\}.$$

Example 8. In the problem LONGLENGTH = $(\text{LONGLENGTH}_h)_{h\geq 1}$, we are given a unary string and we are asked to check that its length is 2^h. Formally, for every $h \geq 1$ the input alphabet is $\Sigma = \{0\}$ and

$$\text{LONGLENGTH}_h = \{0^{2^h}\}.$$

2.2 One-way machines

Definition 2.2. We call 1D the class of problems that can be solved by sequences of 1DFAs with polynomialy many states. In other words,

$$\text{1D} = \{(L_h)_{h\geq 1} | \exists \text{ polynomial } p \text{ and sequence of 1DFAs } (M_h)_{h\geq 1} \text{ such that } L(M_h) = L_h \text{ and } |Q_{M_h}| \leq p(h), \text{ for all } h\}.$$

In a similar fashion, we define 1N and 1A as the classes of problems that can be solved by sequences of 1NFAs and 1AFAs, respectively, of polynomial size.

Definition 2.3. We call 2^{1D} the class of problems that can be solved by sequences of 1DFAs with exponentially many states. In other words,

$$2^{\text{1D}} = \{(L_h)_{h\geq 1} | \exists \text{ polynomial } p \text{ and sequence of 1DFAs } (M_h)_{h\geq 1} \text{ such that } L(M_h) = L_h \text{ and } |Q_{M_h}| \leq 2^{p(h)}, \text{ for all } h\}.$$

Proposition 1. LENGTH $\in$1D.

Proof. A 1DFA can count a word of length h with $h+1$ states. Specifically, $D_h = (\{q_0, q_1, \ldots, q_h\}, \{0\}, q_0, \{q_h\}, \delta)$, and $\delta(q_i, 0) = q_{i+1}$ for all $i = 0, 1, \ldots, h-1$. This machine counts the h first 0s in the input word, then it halts. If the input has length more or less than h then the machine does not accept. Hence, D_h solves LENGTH_h with $h+1$ states, for all h. □

Proposition 2. LONGLENGTH $\in 2^{\text{1D}}$.

Proof. A 1DFA can count a word of length 2^h with 2^h+1 states. Similarly to Proposition 1, $D_h = (\{q_0, q_1, \ldots, q_{2^h}\}, \{0\}, q_0, \{q_{2^h}\}, \delta)$, and $\delta(q_i, 0) = q_{i+1}$ for all $i = 0, 1, \ldots, 2^h - 1$. This machine counts the 2^h first 0s in the input word, then it halts. If the input has length more or less than 2^h then the machine does not accept. Hence, D_h solves LONGLENGTH_h with $2^h + 1$ states, for all h. □

Proposition 3. RETROCOUNT $\notin$1D.

Proof. By contradiction. This problem RETROCOUNT_h cannot be solved by a 1DFA with $poly(h)$ many states. Intuitively, in order to know what the h-th rightmost symbol is, the machine needs to remember each one of the last h digits at all times. In order to remember those digits, it needs 2^h states. Formally, we use the pigeonhole principle: If the machine has fewer than 2^h states, then there are two different strings u_1, u_2, of length h, that work as the part of the input that has already been read, and they both lead to the same state q. These two string are different on at least one position. The rightmost position they differ is of interest to us. We call it the i-th position. We append to u_1, u_2 an arbitrary string u of lenth $h - i$, creating strings $w_1 = u_1 u$ and $w_2 = u_2 u$. When the machine reads the two new strings, the h-th rightmost symbols are the i-th rightmost symbols of u_1, u_2 concatenated with u. The $h-1$ rightmost symbols are the same in both strings w_1, w_2 and reading the h-th rightmost symbol the machine moves to the same state q for both strings. So, on both strings the 1DFA has the same computation after the h-th rightmost symbol, and finishes in the same state. But this state is either accepting or not. Either way, one of the strings has to be accepted and the other has to be rejected, so the same state is acceping and rejecting at the same time. That is clearly false. □

Proposition 4. RETROCOUNT $\in$1N.

Proof. In Example 2, we constructed a 1NFA that recognises RETROCOUNT_4. If we generalize that construction, we can see that every RETROCOUNT_h can be recognised by a $(h+1)$-state 1NFA. □

Proposition 5. INCLUSION $\notin$1N.

Proof. By contradiction. Suppose there is a 1NFA N that recognises INCLUSION_h with $poly(h)$ states. Let $a_1, a_2, \ldots, a_{2^h}$ be a list of all symbols in $\mathcal{P}(\{1, 2, \ldots, h\})$. Then N accepts all inputs of the form $a_i a_i$. So every such input has an accepting path of length 2, of the form $q_0 \to q_i \to q_f$, where q_0 the starting state and q_f an accepting state. When the machine moves to state q_i it has already read the first symbol, and the head is over the second symbol of the input. We do know that there are 2^h symbols a_i, but only $poly(h)$ states q_i. By the pigeonhole principle, this means, that for large enough h there are two first symbols a_i and a_j, where $a_i \neq a_j$, that produce the same middle state $q_i = q_c = q_j$. This means that $q_c \in \delta(q_0, a_i)$ and $q_c \in \delta(q_0, a_j)$ and also $\delta(q_c, a_i) \cap F \neq \emptyset \neq \delta(q_c, a_j) \cap F$. From these two facts it is obvious that $a_i a_j, a_j a_i \in \text{INCLUSION}_h$, but this is not possible since the two sets are not equal. □

Proposition 6. INCLUSION $\in$1A.

Proof. In Example 3, we constructed a 1AFA that recognises INCLUSION_4. If we generalize that construction, we can see that every INCLUSION_h can be recognised by a $(h+3)$-state 1AFA. □

We have shown the computability equivalence of the one-way models, but not the detailed 1NFA-1DFA equivalence. The subset construction is a typical way to construct a 1DFA that recognises the same language as a given 1NFA.

Theorem 3. *For every k-state 1NFA there is an equivalent 2^k-state 1DFA.*

Proof. Let $N = (Q, \Sigma, q_1, F, \delta)$ be a 1NFA. There is a 1DFA $D = (Q_D, \Sigma, \{q_1\}, F_D, \delta_D)$ such that $L(D) = L(N)$, where

- Q_D is the powerset of Q.
- F_D consists of all $X \in \mathcal{P}(Q)$ such that $F \cap X \neq \emptyset$.
- δ_D is the transition function, such that $\delta_D(S, a) = \delta(S, a)$ according to the extension of δ, as defined in Subsection 1.2.1.

It can easily be proven by induction (on the length of w) that, for all $w \in \Sigma^*$: $\delta(q_1, w) = \delta_D(\{q_1\}, w)$. This means that $\delta(q_1, w) \cap F \neq \emptyset \iff \delta_D(\{q_1\}, w) \in F_D$, so $w \in L(N) \iff w \in L(D)$. □

We need to note that the set of states Q_D is the powerset of Q and thus D has exponentially more states. Many of these states might not be reachable, so they may be ignored, but this is the general way to construct an equivalent 1DFA. The above proof is about the computablity equivalence of the two models, but also provides an algorithm to create a 1DFA that is equivalent to the given 1NFA.

Corollary 2. $1\text{N} \subseteq 2^{1\text{D}}$

The natural question that follows is: What is the relation between 1N and 1D?

Theorem 4. 1D$\subsetneq$1N

Proof. We need to show two things. First that 1D$\subseteq$1N and second that 1N$\backslash$1D$\neq \emptyset$.
1D$\subseteq$1N: It is immediate from the definition of the 1NFA and the 1DFA that the former is a generalization of the later. A family of 1DFAs with polynomialy many states is in fact a family of 1NFAs with polynomialy many states. So 1D$\subseteq$1N.
RETROCOUNT $\notin$1D. This has been proven in Proposition 3. This means that there can be no 1DFA with polynomialy many states (with respect to h) that recognises the mentioned problem
RETROCOUNT $\in$1N. As we have seen in Proposition 4, this problem is in 1N.

As a result the exponential blow up of the subset construction is an inescapable barrier. And so, there is a computational gap between 1D and 1N. □

2.3 Two-way machines

Classes analogous to the ones defined above for one-way machines can be defined for two-way machines, as well.

Definition 2.4. We call 2D the class of problems that can be solved by sequences of 2DFAs with polynomially many states. In other words,

$$\begin{aligned} \text{2D} = \{(L_h)_{h\geq 1} | \exists \text{ polynomial } p \text{ and sequence of 2DFAs } (M_h)_{h\geq 1} \\ \text{such that } L(M_h) = L_h \text{ and } |Q_{M_h}| \leq p(h), \text{ for all } h \ \}. \end{aligned}$$

In a similar fashion, we call 2N and 2A the classes of problems that can be solved by polynomial-size sequences of 2NFAs and 2AFAs, respectively.

Proposition 7. INCLUSION $\in$2D.

Proof. We have proven in Example 4 that INCLUSION$_4$ can be solved by a 2DFA. That construction can be generalized to prove that INCLUSION$_h$ can be recognised by a $(2h+3)$-state 2DFA, for every $h \geq 1$. The description is as follows: $D = (Q, \mathcal{P}(\{1, 2, \ldots, h\}), q_1, \{q_f\}, \delta)$, where $Q = \{q_{h+1}, q_{h+2}, q_f\} \cup \bigcup_{1\leq i\leq h}\{q_i, q_i'\}$ and δ is defined as follows:

For $1 \leq i \leq h$, $\delta(q_i, a) = \begin{cases} (q_i', R) & \text{if } i \in a \\ (q_{i+1}, S) & \text{if } i \notin a \end{cases}$

For $1 \leq i \leq h$, $\delta(q_i', a) = \begin{cases} (q_{i+1}, L) & \text{if } i \in a \\ \text{undefined} & \text{if } i \notin a \end{cases}$

$\delta(q_{h+1}, a) = (q_{h+2}, R)$
$\delta(q_{h+2}, a) = (q_f, R)$
$\delta(q_f, \dashv) = (q_f, R)$
The are no transitions other than the ones described. □

Proposition 8. LONGLENGTH $\notin$2N.

Proof. By contradiction. Suppose there is a $poly(h)$-state 2NFA N that recognises LONGLENGTH$_h$. This means that the machine accepts only the word 0^{2^h}. So, on input 0^{2^h}, there is at least one configuration path from the starting configuration to an accepting configuration. For the following proof we take one of these paths into consideration. We

may suppose that there are no cycles in this path (meaning that no configuration appears more than once) since, in case there are cycle, we may remove them and the remaining path will still be accepting.

There are three possible symbols the head may be reading: $\vdash$, 0 and $\dashv$. Suppose N has $k = poly(h)$ states. Let's also suppose N starts its computation from the right end-marker. In the mentioned computation path, the machine may visit the left end-marker at most k times. (If it visits it more than k times, then the computation path visits the same configuration more than once. We already eliminated this possibility.) This means that the machine traverses the whole input at most $k+1$ times (moving from one end-marker to the other). Each time, the machine traverses the 2^h 0s, while having only $poly(h)$ states. By the pigeonhope principle, and for large enough h, there has to be a state that appears more than once in the same traversal. In that case the machine, while moving either towards the left or the right, is at state q and is reading symbol 0, more than once. The interval of 0s between the two earliest such occurrences, is now called a segment. Suppose the length of the segment in the first traversal is l_1. So, this 0^{l_1} segment may be repeated any number of times, and the machine will not notice the difference. For each traversal, we have at least one segment. So, we have at most $k+1$ segments, if we choose one from each traversal. Suppose we have $m \leq k+1$ segments. Then we define the product of the lengths of all those segments $p = l_1 l_2 \cdots l_m$. Machine N accepts 0^{2^h+p}. This is because p is $l_i \cdot p_i$ for some p_i, a multiple of each segments length, so it can be added to the word and the machine will not notice in any of its traversals. So N accepts a word that is not 0^{2^h}. We have reached a contradiction. □

Unfortunately, the separating lines between the classes 2D and 2N are not as clear as for the corresponding classes of one-way automata. We do not know whether 2D and 2N are equal or not. This is a difficult question to answer. Its structure resembles the structure of the P versus NP question, so a similar approach is through complete problems.

2.4 Reductions

A reduction of a problem A to a problem B is a systematic way of transforming the instances of A so that they can be solved as instances of B. In that case, problem A is at most as hard as B, since a machine that solves B can solve A as well, with a small transformation. The machine that transforms the instances of A into instances of B is called a transducer. But in order for a transducer to show that a problem is at most as hard as another, the transformation taking place has to be "easy". The transducer shouldnt be strong enough to solve A on its own. It has to be of limited size and/or head movement. The first transducer that we will define is the one-way deterministic finite transducer, or 1DFT. Like all transducers, 1DFTs are machines with two tapes, one for reading the input and one for printing the output. The machine does not have the ability to write on the first tape or read the second one.

2.4.1 One-way deterministic finite transducer

Definition 2.5. A *one-way deterministic finite transducer* (1DFT) is a five-tuple,

$$T = (Q, \Sigma_A, \Sigma_B, q_1, q_f, \delta)$$

where

- Q is the set of states,

- Σ_A is a finite input alphabet,
- Σ_B is a finite output alphabet,
- $q_1 \in Q$ is the starting state,
- $q_f \in Q$ is the final state, and
- $\delta : (Q \times \Sigma_A) \to (Q \times \Sigma_B^*)$ is the transition function.

