Computing Models and The Power of Writing

Carolina Mejia, J. Andres Montoya

August 9, 2022

Part I
A Basic Model of Computation

1 Pattern Matching: Simple Patterns, Simple Machines

Notation 1 *Let us fix some notation:*

1. *Let Σ be a finite alphabet. We use the symbol Σ^* to denote the infinite set constituted by all the strings of finite length over the alphabet Σ.*
2. *We use the symbol Σ^n to denote the subset of Σ^* constituted by all the strings of length n.*
3. *Let $w \in \Sigma^n$, and let $i \leq n$, we use the symbol $w[i]$ to denote the i-th character of w.*
4. *Let $w \in \Sigma^n$, and let $i < j \leq n$, we use the symbol $w[i, ..., j]$ to denote the substring $w[i]\, w[i+1] \cdots w[j]$.*
5. *We use the symbol $|w|$ to denote the length of w.*

One of the most basic algorithmic tasks is to distinguish between the strings that hold some specific property α and the strings that do not hold this property.

Definition 2 *A language over Σ is a set $L \subset \Sigma^*$.*

Given a language L, defined by some abstract property, it determines an algorithmic problem, the problem:

Problem 3 ***Recognition***(L)

- *Input: $w \in \Sigma^*$.*
- *Problem: decide if $w \in L$.*

We focus on the above type of problems, which are called *decision problems.* Then, when we think in constructing an abstract model of a computing machine, we think in devices that deal with decision problems.

1.1 Pattern Matching, Fixed Patterns and Finite State Automata

Let us begin with one of the most basic algorithmic tasks, the task of recognizing the strings that contain as a substring a fixed pattern $W_0 \in \Sigma^*$. Let $W_0 \in \Sigma^*$ be a short pattern, and let L_{W_0} be the language

$$\{u \in \Sigma^* : \exists i \,(i + |w| - 1 \leq |u| \;\&\; u[i, ..., i + |w| - 1] = W_0)\}.$$

Notice that this is the language that recognizes a search engine when we type pattern W_0 and ask the browser to look for the sites that contain this term. Can we recognize this language using some simple type of devices?

1.1.1 Finite State Automata

Finite-state machines constitute the simplest model of computation. W. McCulloch and W. Pitts introduced this model as a combinatorial model of human neurons and neural activity [1].

Let $L \subset \Sigma^*$ be a language. We want to process the strings in Σ^* and tell apart the strings that belong to L. We need to have the ability of scanning those strings. Therefore, the basic model of computation must have an input unit, where we can write the input string w. This input unit, or tape, can be accessed by the machine in order to read the characters of w, either character by character or in a global fashion. It is not easy to figure out a machine that can read a long string in one time unit. It is very much easier to figure out a machine that can read the characters of w in a sequential fashion. Thus, let us begin with an elementary architecture, an automaton provided with an input tape and a scanning head that can move one-way along its input tape reading the characters of the input string. We have to observe that it is useless to read a string if one cannot save a minimum of information about the characters that are being read. Thus, our machine must be also equipped with a memory unit that allows it to save information about the characters that have been read so far. The formal definition of finite state automata goes as follows:

Definition 4 *A finite state automaton is a tuple $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$, such that:*

1. *Σ is a finite set of characters called the input alphabet of $\mathcal{M}$.*

2. *Q is a finite set of inner states used to save information about the characters that have been read. If $\mathcal{M}$ is in state q after reading the string w, then q corresponds to the summary taken by $\mathcal{M}$ about the text w.*

3. *$q_0 \in Q$ is the initial state. When $\mathcal{M}$ is ready to start the scanning of w, this automaton should be in a state that represents the fact that $\mathcal{M}$ has not read a single character, this state is the initial state q_0.*

4. *$A \subset Q$ is the set of accepting states.*

5. *δ is the transition function of $\mathcal{M}$, which is a function from the set $Q \times \Sigma$ into the set Q. Notice that an element of $Q \times \Sigma$ is a pair (q, a) that can represent the data that can be accessed by the automaton in a given instant of the computation: the character a that is written on the cell that is being read, and the state q that is written on its inner memory at this time instant. On the other hand, an element of Q is a state p that can represent the effect exerted by those data on the computation: a change in the content of the inner memory that switches from the state q to the state p.*

The computation of $\mathcal{M}$, on input w, proceeds as follows:

1. At instant 1 the string w is written on the input tape, the i-th character is written on the i-th cell. The scanning head is placed on the first cell, and the inner state is equal to q_0.