The second coordinate of $\delta(\cdot, \cdot)$ dictates what will be printed on the second tape. The two alphabets may have no relation to each other.

Definition 2.6. Let $L = (L_h)_{h\geq 1}$ and $L' = (L'_h)_{h\geq 1}$ be problems We write $L \leq_{1\mathrm{D}} L'$ and say that L *reduces to* L' *in one-way polynomial size*, if there is a sequence of 1DFTs $(T_h)_{h\geq 1}$ and two polynomials e and s such that every T_h has $\leq s(h)$ states and maps instances of L_h to instances of $L'_{e(h)}$ so that for all x:

$$x \in L_h \Rightarrow T_h(x) \in L'_{e(h)} \text{ and } x \in L_h^c \Rightarrow T_h(x) \in (L'_{e(h)})^c.$$

A special case of a one-way polynomial size reduction called homomorphic reduction, occurs when the 1DFTs have only one state, namely when $s(h) = 1$, for all h.

Two major properties essential to reductions are transitivity and closure.

Transitivity is a property that seems natural when we think of what a reduction means in an abstract way. If L is at most as hard as L', and L' is at most as hard as L'', then surely L should be at most as hard as L''.

Theorem 5. *$L \leq_{1\mathrm{D}} L'$ and $L' \leq_{1\mathrm{D}} L'' \Rightarrow L \leq_{1\mathrm{D}} L''$, for any three problems L, L', L''.*

Proof. For a fixed $h \geq 1$, by $L \leq_{1\mathrm{D}} L'$, there is a 1DFT T with $s(h)$ states, that transforms the instances of L_h to instdances of $L'_{e(h)}$ for some polynomials s, e. Similarly $L' \leq_{1\mathrm{D}} L''$ implies that there is a 1DFT T' with $s'(e(h))$ states, that transforms the instances of $L'_{e(h)}$ to instances of $L''_{e'(e(h))}$.

Let $T = (Q, \Sigma_A, \Sigma_B, q_1, q_f, \delta)$ and $T' = (Q', \Sigma_B, \Sigma_C, q'_1, q'_f, \delta')$. We define a 1DFT $T_c = (Q \times Q', \Sigma_A, \Sigma_C, (q_1, q'_1), (q_f, q'_f), \delta_c)$, which simulates both T and T'. We define δ_c depending on δ and δ':

$$\delta_c((q, q'), a) = ((p, p'), w'), \text{ where } \delta(q, a) = (p, w) \text{ and } \delta'(q', w) = (p', w').$$

This machine transforms an instance of L in to an instance of L''. We can verify this by monitoring the movement of the second coordinate state. The second coordinate state moves the way T' moves, producing the same output, only slower, since for every movement it makes in the second coordinate, it has already made several steps in the first coordinate which simulates the process of T. So the basic idea is that we have two different coordinates to simulate the two different 1DFTs we are combining. The resulting machine T_c has $s(h) \cdot s'(e(h))$ states, and the output is an instance of $L''_{e'(e(h))}$. □

Closure is a similar to transitivity, in abstract way. If a problem is at most as hard as another problem in a complexity class, then the first problem should be in that complexity class as well.

Theorem 6. *$L \leq_{1\mathrm{D}} L'$ and $L' \in 1\mathrm{D} \Rightarrow L \in 1\mathrm{D}$, for any two problems L, L'.*

Proof. For a fixed $h \geq 1$, $L \leq_{1\mathrm{D}} L'$ means that there is a 1DFT T with s(h) states, that transforms the instances of L_h to instdances of $L'_{e(h)}$, for e, s polynomials. $L' \in$1D means that there is a 1DFA M and s' a polynomial, such that the number of states of M is bounded by s'(h). The proof is the same in technical terms as the one for transitivity. We create a 1DFA $M_c = (Q \times Q', \Sigma_A, (q_1, q'_1), F_c, \delta_c)$, combining $T = (Q, \Sigma_A, \Sigma_B, q_1, q_f, \delta)$ and $M = (Q', \Sigma_B, q'_1, F, \delta')$ the same way we combined T and T' previously. The only thing to be defined is the set of accepting states and δ_c:

$$F_c = \{(q_f, p) | p \in F\}$$

and

$$\delta_c((q, q'), a) = (p, p'), \text{ where } \delta(q, a) = (p, w) \text{ and } \delta'(q', w) = p'.$$

□

Example 9. In the problem COMPOSITION = $(\mathrm{COMPOSITION}_h)_{h\geq 1}$ we are given two partial functions $f, g : \{1, 2, \ldots, h\} \rightharpoonup \{1, 2, \ldots, h\}$ and we are asked to check that $f(g(1)) = 1$. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = (\{1, 2, \ldots, h\} \rightharpoonup \{1, 2, \ldots, h\})$ and

$$\mathrm{COMPOSITION}_h = \{fg | f, g \in \Sigma_h \text{ and } f(g(1)) = 1 \ \}.$$

Proposition 9. COMPOSITION $\leq_{1\mathrm{D}}$ RETROCOUNT.

Proof. The 1DFT reads the first symbol f, and prints the symbols $h(f(1)), h(f(2)), \ldots,$ $h(f(h))$ where

$$h(x) = \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{if } x \neq 1. \end{cases}$$

Then it moves to the right and reads the second symbol g and prints $g(1) - 1$ times the symbol 0.

This transormation creates positive instances of $\mathrm{RETROCOUNT}_h$ out of positive instances of $\mathrm{COMPOSITION}_h$ and negative instances of $\mathrm{RETROCOUNT}_h$ out of negative instances of $\mathrm{COMPOSITION}_h$: The question we need to ask is which is the h-th rightmost symbol. The output sequence by the machine is $h(f(1)), h(f(2)), \ldots, h(f(h)), 0, 0,$ $\ldots, 0$. The number of 0s is $g(1) - 1$. So the total length of the sequence is $h + g(1) - 1$. If we want to find the h-th rightmost symbol we have to substract $h - 1$. Therefore the symbol we are looking for is the $g(1)$-th from left. That is $h(f(g(1)))$. So $f(g(1)) = 1$ iff the h-th rightmost symbol is 1. This transducer produces a one-way polynomial reduction: The transducer needs only to print a sequence for a symbol and then a sequence of 0s for the second symbol of the input. So only two states are needed. □

Example 10. In the problem ROLLCALL = $(\mathrm{ROLLCALL}_h)_{h\geq 1}$ we are given a list of numbers from $\{1, 2, \ldots, h\}$ and we are asked to check that every number from that set appears at least once. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \{1, 2, \ldots, h\}$, and

$$\mathrm{ROLLCALL}_h = \{w \in \Sigma_h^* | \text{ every number in } \{1, 2, \ldots, h\} \text{ appears at least once in } w\}.$$

Proposition 10. INCLUSION $\leq_{1\mathrm{D}}$ ROLLCALL.

Proof. There is a transducer that transforms the instances of INCLUSION_h to instances of ROLLCALL_h, with the following process. The transducer reads the first symbol a of the input and then prints all the elements of the set $a_c = \{1, 2, \ldots, h\} \setminus a$, in ascending order. Then, the transducer moves the reading head to the right, reads the second symbol b and prints all the elements of b, in ascending order again.This transormation creates accepting instances of ROLLCALL_h out of accepting instances of INCLUSION_h and negative instances of ROLLCALL_h out of negative instances of INCLUSION_h: Since $a, b \in \{1, 2, \ldots, h\}$, then $a \subseteq b$ iff $a^c \cup b = \{1, 2, \ldots, h\}$. The elements printed on the output tape are those of a^c and b. If $a \nsubseteq b$, then there is a x such that $x \notin b$ and $x \in a$. So, $x \notin b$ and $x \notin a^c$. This means that x was not printed in any of the two strings. This transduced produces a one-way polynomial reduction: This tranducer reads the first symbol and prints a string, then moves to the second state, reads the second symbol and prints another string. This means the transducer needs only two states. □

2.4.2 One-way nondeterministic finite transducer

There are several other types of transducers that define other types of reductions.

Definition 2.7. A *one-way nondeterministic finite transducer* (1NFT) is a six-tuple,

$$T = (Q, \Sigma_A, \Sigma_B, q_1, q_f, \delta)$$

where $Q, \Sigma_A, \Sigma_B, q_1, q_f$ are the same as for 1DFTs and $\delta : (Q \times \Sigma_A) \to \mathcal{P}(Q \times \Sigma_B^*)$ is the transition function.

We say that T transforms an instance $x \in \Sigma_A^*$ into a string $T(x) \in \Sigma_B^*$, if all computations that end at the accepting state output the same string $T(x)$. Otherwise T is not defined on x.

Definition 2.8. Let $L = (L_h)_{h\geq 1}$ and $L' = (L'_h)_{h\geq 1}$ be problems We write $L \leq_{1\text{N}} L'$ and say that L *reduces to* L' *in nondeterministic one-way polynomial size*, if there is a sequence of 1NFTs $(T_h)_{h\geq 1}$ and two polynomials e and s such that every T_h has $s(h)$ states and maps instances of L_h to instances of $L'_{e(h)}$ so that for all x:

$$x \in L_h \Rightarrow T_h(x) \in L'_{e(h)} \text{ and } x \in L_h^c \Rightarrow T_h(x) \in (L'_{e(h)})^c \text{ or } T_h(x) \text{ is undefined.}$$

Lemma 3. *$L \leq_{1\text{N}} L'$ and $L' \leq_{1\text{N}} L'' \Rightarrow L \leq_{1\text{N}} L''$, for any three problems L, L', L''.*

Proof. Nondeterministic one-way polynomial-size reductions are transitive and the proof of transitivity uses the same construction as the corresponding proof for $\leq_{1\text{D}}$.

Let T, T' and T_c be the same as in the case of transitivity for $\leq_{1\text{D}}$. The transducer T_c produces the same output for each input and for every possible path that ends in (q_f, q'_f). This is because, since its simulation of T has reached q_f as the final state, then the simulated output string is same for all possible paths of computation. So the input of the simulated T' is the same for all paths, and the simulation of the second machine reaches q'_f, the accepting state, then the output of T' is the same for all accepting computation paths. This proves that the output of T_c is the same for all accepting paths. So T_c as defined in Theorem 5, satisfies the requirements to reduce L to L''.

□

Lemma 4. *$L \leq_{1\text{N}} L'$ and $L' \in 1\text{N} \Rightarrow L \in 1\text{N}$, for any two problems L, L'.*

Proof. 1N is closed under $\leq_{1N}$, by the same construction as in the case for $\leq_{1D}$. The proof is similar to that of transitivity. □

2.4.3 Two-way deterministic finite transducer

Definition 2.9. A *two-way deterministic finite transducer* (2DFT) is a six-tuple,

$$T = (Q, \Sigma_A, \Sigma_B, q_1, q_f, \delta)$$

where $Q, \Sigma_A, \Sigma_B, q_1, q_f$ are the same asfor 1DFTs and $\delta : (Q \times (\Sigma_A \cup \{\vdash, \dashv\})) \to (Q \times \Sigma_B^* \times \{L, S, R\})$ is the transition function.

Similarly to two-way finite automata we use $\vdash$ and $\dashv$ to mark the start and end of the input respectively, since the head of the input tape will be moving in both directions. The head on the output tape works in the same way as all previous transducers worked, meaning it can move only to the right. For an output to be considered valid, T has to finish its computation in the accepting state. As in Chapter 1, we assume that δ cannot move to the left of the left end-marker, or to the right of the right end-marker, unless it is moving right to a final state.

Definition 2.10. Let $L = (L_h)_{h\geq 1}$ and $L' = (L'_h)_{h\geq 1}$ be problems. We write $L \leq_{2D} L'$ and say that L *reduces to* L' *in two-way polynomial-size*, if there is a sequence of 2DFTs $(T_h)_{h\geq 1}$ and two polynomials e and s such that every T_h has $s(h)$ states and maps instances of L_h to instances of $L'_{e(h)}$ so that for all x:

$$x \in L_h \Rightarrow T_h(x) \in L'_{e(h)} \text{ and } x \in L_h^c \Rightarrow T_h(x) \in (L'_{e(h)})^c \text{ or } T_h(x) \text{ is undefined.}$$

Two-way polynomial-size reductions may not be transitive. The reason is that when we try to simulate the two machines by a new one, the intermediate tape needs to be accessible in a two-way fashion, meaning any cell might be revisited. This creates a problem for the simulating transducer, since it cannot remember the content of the intermediate tape, since the alphabet Σ_B may be exponentially larger than h. As a result, the machine cannot remember the spot of the intermediate tape, since the intermediate string is of arbitrary length. We will define a restriction of the above reduction, one that will have the property of transitivity.

Definition 2.11. We call a sequence of 2DFTs $T = (T_h)_{h\geq 1}$ *laconic* if every T_h performs $\leq p(h)$ printing steps on each input, where p is polynomial.

Definition 2.12. Let $L = (L_h)_{h\geq 1}$ and $L' = (L'_h)_{h\geq 1}$ be problems. We write $L \leq_{2D}^{lac} L'$ and say that L *reduces to* L' *in two-way polynomial-size/print*, if there is a laconic sequence of 2DFTs $(T_h)_{h\geq 1}$ and two polynomials e and s such that every T_h has $s(h)$ states and maps instances of L_h to instances of $L'_{e(h)}$ so that for all x:

$$x \in L_h \Rightarrow T_h(x) \in L'_{e(h)} \text{ and } x \notin L_h \Rightarrow T_h(x) \notin (L'_{e(h)}).$$

Lemma 5. *$L \leq_{2D}^{lac} L'$ and $L' \leq_{2D}^{lac} L'' \Rightarrow L \leq_{2D}^{lac} L''$, for any three problems L, L', L''.*

Proof. For a fixed $h \geq 1$, by $L \leq_{2D}^{lac} L'$, there is a laconic 2DFT T with $s(h)$ states, that transforms the instances of L_h to instdances of $L'_{e(h)}$ and has printing steps bounded by $p(h)$, for some polynomials s, e, p. Similarly $L' \leq_{2D}^{lac} L''$ implies that there is a laconic 2DFT

T' with $s'(e(h))$ states, that transforms the instances of $L'_{e(h)}$ to instances of $L''_{e'(e(h))}$ and has printing steps bounded by $p'(h)$, for some polynomials s', e', p'.

The proof is similar to the initial proof of transitivity, with the addition of two counters of size $p(h)$. The counters keeps in the "memory" of the machine the position of the output head of the first 2DFT and the position of the reading head of the second 2DFT that is being simulated. Every time the second machine is moving its "hypothetical" head to the left, the composing new machine changes the counter, and simulates the first machine from the start. At some point the simulation of the first machine "prints" the cell the second machine is expecting (when the two counters have the same value), and then the second 2DFT being simulated, reads the symbol it was waiting for, and continues its deterministic computation. The number of states needed stays polynomial, since the counter needs only be polynomially large.

Let $T = (Q, \Sigma_A, \Sigma_B, q_1, q_f, \delta)$ and $T' = (Q', \Sigma_B, \Sigma_C, q'_1, q'_f, \delta')$. We define a 1DFT $T_c = ((Q \cup \{q_0\}) \times Q', \Sigma_A, \Sigma_C, (q_1, q'_1), (q_f, q'_f), \delta_c)$, which simulates both T and T'. We define δ_c depending on δ and δ':

$\delta_c((q, q', l_1, l_2), a) = ((p, p', l_1, l_2), w', X)$, where $\delta(q, a) = (p, w, X)$ and $\delta'(q', w) = (p', w')$.

$$\delta_c((q, q', l_1, l_2), a) = \begin{cases} ((q_0, q', 1, l_2), \epsilon_C, L) & \text{if } q = q_0 \text{ and } a \neq \vdash \\ ((q_1, q', 1, l_2), \epsilon_C, R) & \text{if } q = q_0 \text{ and } a = \vdash \\ ((p, q', l_1^*, l_2), \epsilon_C, X) & \text{if } l_1 < l_2 \text{ and } q \neq q_0 \\ ((p, p', l_1^*, l_2^*), w', X) & \text{if } l_1 = l_2 \text{ and } q \neq q_0 \\ ((q_0, q', 1, l_2), \epsilon_C, X) & \text{if } l_1 > l_2 \text{ and } q \neq q_0 \end{cases}$$

where $\delta(q, a) = (p, w, X)$ and $\delta'(q', w) = (p', w', Y)$, $l_1^* = l_1 + |w|$ and

$$l_2^* = \begin{cases} l_2 - 1 & \text{if } Y = L \\ l_2 & \text{if } Y = S \\ l_2 + 1 & \text{if } Y = R. \end{cases}$$

Thus the number of states needed is $O(p(h)^2 s(h) s'(e(h)))$. □

Lemma 6. *$L \leq_{2D}^{lac} L'$ and $L' \in 2N \Rightarrow L \in 2N$, for any two problems L, L'.*

Proof. Closure can be proven the same way, using the fact that the printing steps of the transducer is bounded by a polynomial. In the same way we can combine a 2DFT with bounded printing steps, with a 2DFA (Same proof as above, cartesian product of states, the second coordinate makes a move whenever the first simulated machine "prints" the cell that was on demand etc).

We create a 2DFA $M_c = ((Q \cup \{q_0\}) \times Q', \Sigma_A, (q_1, q'_1), F_c, \delta_c)$, combining $T = (Q, \Sigma_A, \Sigma_B, q_1, q_f, \delta)$ and $M = (Q', \Sigma_B, q'_1, F, \delta')$ the same way we combined T and T' previously. The only thing to be defined is the set of accepting states and δ_c:

$$F_c = \{(q_f, p) | p \in F\}$$

and

$$\delta_c((q, q', l_1, l_2), a) = \begin{cases} ((q_0, q', 1, l_2), L) & \text{if } q = q_0 \text{ and } a \neq \vdash \\ ((q_1, q', 1, l_2), R) & \text{if } q = q_0 \text{ and } a = \vdash \\ ((p, q', l_1^*, l_2), X) & \text{if } l_1 < l_2 \text{ and } q \neq q_0 \\ ((p, p', l_1^*, l_2^*), X) & \text{if } l_1 = l_2 \text{ and } q \neq q_0 \\ ((q_0, q', 1, l_2), X) & \text{if } l_1 > l_2 \text{ and } q \neq q_0 \end{cases}$$

where $\delta(q,a) = (p,w,X)$ and $\delta'(q',w) = (p',Y)$, $l_1^* = l_1 + |w|$ and

$$l_2^* = \begin{cases} l_2 - 1 & \text{if } Y = L \\ l_2 & \text{if } Y = S \\ l_2 + 1 & \text{if } Y = R. \end{cases}$$

This means that 2D is closed under $\leq_{\text{2D}}^{\text{lac}}$. □

2.5 Completeness

First we need to define a problem.

Example 11. In the problem *One-Way Liveness*, OWL = $(\text{OWL}_h)_{h\geq 1}$, the alphabet is the set of all h-tall, two-column, directed graphs, with arrows from the left column to the right column. A string w is seen as an h-tall, $(|w|+1)$-column, directed graph, with arrows from the nodes of each column to the ones of its adjacent one to the right. We are asked to check that there is a path from the leftmost column to the rightmost column. That path is called *live*. When a graph has a live path, we say it is *live*. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \mathcal{P}(\{1,2,\ldots,h\}^2)$ and

$$\text{OWL}_h = \{w \in \Sigma_h^* |\ w \text{ has a live path}\}.$$

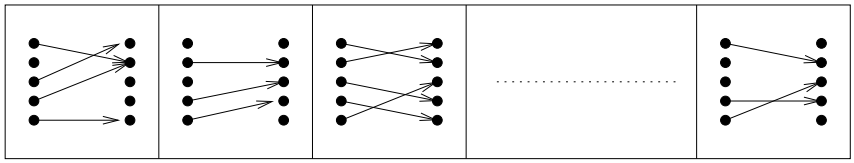

Figure 2.1: A string w of Σ_h^*.

Proposition 11. OWL $\in$1N.

Proof. We will construct an $(h+1)$-state 1NFA N that recognises OWL_h. Let 1NFA $N = (Q, \Sigma, q_s, F, \delta)$, where

- $Q = \{q_s, q_1, q_2, \ldots, q_h\}$ is the set of states: Every state q_i corresponds the i-th node of every column.
- Σ_h is the alphabet defined above.
- $F = Q$: All states are final.

We now describe δ and the reason why it works.

For q_s: $\delta(q_s, a) = \{q_j |$ there is an arrow $(i \to j) \in a\}$. These are the transitions of the starting state. The intention here is that a is the first symbol of the input. If a node, on the right column of a is reachable, then there is a transition to the state representing that node. In a sense, this first transition ignores the first column. There just has to be an i such that $i \to j \in a$.

For q_i : $\delta(q_i, a) = \{q_j |$ there is an arrow $(i \to j) \in a\}$. These are the transitions of the general case. At state q_i, reading symbol a, if $(i \to j) \in a$ then the machine may transition to state q_j.

It is straightforward to see that an accepting computation represents a path from a node in the first column to a node in the last column, which means that a node in the last colum is reachable from a node in the first column. □

The next problem is a generelization of OWL since each graph is also allowed to have arrows from right to left and within the same column.

Example 12. In the problem *Two-Way Liveness*, TWL = $(\text{TWL}_h)_{h\geq 1}$, the alphabet is the set of all h-tall, two-column, directed graphs. A string w is seen as an h-tall, $(|w|+1)$-column, directed graph, with arrows between the nodes of the same column or of adjacent ones. Again, we are asked to check that there is a path from the leftmost column to the rightmost column. Formally, for every $h \geq 1$ the input alphabet is $\Gamma_h = \mathcal{P}(\{1, 2, \ldots, 2h\}^2)$ and

$$\text{TWL}_h = \{w \in \Gamma_h^* | \ w \text{ has a live path}\}.$$

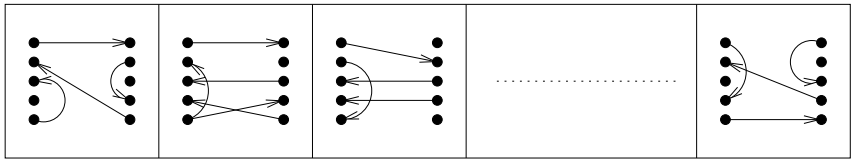

Figure 2.2: A string w of Γ_h^*.

Proposition 12. TWL $\in$2N.

Proof. We will construct a $(2h+1)$-state 2NFA N that recognises TWL_h. We let 1NFA $N = (Q, \Gamma_h, q_s, F, \delta)$, where

- $Q = \{q_s, q_1, q_2, \ldots, q_h, q_{h+1}, q_{h+2}, \ldots, q_{2h}\}$ is the set of states: q_s is the starting state, used only in the first move, $q_1, q_2, \ldots, q_h$ are the states that correspond to the left column of each symbol a and $q_{h+1}, q_{h+2}, \ldots, q_{2h}$ are the states that correspond to the right column.

- $F = \{q_1, q_2, \ldots, q_h\}$: all the left-column states are final states.

We now describe δ and the reason why it works. For $i, j \leq h$ and $a \in \Gamma_h$:

For q_s: $\delta(q_s, a) = \{(q_{h+j}, S) |$ there is an arrow $(i \to (h+j)) \in a\}$. This is similar to the one-way case. From the start state we simply move to the right-column states that are reachable. We have the privilege to ignore the first column. We can completely ignore any arrows of the form $i \to j$ in the first symbol, those thansitions are of no use to us.

For q_{h+i}: $\delta(q_{h+i}, a) = \{(q_u, S) |$ there is an arrow $((h+i) \to u) \in a\} \bigcup \{(q_i, R)\}$, where $1 \leq u \leq 2h$. Whenever the machine is in a right-column state it has two options. It can either change state according to the arrows of the current input symbol without moving the head, or move the head to the right and transition to the corresponding left-column state.

For q_i: $\delta(q_i, a) = \{(q_u, S) |$ there is an arrow $(i \to u) \in a\} \bigcup \{(q_{h+i}, L)\}$, where $1 \leq u \leq 2h$. This is the symmetrical case of the above. The machine has two options at any time its head is on a in state q_i. Either it can follow an arrow from the current symbol a, or it can move its head to the left.

$\delta(q_{h+i}, \vdash) = \emptyset$. This is the case of the machine moving to the left column of the first symbol, and then moving the head to the left to $\vdash$. The path ends here, rejecting the

computation, since there is no information we can further acquire. If there is a live path going through the current configuration, then there is a live path from the starting state as well.

$\delta(q_i, \dashv) = \{(q_i, R)\}$. This is an accepting situation. Following arrows, the machine has reached the right column of the last symbol. There, it has the non-deterministic option to move to the right. Moving to the right, it reads $\dashv$, deducing that the previous symbol was the last one and that there is a path reaching the last column Therefore it moves right and halts in an accepting state.

The above are the only acceptable moves of the machine.

Again, it is straightforward that the machine moves nondeterministically following every possible arrow. If there is a live path, then there is an accepting computation path for N, and vice versa. □

These problems are proven to be complete in 1N and 2N respectively. Here is the proof of the second case. The first case can be deduced easily from the second one.

Lemma 7. *Every problem in* 2N *can be reduced to* TWL

Proof. The basic idea is that a $2h$-tall, two-way, multi-column graph can simulate the transitions of a given h-state 2NFA. Note that in the following proof we will be using 2NFAs that may only move their head either left or right at any step of the computation (the head cannot remain stationary). This is equivalent to the standard 2NFA, since it can be simulated by doubling the states, so for every state, every time the original 2NFAs head has to remain stationary, the head moves to the left (or the right), the machine enters the clone state and returns to the original state and tape cell. We will show how to encode an input $a_1a_2\cdots a_n$ of an h-state 2NFA $N = (Q, \Sigma, q_1, F, \delta)$ where $Q = \{q_1, q_2, \ldots, q_h\}$ into an instance of TWL $g(x) = g(\vdash)g(a_1)g(a_2)\cdots g(a_n)g(\dashv)$, in such way that N accepts the input iff there is a live path in the instance of TWL. We need to define g so that N accepts x iff there is a path from the leftmost to the rightmost column in $g(x)$. Obviously, $g : \Sigma \cup \{\vdash, \dashv\} \to \Gamma_{2h}$ where Γ_{2h} is the set of all $2h$-tall, directed, two-column graphs.

The symols of Γ_{2h} that g uses look like this:

The left nodes are named $l_1, l_2, \ldots, l_h, L_1, L_2, \ldots, L_h$ and the nodes on the right column are named $R_1, R_2, \ldots, R_h, r_1, r_2, \ldots, r_h$. In order to define g, we describe its behavior over $\vdash$, Σ, and $\dashv$.