2. At instant 2 the scanning head is placed on cell 2, and the inner state has changed to $\delta(q_0, w[1])$.

3. At instant i the scanning head is placed on cell i. The inner state, at this time instant, is equal to $\widehat{\delta}(q_0, w[1,...,i])$, where $\widehat{\delta}$ is the extension of δ to the set $Q \times \Sigma^*$, extension that is defined by the following recursion:

   - For all $q \in Q$ and for all $a \in \Sigma$ the equality $\widehat{\delta}(q,a) = \delta(q,a)$ holds.
   - For all $q \in Q$ and for all $u \in \Sigma^n$ the equality
   $$\widehat{\delta}(q,u) = \delta\left(\widehat{\delta}(q, u[1,...,n-1]), u[n]\right)$$
   holds.

4. At instant $|w|+1$ the scanning head is placed on the first empty cell. This is the halting condition for $\mathcal{M}$, which is always reached after $|w|+1$ steps. The state reached by the automaton is the state $\widehat{\delta}(q_0, w)$. We say that $\mathcal{M}$ accepts the input, if and only if, state $\widehat{\delta}(q_0, w)$ belongs to A.

It is important to observe that:

1. Finite state automata cannot write. Those automata can only read and save a bounded amount of information from its reads (at most $\log(|Q|)$ bits).

2. Finite state automata are read-once devices, that is: the i-th cell is scanned only once, at the i-th instant of the computation.

3. Finite state automata are *real-time devices.*

We ask: which are the languages that can be recognized by finite state automata?

1.2 Pattern Matching and DFA's

We can use DFA's to recognize a language like L_{W_0}. Let $\mathcal{M}_{W_0}$ be the DFA that is defined by:

- The set of states is the set $Q = \left\{q_w : w \in \Sigma^{\leq n}\right\} \cup \{q\}$, where n is the length of W_0.
- The initial state is q_ε.
- The set of accepting states is the set $\{q, q_{W_0}\}$.

- The transition function is the function defined by

$$\delta\left(q_u, a\right)=\begin{cases} q_{ua}\text{, if } |u|<n \\ q_{u_2\cdots u_n a}\text{, if } u=u_1\cdots u_n\neq W_0 \\ q\text{, if } u=W_0\end{cases}$$

  Moreover, we have that $\delta(q,a)$ is equal to q for all $a\in\Sigma$. It is easy to check that:

1. If $w\in L_{W_0}$, then the computation of $\mathcal{M}_{W_0}$ on input w ends with the automaton in an inner state $p\in\{q,q_{W_0}\}$.

2. If $w\notin L_{W_0}$, then the computation of $\mathcal{M}_{W_0}$ on input w ends with the automaton in an inner state $p\notin\{q,q_{W_0}\}$.

Notation 5 *We say that $\mathcal{M}_{W_0}$ recognizes (accepts) the language L_{W_0} and we write the equation $L\left(\mathcal{M}_{W_0}\right)=L_{W_0}$ to symbolize this fact.*

How can we check that the above combinatorial construction works and that $\mathcal{M}_{W_0}$ actually recognizes L_{W_0}? The first question to ask refers the *meaning* of the set Q. Let $q_u\in Q$, the inner state q_u is more than a piece of the domain of δ. This inner state encodes information about the state of the computation.

Suppose that we want to decide whether a long string of letters S contains a specific chain w of length 10, suppose that we are read-once machines like $\mathcal{M}_{W_0}$. We claim that it would be a good idea to save the last 10 letters we read. We can save this information in a small blackboard on which only up to 10 letters can be written. Let $u_{t-9}\cdots u_t$ be the message that is written on the above blackboard, and suppose we read a new character u. We *shift* $u_{t-9}\cdots u_t$ and replace this string by $u_{t-8}\cdots u_t u$. It takes one time unit to read the short string that is now written on the blackboard. If this this string is different of w we continue with the scanning of S. On the other hand, if the string written is equal to w we skip the scanning and accept S, but we continue with our walk until the end of S (we enter a special skip state and we stop writing on the blackboard). We reject only if we reach the end of S without reaching the skip state.

The above procedure corresponds to the computation of a DFA that looks for the pattern w. The small blackboard is the memory unit of this DFA, and the message that is written on it, at some instant of the computation, is all information about the state of the computation that can be accessed by this DFA. Those messages are the states. Then, the initial state has to be equal to q_ε because the empty word is precisely the string that is written on the blackboard before the scanning of the first character. The accepting states are the skip state and the pattern w. The transition function is the delete-shift-inserte operation that was described in the previous lines.