- $g(\vdash) = \{l_1 \to R_1\} \bigcup \{r_i \to R_j |$ for every $(q_j, R) \in \delta(q_i, \vdash)\}$.

  The first set of the union is for the starting state. It states that the only node from the leftmost column from which there can be a path is the one that corresponds to the starting state. The second set of the union is for the case where the head moves to $\vdash$ and then moves back right to state q_j. This is case (a) of Figure 2.3, where q_1 is the starting state.

- $g(a) = \{l_i \to R_j |$for every $(q_j, R) \in \delta(q_i, a)\} \bigcup \{r_j \to R_j |$for every $(q_j, R) \in \delta(q_i, a)\}$ $\bigcup \{l_i \to L_j |$for every $(q_j, L) \in \delta(q_i, a)\} \bigcup \{r_i \to L_j |$for every $(q_j, L) \in \delta(q_i, a)\}$.

  Whenever $(q_j, R) \in \delta(q_i, a)$, we add two arrows that point to R_j: One from l_i, in case the machine's previous move was from left to right, and one from r_i, in case the machine's previous move was from right to left. Symmetricaly, whenever $(q_j, L) \in \delta(q_i, a)$ we add two arrows that point to L_j: One from l_i, in case the machine's previous move was from left to right, and one from r_i, in case the machines previous move was from right to left. This is case (b) of Figure 2.3.

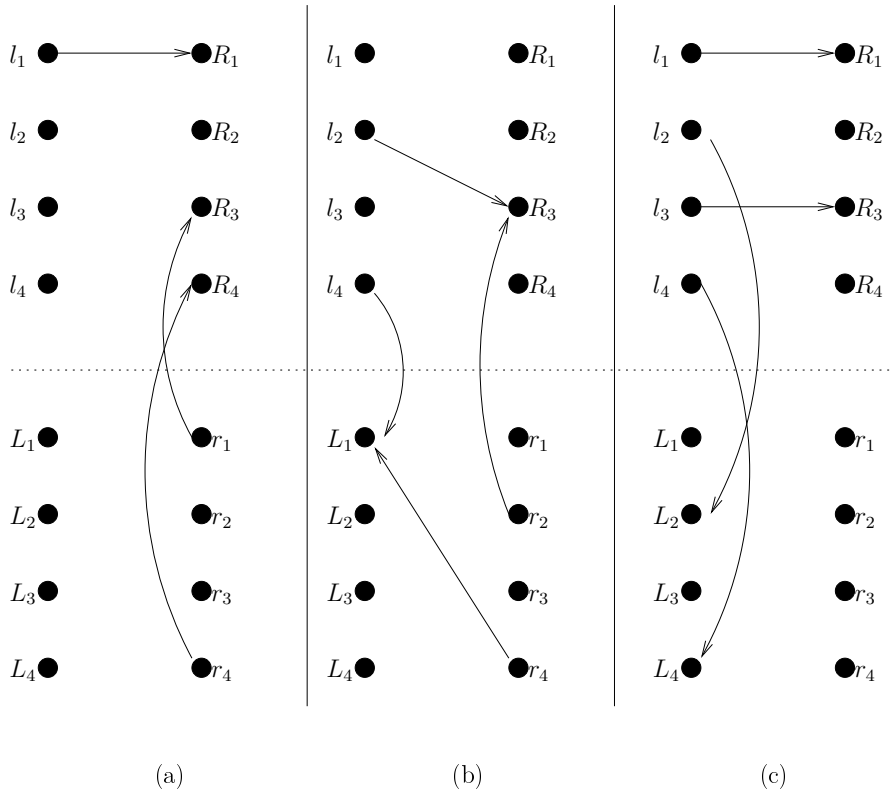

Figure 2.3: The three cases for $g(x)$, in an example where $h = 4$.

- $g(\dashv) = \{l_i \to L_j | \text{for every } (q_j, L) \in \delta(q_i, \dashv)\} \bigcup \{l_i \to R_i | \text{for every } q_i \in F\}$.

  Similarly for the right end-marker, the first set of the union is for the case where the head moves to $\dashv$ and to state q_i, but q_i is not a final state, so it moves back left to state q_j. The second set is for the final states. In order for a path to reach the rightmost column it needs to be at an accepting state reading the right end-marker, so a path can only end on an accepting configuration. This is case (c) of Figure 2.3, where q_1 and q_3 are the final states.

So, now we can rephrase: x is accepted by N iff $g(x)$ is live; or, equivalently, there is a path in N from state q_1 on $\vdash$ to a final state on *dashv* iff there is a path from the leftmost to the rightmost column in $g(x)$. So, for every possible transition on the graph of configurations of N (described as $C_{N,w}$ in Chapter 1), there is an arrow in $g(\vdash w \dashv)$, and every possible computation path of N corresponds to a path in $g(\vdash w \dashv)$. Also, concerning the end-markers, the only node that has on arrow from the leftmost column is the one corresponding to the starting state, so there cannot be a live path from another node (valid computation paths start from the starting configuration), and the only nodes that are reachable on the rightmost column are the ones that correspond to final states, so live paths end only at those nodes (valid computations end at accepting configurations). This concludes the encoding of the computation of N as an instance of TWL_{2h}, so that whenever N has an accepting computation there is a live path in the graph and conversely. □

Lemma 8. *If* TWL∈2D *via a sequence 2DFAs with $s(h)$ states, then any problem in* 2N *can be solved by a sequence of 2DFAs with $s(h)$ states.*

Proof. Given a 2DFA G that solves TWL and a 2NFA N with poly(h)-states, we can construnct a 2DFA D that recognises the same problem as N, with the same set of states as G, by simulating the computation of G. We need only to clarify how D works on the edges, since the input of G has an extra symol on the edges ($g(\vdash)$ and $g(\dashv)$). In any other case $\delta_D(q, a_i) = \delta_G(q, g(a_i))$.

Right end-marker: when the head of G moves to the right to $g(\dashv)$ then if the head doesnt leave the two last symbols $g(\dashv)$ and $\dashv$ the movement of G is predetermined since it is no longer affected by the input and will either, eventually move back to the last symbol of the input or it will end its computation in some state, while reading $g(\dashv)$ or $\dashv$. In that sence, if G ends up in an accepting state over $\dashv$, D can move right away to $\dashv$ and to a final state. Similarly for the case G moves to a non final state with no transitions, then D does the same. In case G eventually moves its head left from $g(\dashv)$, to symbol $g(a_n)$ and in state q, then D moves to the right to $\dashv$ and then back to symbol a_n and state q.

Left end-marker can be dealt in a similar way to the right end-marker. When G moves left to $g(\vdash)$ then the machine will, at some point, either move right back to $g(a_1)$ and to state q, or it will not. In the first case D, after moving left to $\vdash$, moves back to the right, to $g(a_1)$ and q, and in the second case the machine moves to the left to $\vdash$ to a non final state with no transitions for $\vdash$.

Starting state: G starts from symbol $g(\vdash)$, while D starts from a_1, So to find the starting state we monitor the movement of G, until its head reaches $g(a_1)$ (again the movement is predetermined). The head might move between $g(\vdash)$ and $\vdash$ but eventually it will move to the right to symbol $g(a_1)$ and state q'. State q' is set as the starting state of D. There is the chance the machine does not move to $g(a_1)$ and halts. In that case the machine recognises the empty language. This completes the description of D. □

Theorem 7 (The Sakoda-Sipser completeness theorem)**.** TWL *is* 2N*-complete.*

Proof. 2D is closed under the above reduction therefore 2N is closed as well. This means TWL is 2N-complete.

□

It can be proven in an analogous way that OWL is 1N-complete.

Corollary 3. OWL *is* 1N*-complete.*

2.6 Hierarchy

By analogy to P, NP, coNP and the polynomial hierarchy of Turing machine complexity, there are two hierarchies, one for one-way machines, and another for two-way machines, which generalize the classes 1N, co1N and 2N, co2N, respectively.

Definition 2.13. $1\Sigma_k$ and $1\Pi_k$ are the classes of problems that can be solved by one-way alternating finite automata with polynomially many states, that make at most $k-1$ alternations between existential and universal states, starting with an existential or universal state, respectively.

Here $1\Pi_0 = 1\text{D} = 1\Sigma_0$, $1\Sigma_1 = 1\text{N}$ and $1\Pi_1 = \text{co1N}$.

Proposition 13. $1\Sigma_k = \text{co}1\Pi_k$.

Proof. let L be a problem in $1\Pi_k$. Then there is a sequence of 1AFAs, with polynomially many states that make at most $k-1$ alternations between existential and universal states, starting with a universal state, that solves L. We construct another sequence of 1AFAs. This one has the same states and the same transitions with the previous one, but the set of universal states and the set of final states has been reversed. The sequence solves problem $L^c \in \text{co}1\Pi_k$, but the problem is automatically in $1\Sigma_k$ by Definition 2.13.

In a similar fashion, for every problem $L \in 1\Sigma_k$ there is L^c which can be solved by a sequence of 1AFAs, according to the above construction. This proves that $L \in \text{co}1\Pi_k$. □

There are results concerning this hierarchy:

Theorem 8. *(V. Geffert [11]) There are witnesses for the following proper inclusions, for all $k \geq 1$:*

$$\begin{array}{ll} 1\Sigma_k \supset 1\Sigma_{k-1} & 1\Pi_k \supset 1\Sigma_{k-1} \\ 1\Sigma_k \supset 1\Pi_{k-1} & 1\Pi_k \supset 1\Pi_{k-1} \end{array}$$

Also, $1\Sigma_k$ and $1\Pi_k$ are incomparable, for all $k \geq 1$.

This means that this hierarchy is infinite and redundant. In a completely identical way there is the two-way hierarchy.

Definition 2.14. $2\Sigma_k$ and $2\Pi_k$ are the classes of problems that can be solved by two-way alternating finite automata with polynomially many states, that make at most $k-1$ alternations between existential and universal states, starting with an existential or universal state respectively.

Here $2\Pi_0 = 2\text{D} = 2\Sigma_0$ and $2\Sigma_1 = 2\text{N}$.

Theorem 9. *(V. Geffert [11]) There are witnesses for the following proper inclusions for $k \geq 2$:*

$$\begin{array}{ll} 2\Sigma_k \supset 2\Sigma_{k-1} & 2\Pi_k \supset 2\Sigma_{k-1} \\ 2\Sigma_k \supset 2\Pi_{k-1} & 2\Pi_k \supset 2\Pi_{k-1} \end{array}$$

Also, $2\Sigma_k$ and $2\Pi_k$ are incomparable, for all $k \geq 2$.

Similarly to the one-way hierarchy, this means that this hierarchy is infinite.

The differences with the one-way hierarchy are the following. It is not known whether $co2\Sigma_k = \Pi_k$. As described earlier, it is not known whether $2\Sigma_0 \subseteq 2\Sigma_1$ is a proper inclusion ($2D \subseteq 2N$), as also $2\Pi_0 \subseteq 2\Pi_1$. Finally it is not known whether $2\Sigma_1 = 2N$ and $2\Pi_1$ are comparable or not.

In the same theorem, Geffert proves some relations between the two hierarchies. Apart from the obvious $2\Sigma_k \supset 1\Sigma_k$ and $2\Pi_k \supset 1\Pi_k$, he also proves the following.

Theorem 10. *(V. Geffert [11]) There are witnesses for the following proper inclusions for $k \geq 2$. (the first group are results from the previous theorems):*

$$\begin{array}{ll} 2\Sigma_k \supset 1\Sigma_{k-1} & 2\Pi_k \supset 1\Sigma_{k-1} \\ 2\Sigma_k \supset 1\Pi_{k-1} & 2\Pi_k \supset 1\Pi_{k-1} \end{array}$$

and for the following classes of problems there are witnesses that prove them not to be empty

$$\begin{array}{ll} 1\Sigma_k \setminus 2\Sigma_{k-1} & 1\Pi_k \setminus 2\Sigma_{k-1} \\ 1\Sigma_k \setminus 2\Pi_{k-1} & 1\Pi_k \setminus 2\Pi_{k-1} \end{array}$$

Finally, $2\Sigma_k$ and $1\Pi_k$ are incomparable and so are $1\Sigma_k$ and $2\Pi_k$.

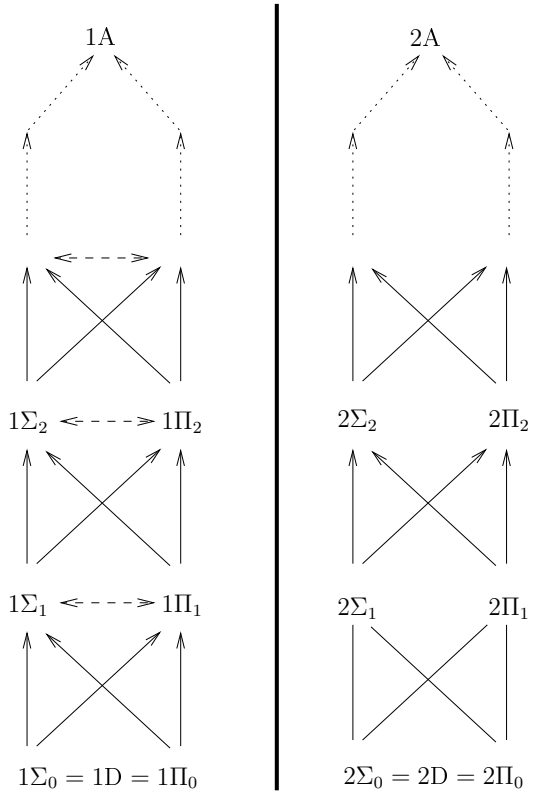

Figure 2.4: The two hierarchies for one-way and two-way machines, respectively.

2.7 Relations to computational complexity of TM

So, the basic problem this chapter revolves around is the 2D versus 2N question. Sakoda and Sipser conjectured that 2D$\subsetneq$2N. The question is hard to answer so, even stronger conjectures were made. First we define three restrictions of 2N:

Definition 2.15. The restriction of 2N to instances of exponential length is called 2N/exp. Respectively, 2N/poly is the restriction of 2N to instances of polynomial length, and 2N/const to instances of constant length.

At this point we need to define some restrictions of TWL:

Example 13. The problem LongTWL = $(\text{LongTWL}_h)_{h\geq 1}$ is a restriction of TWL. The alphabet is the same as in TWL and the input is bounded by an exponential function of h. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \mathcal{P}(\{1, 2, \ldots, 2h\}^2)$ and

$$\text{LongTWL}_h = \{w \in \Sigma_h^* |\ |w| \leq 2^h \text{ and } w \text{ has a live path}\}.$$

Example 14. The problem ShortTWL = $(\text{ShortTWL}_h)_{h\geq 1}$ is a restriction of TWL. The alphabet is the same as in TWL and the input is bounded by a polynomial function of h. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \mathcal{P}(\{1, 2, \ldots, 2h\}^2)$ and

$$\text{ShortTWL}_h = \{w \in \Sigma_h^* |\ |w| \leq h \text{ and } w \text{ has a live path}\}.$$

Example 15. The problem CTWL = $(\text{CTWL}_h)_{h\geq 1}$ is a restriction of TWL. The alphabet is the same as in TWL and the input is of length 2. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \mathcal{P}(\{1, 2, \ldots, 2h\}^2)$ and

$$\text{CTWL}_h = \{w \in \Sigma_h^* |\ |w| = 2 \text{ and } w \text{ has a live path}\}.$$

Theorem 11. longTWL *is complete in* 2N/exp.

Proof. According to the Sakoda-Sipser theorem TWL is 2N-complete, therefore any $L \in$2N can be reduced to TWL. Also longTWL $\in$2N/exp since longTWL is a problem in 2N with instances of exponential length. So, if $L \in$2N/exp, then it can be reduced to longTWL, since the g transformation described in Lemma 3 increases the length of all instances by 2. In case the instances of L are of length $O(f(h)) \geq O(2^h)$ then we define $h' = p(h)$ such that $O(2^{h'}) = O(2^{p}(h)) \geq O(f(h))$. Then L_h reduces to longTWL$_{h'}$. In other words, the reduction described keeps the size of the instances of L in 2N/exp.

□

Theorem 12. ShortTWL *is complete in* 2N/poly.

Proof. TWL is 2N-complete. Also shortTWL $\in$2N/poly since shortTWL is in 2N and its instances are of polynomial length. If $L \in$2N/poly then L reduces to TWL. But L has instances of polynomial length. Threrefore, through the g transformation, an instance of L reduces to an instance of TWL of polynomial length as well In case the instances of L are of length $O(p(h)) \geq O(h)$ then we can define $h' = p(h)$. Then L_h reduces to shortTWL$_{h'}$. Therefore ShortTWL is 2N/poly-complete.

□

It is still unknown whether any of these restrictions are in 2D, so we can focus on the smallest of these subclasses, TWL/const. The difference of this subclass from the other two, is that the other two have known complete problems, the two mentioned above. It is not known if 2N/const has a complete problem, and it hasn't been proven that CTWL$\in$2N/const is complete. This is an issue that will be considered in the next chapter. Furthermore, it is not known whether 1N$\subseteq$2D either. As a result, the frontline of problems that are conjectured not to be in 2D are CTWL, TWL and OWL. So, our basic question remains, the relation between 2D and 2N is uncertain. Through this chapter the analogy of P versus NP question becomes more clear, the distance between deterministic polynomial resource and non-deterministic polynomial resource. But is this quenstion important to complexity? What is the actual relation of the 2D versus 2N question with Turing machince computational complexity? It turns out that the size complexity of two-way finite automata is related to the space complexity of Turing machines. The result connecting two-way automata size complexity with Turing machine space complexity is the following:

Theorem 13 (C. Kapoutsis [15])**.**

$$\begin{aligned} &\text{2D} \supseteq \text{2N/poly} \iff \text{L/poly} \supseteq \text{NL} \\ &\text{2D} \supseteq \text{2N/exp} \iff \text{LL/polylog} \supseteq \text{NLL}\,, \end{aligned}$$

L is complexity class of all problems that can be solved by a deterministic Turing machine in logarithmic space. NL is complexity class of all problems that can be solved by a nondeterministic Turing machine in logarithmic space. LL is complexity class of all problems that can be solved by a deterministic Turing machine in $\log\log n$ space, where n is the length of the input. NLL is complexity class of all problems that can be solved by a nondeterministic Turing machine in $\log\log n$ space. L/poly is complexity class of all problems in L that have instances shorter than a polynomial of n. LL/polylog is complexity class of all problems in LL that have instances shorter than a polynomial of $\log n$.

We also note that the L versus NL qestion is the biggest open problem in Turing machine space complexity.

Chapter 3

CTWL and 2N/const

3.1 Introduction

In this last chapter we will describe a small effort to prove CTWL complete in 2N/const. The other restrictions of TWL can be proven to be complete problems in their according classes, immediately from the Sakoda-Sipser theorem: longTWL is 2N/exp-complete and shortTWL is 2N/poly-complete. But there is something fundamental about CTWL and 2N/const that does not seem right. There is a gap in all the possible instances of problems in 2N/const, which are unbounded, and the instances of CTWL, that are of constant length 2. The Sakoda-Sipser theorem proves that an instance of length m of a problem L in 2N can be transformed into an instance of TWL of length $m + 2$, and then it is shown how to construct a 2DFA that solves L out of a 2DFA that solves TWL, with twice as many states. This is why this proof is valid for proving that longTWL is 2N/exp-complete and shortTWL is 2N/poly-complete. It keeps the length of the input practically the same. But this is not helpful with the case of CTWL, since this problem has instances of constant length 2, and we need to prove that we can reduce to it any problem with instances of constant length no matter what this constant may be. The same question can be expressed for any other problem-candidate for 2N/const-completeness. It seems that our reductions are not suitable for this kind of work.

The reductions we have introduced so far are not capable of proving CTWL to be 2N/const-complete, or any problem for that matter. Let's suppose we want to prove CTWL to be 2N/const-complete. In order for this to be true, any other problem in 2N/const needs to reduce to it. Let's define problem 4TWL, the restriction of TWL to instances of length 4. This means we need to map every instance of length 4 to an instance of length 2. If all 4 symbols of the input of 4TWL have vital information for determining inclusion or exclusion in 4TWL, then by the pigeonhole principle, we need to include the information of 2 cells of the original instance to one cell of the output. Can this be done? The size of the alphabet of TWL_h is $2^{(2h)^2}$, and surely a polynomial-size transducer can not keep in its memory even one symbol of this alphabet. If it could, it would be useless for proving reductions in 2N. Due to the size of the alphabet we could apply the pigeonhole principle to the number states, compared to the number of possible h-tall, directed, two-column graphs. The result is the transducer cannot transfer the information intact. The same argument is valid for any problem in 2N/const. For any problem we want to prove to be complete, there is another problem with greater length of instances, in an exponential alphabet. Restrictions of TWL provide such examples.

3.2 Defining a new reduction

So, we need a reduction that can print the information of k cells into fewer cells, without keeping the information of any cell in its memory. Could there be a reduction, with respect to which CTWL can be proven to be 2N/const-complete? How would such a reduction manage to compress the information of the input tape? A natural idea is to be able to print more than once in every cell of the output. But then, should the machine be able to read the output tape? That would make it too powerful. That is because keeping in its memory once one symbol, would equal in memory an exponential number of states (in case the alphabet is exponential). But can the machine print on a cell that has been printed on before, without knowing what was printed on it? Let's suppose it can. Let's suppose the cells of the tape have a stucture, that allows the machine to print on top of a cell that has something printed on, without erasing the already accumulated information, but combining the existing information and the information that is about to be printed. This way the machine will be able to accumulate information, and build up to the final output of the process. But this requires a different kind of tape, one with some kind of structure, and an alphabet that is related to that structure.

Let's make it a bit more practical. Let's take a look at the alphabet of TWL, h-tall, two-way, two-column graphs. Every symbol of that alphabet can be represented as a set of arrows. Similar to the way we represent any graph as a set of edges, an h-tall, two-way, two-column graph can be a subset of the set $[2h] \times [2h]$. So, each cell can be structured so that it can have a special spot for each element in that set. Every time the machine prints the set a over the previous "symbol"-set b, it adds the elements of a over the elements of b that are printed already. So there is a operation applied on the tape, the union of a and b.

Now let's take a step back and look at a more general case. Let Σ be the alphabet and $\sqsubseteq$ a partial order on Σ. Before we define the machines we need some basic terms of order theory.

Definition 3.1. Let $(\Sigma, \sqsubseteq)$ be a partially ordered set and $S \subseteq \Sigma$. An element $x \in \Sigma$ is an *upper bound* of S if $s \sqsubseteq x$ for all $s \in S$. A *lower bound* is an $x \in \Sigma$ such that $x \sqsubseteq s$ for all $s \in S$. The set of all upper bounds of S is S^u and the set of all lower bounds is S^l. If S^u has a least element x, then x is called a *least upper bound*. Dually, if S^l has a greatest element x, then x is called the *greatest lower bound*.

Definition 3.2. We define $x \vee y$ to be the least upper bound of $\{x, y\}$, and we call it x *join* y. Dually, we define $x \wedge y$ to be the greatest lower bound of $\{x, y\}$, and we call it x *meet* y.

Definition 3.3. A partially ordered set (poset) P is a *lattice* if $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$. A partially ordered set P is a *join-semilattice* if $x \vee y$ exists for all $x, y \in P$.

Definition 3.4. Let L be a lattice. An element $x \in L$ is *join irreducible* if $x \neq \bot$ and $x = a \vee b$ implies $x = a$ or $x = b$ for all $a, b \in L$.

Unfortunately, this definition cannot serve us if we need to use the more general structure of a partially ordered set. For this case we will define a similar property.

Definition 3.5. Let $\sqsubseteq$ be a partial order over Σ. We call *prime elements* of Σ, $Primes(\sqsubseteq)$, the set of all $a \in \Sigma$ such that $a \sqsubseteq b_1 \vee b_2 \Rightarrow a \sqsubseteq b_1$ or $a \sqsubseteq b_2$.

Let's define now the deterministic automaton that has a tape with the required structure.

Definition 3.6. A *2DFA with structured-cell input* (2DFA$_{\sqsubseteq}$) is a tuple $M = (Q, \Sigma, q_0, q_f, \delta, \sqsubseteq$) where

- Q is the set of states,
- Σ is the input alphabet,
- q_0 is the starting state,
- q_f the final state,
- $\delta : \big(Q \times (Primes(\sqsubseteq) \cup \{\vdash, \dashv\}) \times \{+, -\}\big) \to \big(Q \times \{L, S, R\}\big)$ is the transition function, and
- $\sqsubseteq$ is a partial order on Σ.

So $(\Sigma, \sqsubseteq)$ is a partially ordered set and $Primes(\sqsubseteq)$ is the set of prime elements of that poset. The transition function δ is defined over the prime elements. Let $x \in Primes(\sqsubseteq)$ and $a \in \Sigma$. Then $\{+, -\}$ are the elements that define the two seperate cases, $x \sqsubseteq a$ or $x \not\sqsubseteq a$. In the special case where the head of the machine is over an end-marker, $\delta(q, \vdash, +) = \delta(q, \vdash, -)$ and $\delta(q, \dashv, +) = \delta(q, \dashv, -)$. This model is a restricted 2DFA model, that can have only transitions based on the prime elements of Σ. When $x \sqsubseteq y \iff x = y$, the automaton is a standard 2DFA. We continue with the definition of the transducer.

Definition 3.7. A *2DFT with structured-cell input and structured-cell output* (2DFT-OP where OP stands for over printing) is a tuple $T = (Q, \Sigma_1, \Sigma_2, q_0, q_f, \delta, \sqsubseteq_1, \sqsubseteq_2)$ where $Q, \Sigma_1, \Sigma_2, q_0, q_f$ are as in a standard 2DFT, and:

- $\sqsubseteq_1$ is a partial order on Σ_1,
- $\sqsubseteq_2$ is a partial order on Σ_2,
- $\delta : \big(Q \times (Primes(\sqsubseteq_1) \cup \{\vdash, \dashv\}) \times \{+, -\}\big) \to \big(Q \times \{L, S, R\} \times (\Sigma_2 \cup \{\lrcorner\}) \times \{S, R\}\big)$ is the transition function.