We have to observe that $\mathcal{M}_{W_0}$ is identical to the above DFA. The former automaton has a blackboard where it can write on up to n characters from Σ. Given $u\in\Sigma^{\leq n}$, the state q_u corresponds to the fact that u is the string that is written on the blackboard of $\mathcal{M}_{W_0}$. We have to observe that any DFA $\mathcal{M}$ is analogous to $\mathcal{M}_{W_0}$. Automaton $\mathcal{M}$ has a small blackboard where it writes small messages that encode the most relevant facts about the computation. Each time $\mathcal{M}$ reads a new character, it deletes the actual message and updates its inner state by writing a new message on this blackboard.

Exercise 6 *Construct a DFA that recognizes the language $\{w\}$, where w is a string in Σ^*.*

Exercise 7 *Let u, v be two strings, let $L_{u,v}$ be the language constituted by all the strings that contains non-overlapping occurrences of u and v. Construct a DFA that accepts $L_{u,v}$.*

Exercise 8 *Suppose that L is recognized by a DFA, prove that co-L can be recognized by a DFA.*

Exercise 9 *Suppose that L and T are recognized by DFA's, prove that $L \cup T$ and $L \cap T$ can be recognized by DFA's*

Exercise 10 *Prove that any finite set of strings can be recognized by a DFA.*

Exercise 11 *Let $w = w_1 \cdots w_n$. The reverse of w is the string $w^R = w_n \cdots w_1$. Let L be a language. We use the symbol L^R to denote the language*

$$\left\{w^R : w \in L\right\}.$$

Suppose that L can be recognized by DFA's. Prove that L^R can be recognized by DFA's

Exercise 12 *Let $\Sigma = \{0, 1, ..., 9\}^3$ and let $(a, b, c) \in \Sigma$. Observe that we can write this character (and any other character) as a column vector $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$. Let S be the language*

$$\left\{ \begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \cdots \begin{pmatrix} a_m \\ b_m \\ c_m \end{pmatrix} : m \geq 1 \text{ and } a_1 \cdots a_m + b_1 \cdots b_m = c_1 \cdots c_m \right\}.$$

Prove that this language can be recognized by DFA's.

Exercise 13 *Let $L, T \subset \Sigma^*$ be two languages, we define the concatenation of L and T by*

$$L \cdot T = \{uw \in \Sigma^* : u \in L \text{ and } w \in T\}.$$

Suppose that L and T are recognized by DFA's. Prove that $L \cdot T$ can be recognized by DFA's.

References

[1] W. McCulloch, W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. Bulletin of Mathematical Biophysics. 5 (4): 115 - 133, 1943.

[2] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing, Boston, 1997.

2 Variable Patterns and the Need of Markers

Each time that we use a search engine we type a different pattern W. A search engine cannot be equal to a single DFA like $\mathcal{M}_{w_0}$. The search engine of our browser must be able to solve *The Pattern Matching Problem.*

Definition 14 *Let # is a symbol that does not belong to* Σ*. The Pattern Matching Problem over the alphabet* Σ *is the language:*

$$PM_\Sigma = \{w\#u : w, u \in \Sigma^*,\ w \text{ occurs in } u \text{ as a substring}\}.$$

Can DFA's recognize this language? We prove that DFA's cannot recognize this language. We prove this for $|\Sigma| = 1$. The proof works for any alphabet size.

Theorem 15 *Let* $|\Sigma| = 1$*, the language* PM_Σ *cannot be recognized by DFA's.*

Proof. Suppose that PM_Σ can be recognized by a DFA $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$. We define a function $F_\mathcal{M} : \mathbb{N} \to Q$ as follows

$$F_\mathcal{M}(n) = \widehat{\delta}(q_0, a^n).$$

We have that $F_\mathcal{M}$ cannot be injective. Thus, let $n < m$ such that $F_\mathcal{M}(n) = F_\mathcal{M}(m)$. Notice that $a^n\#a^n \in PM_\Sigma$ but $a^m\#a^n \in PM_\Sigma$. We also have that

$$\widehat{\delta}(q_0, a^n\#a^n) = \widehat{\delta}\left(\widehat{\delta}(q_0, a^n), \#a^n\right) = \widehat{\delta}\left(\widehat{\delta}(q_0, a^m), \#a^n\right) = \widehat{\delta}(q_0, a^m\#a^n),$$

and we get a contradiction. ■

Definition 16 *Let Squares be the language*

$$\{w\#w : w \in \Sigma^*\}.$$

Notice that Squares is a sublanguage of PM_Σ*.*

We claim that recognizing Squares is exactly as hard as recognizing PM_Σ. The analysis of Squares is easier than the analysis of PM_Σ. Therefore, we focus on the language Squares.