So $(\Sigma_1, \sqsubseteq_1)$ and $(\Sigma_2, \sqsubseteq_2)$ are partially ordered sets and $Primes(\sqsubseteq_1)$ and $Primes(\sqsubseteq_2)$ are their sets of prime elements, respectively. The output tape head can move only to the right or remain stationary. The transition function δ is defined over the prime elements of Σ_1 and the end-markers. Let $x \in Primes(\sqsubseteq_1)$ and $a \in \Sigma_1$. Then $\{+, -\}$ are the elements that define the two seperate cases, $x \sqsubseteq_1 a$ or $x \not\sqsubseteq_1 a$. In other words, in every state the machine has two transitions for a prime element. One transition in case the prime element is contained in the symbol being read, and one transition for the other case. This machine behaves in the same way as the 2DFA$_{\sqsubseteq}$, over the end-markers. This model can only have transitions based on the prime elements of Σ_1 (and the end-markers), and can print symbols of Σ_2 or $\lrcorner$. On the output tape the machine can print a symbol on top of another symbol. To write $a \in \Sigma_2 \cup \{\lrcorner\}$ on a cell that contains $b \in \Sigma_2 \cup \{\lrcorner\}$ means to replace b with $plub(b, a)$, where

$$plub(b, a) = \begin{cases} a \vee b & \text{if } a \vee b \text{ exists} \\ a & \text{if } a \vee b \text{ does not exist.} \end{cases}$$

The above definition allows us to define the following reduction.

Definition 3.8. Let $(L_h)_{h\geq 1}$ and $(L'_h)_{h\geq 1}$ be problems. We write $L \leq_{\text{2D-OP}} L'$ and say that L *reduces to* L' *in two-way overprint polynomial-size*, if there is a sequence of 2DFT-OPs $(T_h)_{h\geq 1}$ and two polynomials e and s such that every T_h has $s(h)$ states and maps instances of L_h to instances of $L'_{e(h)}$ so that for all x:

$$x \in L_h \Rightarrow T_h(x) \in L'_{e(h)} \text{ and } x \in L_h^c \Rightarrow T_h(x) \in (L'_{e(h)})^c.$$

We will show that the reduction we defined can solve the problem we encountered earlier: This machine can transform an instance of a given length to an instance of smaller length.

Example 16. The problem $4\text{TWL} = (4\text{TWL}_h)_{h\geq 1}$ is a restriction of TWL. The alphabet is the same as in TWL and the input is of length 4. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \mathcal{P}(\{1, 2, \ldots, 2h\}^2)$ and

$$4\text{TWL}_h = \{w \in \Sigma_h^* \mid |w| = 4 \text{ and } w \text{ has a live path}\}.$$

Proposition 14. $4\text{TWL} \leq_{\text{2D-OP}} \text{CTWL}$.

Proof. There is a 2DFT-OP T that reduces 4TWL_h to CTWL_{3h}. The process is described below. Suppose that the instance of 4TWL is $w = abcd$.

On reading a, the output head is over the first cell of the output tape. T prints the symbol e which is obtained from a as follows. For every arrow $i \to j \in a$:

- if $i, j \leq h$, then $i \to j \in e$;
- if $i \leq h$ and $j > h$, then $i \to (3h + j) \in e$;
- if $i > h$ and $j \leq h$, then $(3h + i) \to j \in e$; and
- if $i, j > h$, then $(3h + i) \to (3h + j) \in e$.

Note that e and a have the same number of arrows. Also $i \leq h$ means that i is on the left column of a and $i > h$ means that i is on the right column of a. Then the machine moves its input head and its output head to the right.

T reads the second symbol b, and prints the symbol f_1 that is derived from b as follows. For every arrow $i \to j \in b$:

- if $i, j \leq h$, then $i \to j \in f_1$;
- if $i \leq h$ and $j > h$, then $i \to (h + j) \in f_1$;
- if $i > h$ and $j \leq h$, then $(h + i) \to j \in f_1$; and
- if $i, j > h$, then $(h + i) \to (h + j) \in f_1$.

Afterwards the reading head moves to symbol c, while the output head stays on the same cell.

Similarly, on the next move, T reads the third symbol c, and prints on the same cell the symbol f_2 which is derived from c as follows. For every arrow $i \to j \in c$:

- if $i, j \leq h$, then $(h + i) \to (h + j) \in f_2$;
- if $i \leq h$, and $j > h$ then $(h + i) \to (2h + j) \in f_2$;
- if $i > h$, and $j \leq h$ then $(2h + i) \to (h + j) \in f_2$; and

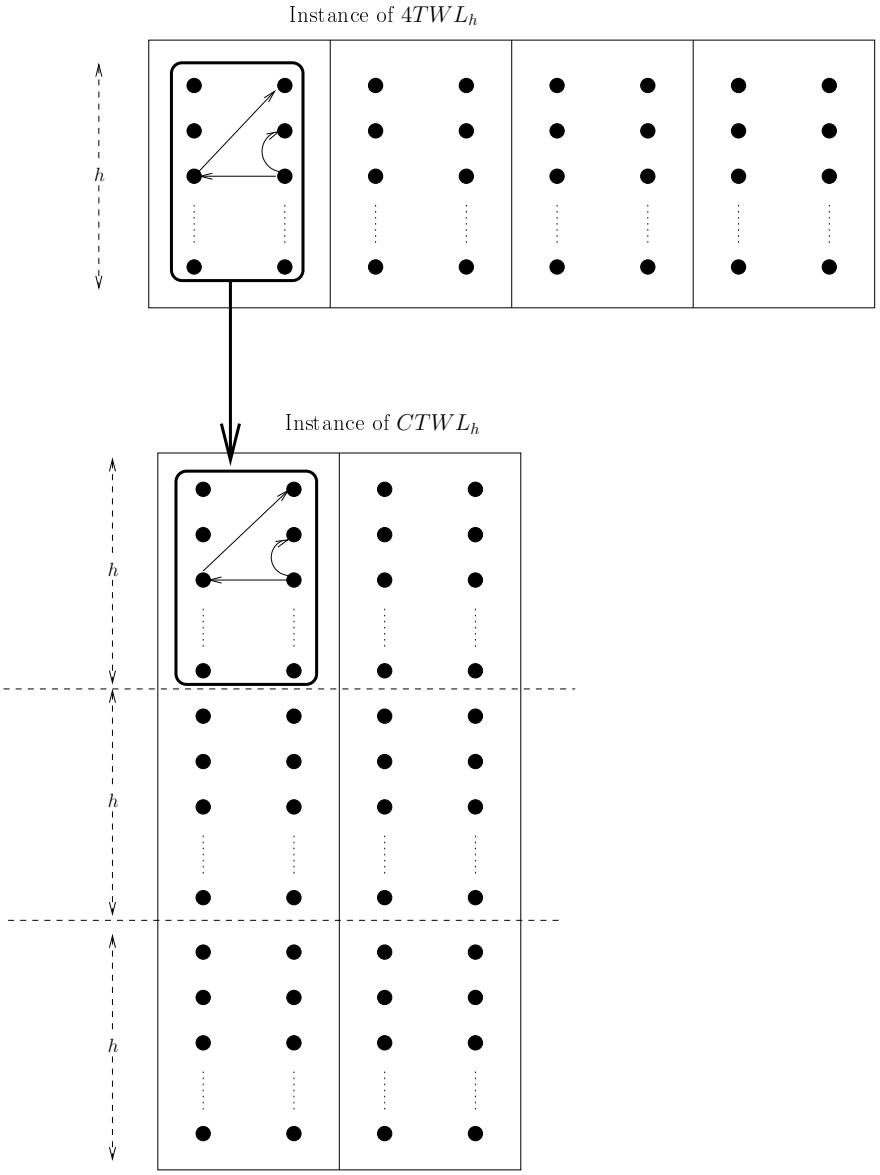

Figure 3.1: The first step of the transducer.

- if $i, j > h$, then $(2h + i) \to (2h + j) \in f_2$.

Again T moves its input head to the right, and the output head remains stationary.

Finally, T reads the fourth symbol d, and prints on the same cell the symbol f_3 which is derived from d as follows. For every arrow $i \to j \in c$:

- if $i, j \leq h$, then $(2h + i) \to (2h + j) \in f_3$;
- if $i \leq h$ and $j > h$, then $(2h + i) \to (5h + j) \in f_3$;
- if $i > h$ and $j \leq h$, then $(5h + i) \to (2h + j) \in f_3$; and
- if $i, j > h$, then $(5h + i) \to (5h + j) \in f_3$.

Then the machine moves both of its heads to the right over the right end-marker and halts.

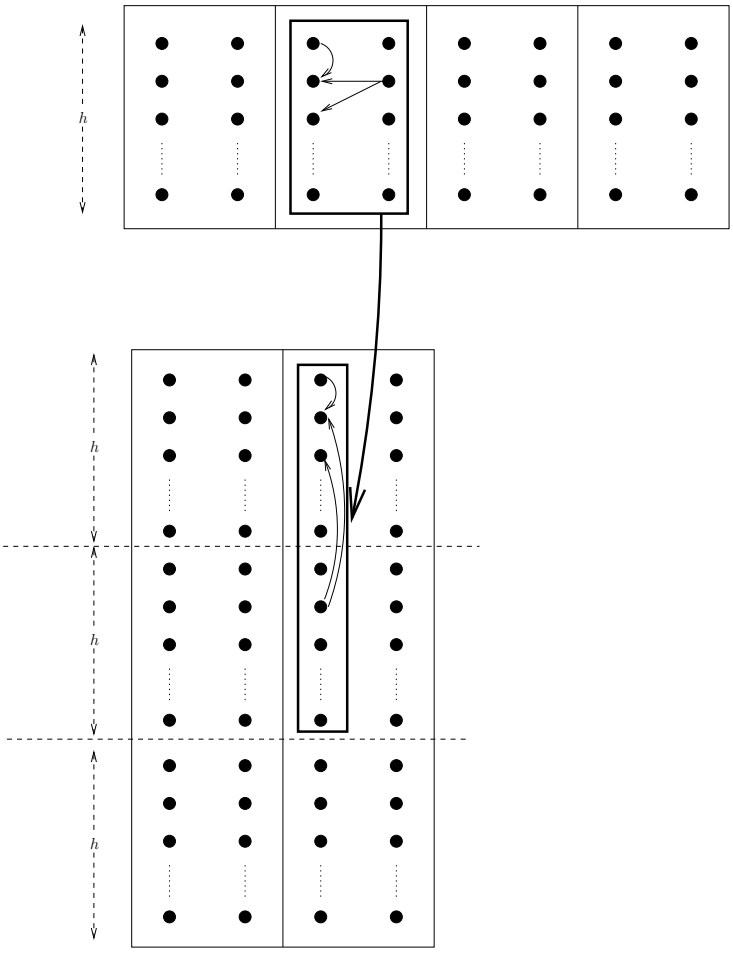

Figure 3.2: The second step of the transducer.

The main idea is that while in b we have two columns with arrows, in f_1 we move those two columns and place the first one on top of the second one. In that way, we have the same arrows and the same problem, in one fewer column, but in the same number of nodes. The process continues in the same sense and we create a $3h$-tall left column with three h-tall blocks. Each block corresponds to a column of the input *abcd*. Accordingly, the arrows of c move between the second block of the left column and the third block. Symbol d connects the third block with the right column. The whole idea is represented in the figures. In that way, a path that is live in the instance of 4TWL_h is live in in the instance CTWL_{3h}, and vice versa. The transducer described above needs only four states, to count which symbol it is reading, in order to print on the correct block. □

3.3 Outcome

Theorem 14. *If L is solved by a k-state 2NFA and all its instances are of length $\leq l$, then $L \leq_{\text{2D-OP}} \text{CTWL}_{(l+1)2k}$ via an $(l+1)$-state 2DFT-OP, where $\sqsubseteq_1$ is equality and $\sqsubseteq_2$ is the standard $\subseteq$.*

Proof. The main idea is similar to Lemma 3. Let $h = (l+1)2k$ If a $2k$-tall, two-way, multi-column graph can simulate the transitions of an arbitrary k-state 2NFA N on some

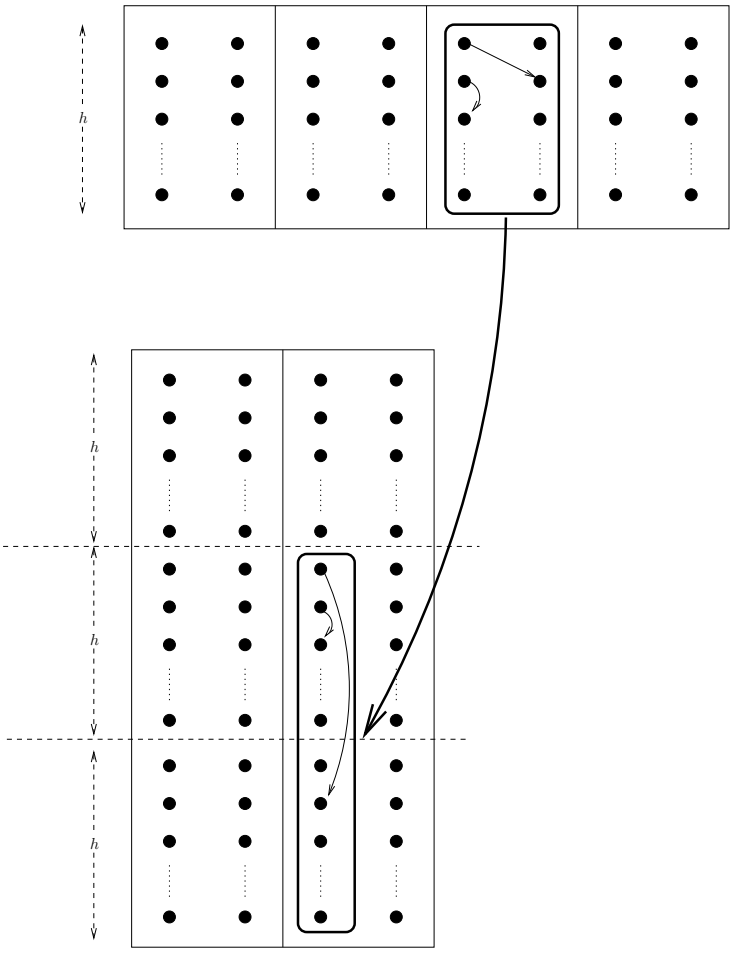

Figure 3.3: The third step of the transducer.

input x, then an h-tall, two-way, 3-column graph can simulate N on x as well, as long as there is a bound l for the length of x. Conveniently, the input of CTWL$_h$ is a h-tall, two-way, 3-column graph. Let $N = (\{q_1, \ldots, q_k\}, \Sigma_N, q_1, F_N, \delta_N)$ be a k-state 2NFA, with all instances of length $\leq l$. We will define the 2DFT-OP T that transforms the input string of N, $x = \vdash a_1 a_2 \cdots a_s \dashv$, into an instance $T(x)$ of CTWL$_h$, where $s \leq l$.

$$T = (\{p_0, p_1, \ldots, p_{l+1}\}, \Sigma_N, \Sigma_2, p_0, p_{l+1}, \delta_T, =, \subseteq)$$

where Σ_2 is the set of h-tall, two-way, two-column graphs, represented as sets of arrows $i \to j$ for $i, j \in \{1, 2, \ldots, 2h\}$. We define δ_T in the following way:

- $\delta_T(p_0, \vdash) = (p_1, R, g(\vdash, 0), R)$ is the transition of T on the starting configuration,
- $\delta_T(p_i, a) = (p_{i+1}, R, g(a, i), S)$ is the transition of T in the general case,
- $\delta_T(p_i, \dashv) = (p_{l+1}, R, g(\dashv, i), R)$ is the transition of T when the head reads the right end-marker, and the output head prints, moves to the right and the machine ends its process.

We can sum up the movement of the head by looking at the above transitions: The reading head moves one step at a time. The printing head starts at the first symbol, prints $g(\vdash, 0)$ and moves to the next cell. It remains at the second cell, printing $g(a, i)$ over the cell

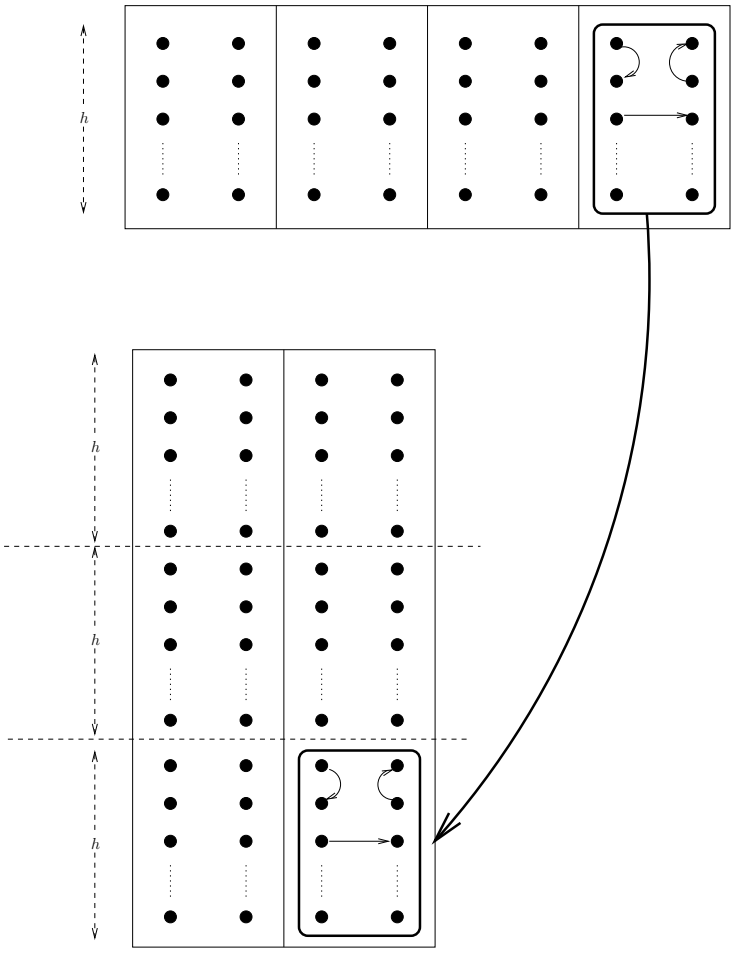

Figure 3.4: The fourth step of the transducer.

each time the reading head reads symbol a, while being in state q_i and then the machine moves to state q_{i+1}. When the reading head reads $\dashv$, the printing head prints $g(\dashv, i)$, and moves to the right, while the machine moves to the final state. Now we only need to define $g : \big(\Sigma_N \cup \{\vdash, \dashv\}\big) \times \{0, 1, \ldots, l+1\} \to \Sigma_2$ to conclude this construction of T. g takes in to account the symbol being read and the state machine T is in, and returns the symbol of Σ_2 that is going to be printed over. Let $h_i = (i-1)2k$. We define g in relation to δ_N:

- $g(\vdash, 0) = \{1 \to h+1\} \cup \{h+k+m \to h+j | (q_j, R) \in \delta_N(q_m, \vdash)\}$.

  This makes δ_T print an arrow for the starting configuration on the left column to the middle column. $h+j$ is the j-th node on the second column in each graph of Σ_2. The second part is about all the computation paths that, when coming from the symbol a_1 (first in input) to $\vdash$, move to the right of $\vdash$ back to a_1 and to state q_j.

- $g(a, i) = \{h_i + m \to h_i + 2k + j | (q_j, R) \in \delta_N(q_m, a)\}$
  $\cup \{h_i + 3k + m \to h_i + 2k + j | (q_j, R) \in \delta_N(q_m, a)\}$
  $\cup \{h_i + m \to h_i + k + j | (q_j, L) \in \delta_N(q_m, a)\}$
  $\cup \{h_i + 3k + m \to h_i + k + j | (q_j, L) \in \delta_N(q_m, a)\}$.

  Here, we have separated the h-tall, two-column graph in $l+1$ blocks. Each block is $2k$-tall. h_i is the offset of the i-th block. Every arrow that we describe is between that

block and the next one. To sum up, for every possible transition N can make while reading symbol a, there are two arrows in g that represent exactly that transition (one arrow for the case where the head has arrived in a from the right and one more for the case where the head has arrived in a from the left).

- $g(\dashv, i) = \{h_i + m \to h_i + k + j | (q_j, L) \in \delta_N(q_m, \dashv)\}$
  $\cup \{h_i + m \to h + h_i + m | q_m \in F_N\}$.

  Similarly to the first case, we have two possible movements under the rigth end-marker. The first set corresponds to computation paths that go right to $\dashv$, read the symbol, and move left to a_s to a state q_j. The second set corresponds to the final movement of any accepting configuration path: from being some state on the right end-marker, the machine moves to the right into an empty cell without changing state, and halts.

The above transformation is identical to the Sakoda-Sipser transformation, with only difference that we place the graphs in blocks we have created vertically in the middle column of the input. If N accepts x, then, there is a path from the starting configuration, through the transitions of δ_N, that ends at an accepting configuration. Equivalently, for a path in $\text{CTWL}_{2(l+2)s}$ to be live, it has to start from the first column (from the first column only the starting configuration of N has arrow to the middle column), it must follow the arrows that correspond to the transitions of δ_N (for every arrow in $g(x)$ there is a transition from one configuration to another, and vice versa), and finally it has to reach the third column (the only configurations reachable in the third column are those that correspond to the accepting configurations, through valid transitions). So now we can rephrase: x is accepted by $N_h \iff T(x)$ is live. □

The above lemma has two concequences.

Corollary 4. *Every problem in* 2N/const $\leq_{\text{2D-OP}}$*-reduces to* CTWL*.*

Proof. Let $(L_h)_{h\geq 1}$ a problem in 2N/const. This means the instances of L_h are of length $\leq l = const(h)$ and are solved by a 2NFA the number of states $k \leq p(h)$, for some polynomial p. According to Theorem 14 there is a family of 2DFT-OPs that can reduce $(L_h)_{h\geq 1}$ to $\text{CTWL}_{(l+1)2k}$. If we replace l and k accordingly we can reduce $(L_h)_{h\geq 1}$ to $\text{CTWL}_{(const(h)+1)2p(h)}$, where $(const(h) + 1)2p(h)$ is a polynomial of h, via a family of $const(h)$-state 2DFT-OPs. □

Corollary 5. *Every problem in* 2N/poly $\leq_{\text{2D-OP}}$*-reduces to* CTWL*.*

Proof. Proof is similar to the one above. The instances of L_h are of length $\leq s(h)$ and $k \leq p(h)$. So, every $(L_h)_{h\geq 1}$ can reduce to $\text{CTWL}_{(s(h)+1)2p(h)}$ via a family of $s(h)$-state 2DFT-OPs.

□

Theorem 15. *If L reduces to L' via an r-state 2DFT-OP, with output cells structured by a semilattice and L' can be solved by an s-state 2DFA$_\sqsubseteq$ that has instances of length $\leq l$, then L is solved by a 2DFA with $O(rsl^2)$ states.*

Proof. Let $T = (Q_T, \Sigma_1, \Sigma_2, q_0, q_f, \delta_T, =, \sqsubseteq)$ be the 2DFT-OP and $M = (Q_M, \Sigma_2, p_0, p_f, \delta_M, \sqsubseteq)$ be the $2DFA_{\sqsubseteq}$ of the statement. We will construct a 2DFA D with $O(rsl^2)$ states that recognises L. We now describe $D = (Q_D, \Sigma_1, s_0, F_D, \delta_D)$:

- $Q_D = (Q_T \cup \{q'\}) \times Q_M \times [l+1] \times [l+1]$ is such that we can keep in track of the states of T, M and their heads while we simulate their computation,
- $s_0 = (q_0, p_0, 1, 1)$ since in the beginning both machines are in their starting states and both heads are on the first cell of their tape,
- $F_D = \{(q_f, p_f, l_1, l_2) | 0 \leq l_1, l_2 \leq l+1\}$ and
- $\delta_D : Q_D \times (\Sigma_1 \cup \{\vdash, \dashv\}) \to Q_D \times \{L, S, R\}$ will be defined below.

In defining δ_D; we have to keep track of both machines being simulated. We have the following cases:

- If D is in a state of the form $(q', ..., 0, ...)$ then the simulation of T has been restarted. While D is in any of these states, it moves its head all the way to the first symbol of the tape, and starts the simulation of T again.
  - $\delta_D((q', p, 0, l_2), a) = ((q', p, 0, l_2), L)$, for all $a \in \Sigma_2$. This movement is while the head moves to the left.
  - $\delta_D((q', p, 0, l_2), \vdash) = ((q_0, p, 1, l_2), R)$. This movement is the last one: after the head has reached the left end-marker, it moves right to the first symbol of the input and to the starting state of T.
- If D is in a state of the form $(q_f, p, ..., ...)$ and its head is reading $\dashv$ then the simulation of T has halted. We have two subcases:
  - If $p = p_f$, then $\delta_D((q_f, p_f, l_1, l_2), \dashv) = (q_f, p_f, l_1, l_2), R)$. This means that the simulation of M halts and then D halts, as well.
  - If $p \neq p_f$, then $\delta_D((q_f, p, l_1, l_2), \dashv) = (q', \pi_1(\delta_M(p, \dashv, +)), 0, l_2^*), L)$, where

$$l_2^* = \begin{cases} l_2 & \text{if } \pi_2(\delta_M(p, \dashv, +)) = S \\ l_2 - 1 & \text{if } \pi_2(\delta_M(p, \dashv, +)) = L \end{cases}$$

    Note that π_i is used the it was defined in Chapter 1. This is the case when D simulates the computation of M and the head of M moves to the right end-marker. What actually happens is that M moves to the right after the last cell, and D starts simulating T to see what will be printed on that cell. But that cell will not be printed, instead machine T moves to $\dashv$ and enters the final state, so D knows that T will halt. This means that the head of M is over the right end-marker of its simulated input. So T needs not be simulated any further, D knows the moves M makes over the right end-marker.
- If D is in a state (q, p, l_1, l_2) where $q \neq q', q_f$ and $l_1 < l_2$, then T hasn't reached the point where it prints on the cell from which M is demanding to read. We set:

$$\delta_D((q, p, l_1, l_2), a) = (\pi_1(\delta_T(q, a)), p, l_1^*, l_2)$$

  where

$$l_1^* = \begin{cases} l_1 & \text{if } \pi_4(\delta_T(q, a)) = S \\ l_1 + 1 & \text{if } \pi_4(\delta_T(q, a)) = R. \end{cases}$$

This is simulating one movement of T. We don't need $\pi_3(\delta_T(q,a)) \in (\Sigma_2 \cup \{\textvisiblespace\})$, because we don't care what T prints on other cells than cell l_2 which is the one M is reading at the moment.

- If D is in a state (q,p,l_1,l_2) where $q \neq q', q_f$ and $l_1 = l_2 \geq 1$, then T is about to print a symbol on the cell from which M is reading. M checks if $x \sqsubseteq \pi_3(\delta_T(q,a))$.
  - If $x \sqsubseteq \pi_3(\delta_T(q,a))$, then $\delta_D((q,p,l_1,l_2),a) = (q', \pi_1(\delta_M(p,x,+)), 0, l_2^*), L)$.
  - If $x \not\sqsubseteq \pi_3(\delta_T(q,a))$ then $\delta_D((q,p,l_1,l_2),a) = (\pi_1(\delta_T(q,a),p,l_1^*,l_2), \pi_2(\delta_T(q,a)))$. Where
$$l_2^* = \begin{cases} l_2 - 1 & \text{if } \pi_2(\delta_M(p,x,+)) = L \\ l_2 & \text{if } \pi_2(\delta_M(p,x,+)) = S \\ l_2 + 1 & \text{if } \pi_2(\delta_M(p,x,+)) = R \end{cases}$$
    and l_1^* is as in above. The first case is for when M reads x in the symbol being printed; then M makes a move and D restarts the simulation of T. The second case is when M doesn't read x in the symbol; then D continues with the simulation of T.

- If D is in a state (q,p,l_1,l_2) where $q \neq q', q_f$, $l_1 = l_2 + 1$ and $l_2 \geq 1$, then the simulation of T has printed everything it could have on cell l_2 and the simulation moved on. We can now make the $\delta_M(p,x,-)$ move of M and restart the simulation of T.

  $\delta_D((q,p,l_1,l_2),a) = (q', \pi_1(\delta_M(p,x,-)), 0, l_2^*), L)$, where l_2^* is defined as above.

- If D is in a state (q,p,l_1,l_2) where $q \neq q', q_f$ and $l_2 = 0$, then this is the case where, during the simulation of M, its head is supposed to read the left end-marker. Then D doesn't need to simulate T, because the movement of M over $\vdash$ is independent of the input.

  $\delta_D((q,p,1,0),a) = ((q, \pi_1(\delta_M(p,\vdash,+)), 1, l_2^*), S)$, where
$$l_2^* = \begin{cases} 0 & \text{if } \pi_2(\delta_M(p,\vdash,+)) = S \\ 1 & \text{if } \pi_2(\delta_M(p,\vdash,+)) = R \end{cases}$$
  Note that the simulation of T has not started, and it will not start unless l_2 increases to 1.

There are no valid transitions that are not described in the cases above. This concludes the description of D. D simulates M and, every time M needs to check whether a prime element is in a cell, D starts simulating T to produce all the symbols that would be printed on that cell. Notice, however, that D cannot know the result of the overprinting in that cell, since it cannot remember a symbol of a potentially exponentially large alphabet. This is why we need $(\Sigma_2, \sqsubseteq)$ to be a semilattice. Because then x is $\sqsubseteq$ of the final result of the overprinting, iff it is $\sqsubseteq$ of one of the (partial) prints, by definition of the prime element. After that, M finally makes its move. Then D resets the computation of T to the initial configuration, while moving the reading head to the start of the input, and repeats the process. The simulation of the transitions of M on the end-markers has been clarified, and so has the halting of the machine. □

Corollary 6. *If $L \leq_{\text{2D-OP}} L'$ via a 2DFT-OP with output cells structured by a semilattice $\sqsubseteq$ and $L' \in$2D/poly via a 2DFA$_\sqsubseteq$ with polynomially many states and input cells structured by the same $\sqsubseteq$, then $L \in$2D.*

Proof. If every L_h reduces to $L'_{e(h)}$ via an $r(h)$-state 2DFT-OP, where $e(h)$ and $r(h)$ are polynomials, and every L'_h is solved by an $s(h)$-state 2DFA$_\sqsubseteq$ and has instances of length $\leq l(h)$, where $s(h)$ and $l(h)$ are also polynomials, then every L_h is solved by a 2DFA of size $O(r(h)s(e(h))l(e(h))^2)$, which is polynomial in h. □

An immediate consequence of Corollaries 5 and 6 is the following:

Corollary 7. *If* CTWL$\in$ 2D *via a 2DFA$_\subseteq$ whose input cells are structured by the standard* $\subseteq$*, then* 2N/poly$\subseteq$2D*.*

3.4 Generalization of the 2DFT-OP

We further investigate the case where the tape may have an arbitrary operation rather than the join of two elements of the alphabet.

Definition 3.9. A *two-way determistic finite transducer with an operating output tape* is a tuple

$$T = (Q, \Sigma_1, \Sigma_2, q_0, q_f, \delta, O_t)$$

where $Q, \Sigma_1, \Sigma_2, q_0, q_f$ are as in Definition 3.7, $\delta : Q \times (\Sigma_1 \cup \{\vdash, \dashv\}) \to Q \times \{L, S, R\} \times (\Sigma_2 \cup \{\textvisiblespace\}) \times \{S, R\}$ is the transition function, and $O_t : (\Sigma_2 \cup \{\textvisiblespace\}) \times (\Sigma_2 \cup \{\textvisiblespace\}) \to (\Sigma_2 \cup \{\textvisiblespace\})$ is the operation of the output tape.

When the head is over a cell, containing a symbol x and prints on that cell a symbol y, then the result of the overprinting is $O(x, y)$. When the symbol $\textvisiblespace$ is being printed, the cell remains unchanged. Formally $O(x, \textvisiblespace) = x$, for all x. Note that O is not necessarily commutative.

There seems to be a problem with this transducer, in terms of transitivity. A composition of two such transducers will have the same problem the 2DFT has. It cannot track down the movement of the two hypothetical heads of the two machines being simulated, namely the output head of the first machine and the input head of the second machine. But even if we restrict the print size polynomially, so that the simulator can keep two counters to track down the simulated heads, another problem still remains. How can a machine simulate the operation of the simulated intermediate tape without keeping in its memory at least one tape symbol? It seems it cannot. We futher investigate the weaknesses of the reduction defined by such a transducer. We first need to define the following problem.

Example 17. The problem 1OWL $= (1\text{OWL}_h)_{h\geq 1}$ is a restriction of OWL. The alphabet is the same as in OWL and the input is of length 1. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \mathcal{P}(\{1, 2, \ldots, h\}^2)$ and

$$1\text{OWL}_h = \{a \in \Sigma_h |\ a \text{ has at least one arrow}\}.$$

This problem is trivial. It can be solved by a 1-state 1DFA. The state checks whether there is an arrow from the left column to the right (any arrow at all). If there is at least one, then the head moves to the right. If there is none, then the machine halts. So, 1OWL $\in$1D.

Proposition 15. OWL *reduces to* 1OWL *via a 2DFT with an operating output tape.*

Proof. We describe the transducer $T = (Q, \Sigma_h, \Sigma_h, q_0, q_0, \delta, O_t)$ that reduces OWL_h to $1\mathrm{OWL}_h$. We let $Q = \{q_0\}$ and choose δ so that

$\delta(q_0, a) = (q_0, R, a, S)$, for all $a \in \Sigma_h$, and

$\delta(q_0, \dashv) = (q_0, R, \llcorner, R)$.

The computation of T is straightforward. At every step, it reads the next symbol a and it overprints it on the first cell of the output tape. When the reading head reaches $\dashv$, then the output head moves to the right and the computation ends.

More important, is the operation of the output tape. We let $O(b, a) = a \circ b$, the composition of a and b, if seen as relations on $\{1, 2, \ldots, h\}$.

The final output is a single symbol which is the composition of all symbols of the input. An arrow from the left column to the right column, in that symbol, is equivalent to a path from the left column of the first symbol of the input to the right column of the last symbol of the input. Note that the transducer does not use the ability of the input head to move backwards. □

Proposition 15 shows a reduction of a 1N-complete problem to a problem of 1D, while we know that 1D⊊1N. The next example exploits even further the power of the operating output tape.

Example 18. The problem $\mathrm{TRIVIAL} = (\mathrm{TRIVIAL}_h)_{h \geq 1}$. Formally, for every $h \geq 1$ the input alphabet is $\Sigma_h = \{1, 2, \ldots, 2^h + 1\}$ and

$$\mathrm{TRIVIAL}_h = \{2^h\}.$$

This problem is (trivially) in 1D since it can be solved by a single state 1DFA that looks at the one cell input and if it reads 2^h then accepts, otherwise declines.

Proposition 16. LONGLENGTH *reduces to TRIVIAL via a 2DFT with an operating tape.*

Proof. The following transducer T reduces $\mathrm{LONGLENGTH}_h$ to $\mathrm{TRIVIAL}_h$. We let $T = (Q, \Sigma_1, \Sigma_2, q_0, q_0, \delta, O_t)$, where

- $Q = \{q_0\}$ is the set of states
- $\Sigma_1 = \{0\}$
- $\Sigma_2 = \{1, 2, \ldots, 2^h + 1\}$

And δ is such that:

$\delta(q_0, 0) = (q_0, R, 1, S)$, and

$\delta(q_0, \dashv) = (q_0, R, \llcorner, R)$.

At each step, T reads the next input symbol (it can be either 0 or $\dashv$), prints 1 onto the first cell of the output tape, and moves the input head to the right, while the output head remains stationary. When the machine reads $\dashv$, it prints nothing, moves each head to the right and halting.

The operation of the output tape is defined as follows:

$$O(a, 1) = \begin{cases} 1 & \text{if } a = \llcorner \\ a + 1 & \text{if } a \leq 2^h \\ a & \text{if } a = 2^h + 1. \end{cases}$$

So, the transducer counts in one cell of the output tape the number of 0s in the input tape. If the number of 0s exceeds 2^h, then the counter gets stuck at $2^h + 1$. The first tape has 2^h 0s iff the second tape has 2^h as the final result of the overprinting. □

So, given that LONGLENGTH$\notin$2N by Proposition 8 and that TRIVIAL$\in$1D we see that a very difficult problem reduces to a trivial problem via a 2DFT-OP with an operating output tape. This reduction proves to be very strong, so strong that is actually of no use to us.

Corollary 8. *The reduction that can be defined by the generalization of 2DFT-OP, with an operating output tape, is such that none of* 1N, 2D *or* 2N *is closed under it.*

3.5 Conclusions

In this chapter we defined a new type or reduction, trying to prove CTWL to be 2N/const-complete. The whole idea was based on compression of information. The essence of the TWL problem is the same in its restrictions. One column with random arrows between its nodes is enough to make this problem difficult enough. But there seems to be a gap between TWL and ShortTWL or CTWL afterall, and it is the distance between having instances that are countable or not (by a small 2DFA). But the effort to compress the information that way, created another problem, on the way the machine accesses the information. As a result, this reduction took the problem to a different direction than the one that was intended. Finally, the generelization of the reduction is shown to be very strong to be of any practical use. The operation of the tape can be defined to deal with the problem, and reduce it to trivial, no matter how difficult the starting problem is.

Bibliography

[1] Christos H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[2] Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation (2nd ed.)*, Prentice Hall, 1998.

[3] John E. Hopcroft and Rajeev Motwani and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation - International Edition (3rd ed)*, Addison-Wesley, 2007.

[4] Brian A. Davey and Hilary A. Priestley, *Introduction to Lattices and Order (2nd ed.)*, Cambridge University Press, 2002.

[5] John L. Bell and Moshé Machover, *A course in mathematical logic*, North-Holland, 1977.

[6] Martin D. Davis and Ron Sigal and Elaine J. Weyuker, *Computability, Complexity, and Languages - Fundamentals of Theoretical Computer Science (2nd ed.)*, Academic Press, 1994.

[7] K. N. King, *Alternating Multihead Finite Automata*, Theor. Comput. Sci. **61** (1988), pp. 149-174.

[8] William J. Sakoda and Michael Sipser, *Nondeterminism and the Size of Two-Way Finite Automata*, Symposium on Theory of Computing (San Diego, 1978), ACM, 1978, pp. 275-286.

[9] Ashok K. Chandra and Dexter Kozen and Larry J. Stockmeyer, *Alternation*, J. ACM **28** (1981), pp. 114-133.

[10] Richard E. Ladner and Richard J. Lipton and Larry J. Stockmeyer, *Alternating Pushdown and Stack Automata*, SIAM J. Comput. **13** (1984), pp. 135-155.

[11] Viliam Geffert, *An alternating hierarchy for finite automata*, Theor. Comput. Sci. **445** (2012), pp. 1-24.

[12] M. O. Rabin and D. Scott, *Finite Automata and Their Decision Problems*, IBM Journal of Research and Development (1959).

[13] Christos A. Kapoutsis and Giovanni Pighizzini, *Two-Way Automata Characterizations of L/poly versus NL*, Proceedings of International Computer Science Symposium in Russia (2012), pp. 217-228.

[14] Christos A. Kapoutsis, *Minicomplexity*, Proceedings of Descriptional Complexity of Formal Systems (2012), pp. 20-42.

[15] Christos A. Kapoutsis, *Two-Way Automata versus Logarithmic Space*, Proceedings of International Computer Science Symposium in Russia (2011), pp. 359-372.

[16] Christos A. Kapoutsis, *Size Complexity of Two-Way Finite Automata*, Developments in Language Theory (2009), pp. 47-66.

[17] Panos Rondogiannis, *Semantics of Programming Languages*, preprint (2002).