2.1 2DFA's and 1-Pebble Automata

Suppose we have to cope with the language Squares. We would like to be allowed to go back to the left in order to read some characters that were previously scanned but which were forgotten. Two-way finite state automata (2DFA's, for short) are automata that can move in both directions, and which can visit cells that were previously accessed. As with DFA's, there are a finite number of states with transitions between them based on the current character, but each transition is also labelled with a value indicating whether the machine will move its head to the left, right, or stay at the same position.

Definition 17 *A 2DFA is a tuple $\mathcal{M} = (Q, \Sigma, q_0, T, A, \delta)$ that behaves analogously to a finite state automaton, except that:*

- *$A \subseteq T \subseteq Q$. The set T is the set of halting states, which contains the set of accepting states and a possibly empty set of rejecting states. If $\mathcal{M}$ reaches a state in T the computation halts.*
- *The transition function δ is a function from the set $Q \times (\Sigma \cup \{\square\})$ into the set $Q \times \{-1, 0, 1\}$. The symbol $\square$ is used to indicate that the scanning head is placed on an empty cell. The symbols in the set $\{-1, 0, 1\}$ are used to indicate whether the scanning head will move to the left, stay at the same position or move to the right.*

Definition 18 *Let $\mathcal{M} = (Q, \Sigma, q_0, T, A, \delta)$ be a 2DFA, and let $w \in \Sigma^*$, we say that $\mathcal{M}$ accepts w, if and only if, the computation of $\mathcal{M}$, on input w, reaches an accepting state.*

The language accepted by $\mathcal{M}$, denoted with the symbol $L(\mathcal{M})$, is equal to the set of strings that are accepted by $\mathcal{M}$.

Does there exist a 2DFA $\mathcal{M}$ that accepts the language that we call Squares?

2DFAs were introduced in a seminal 1959 paper by Rabin and Scott [1]. Rabin and Scott proved that 2DFA's can recognize exactly the same languages as DFA's. We prove something weaker, we prove that 2DFA's cannot recognize the language Squares. We get as a consequence that 2DFA's cannot solve the pattern matching problem with variable patterns (2DFA's cannot recognize the language PM_Σ).

2.1.1 One-Pebble Automata

Suppose we are a 2DFA. We can go back to the left in order to check some cells that were previously visited but which we have forgotten. How can we recognize those cells among the ocean of similar cells? If we cannot distinguish the cells that we want to check it becomes useless to move leftward. Let us provide 2DFA's with a mechanism for marking cells. Let us imagine 2DFA's provided with a certain number of pebbles. Those pebbles can be placed on the tape, can be sensed and can be picked up from the tape when necessary. We use the term pebble automata to designate those computing devices. Let us first consider pebble automata provided with a single pebble.

Definition 19 *A one-pebble 2DFA is a tuple $\mathcal{M} = (Q, \Sigma, q_0, T, A, \delta)$ that behaves analogously to a 2DFA, except that it is provided with one pebble.*

Let $\mathcal{M}$ be a one-pebble 2DFA. The transition function δ is a function from the set

$$Q \times \{0, 1\} \times (\Sigma \cup \{\square\}) \times \{0, 1\}$$

into the set

$$Q \times \{-1, 0, 1\} \times \{0, 1, 2\} \times \{0, 1\}.$$

Let w an input of $\mathcal{M}$, and let us suppose that $\mathcal{M}$ is processing input w. Let t be a time instant along this computation. The automaton can see, at instant t, its own inner state, whether it has used its pebble, the character from $\Sigma \cup \{\square\}$ that is written on the cell it is scanning, and it possibly sees the pebble placed on this cell. Given these data the automaton updates its inner state, decides the direction of its next move and decides what to do with the pebble.

Theorem 20 *One-pebble automata cannot recognize the language Squares.*

Proof. Let $\mathcal{M} = (Q, \Sigma, q_0, T, A, \delta)$ be a one-pebble automaton. We prove that $\mathcal{M}$ cannot recognize Squares.

Let $w\#u$ be an input of $\mathcal{M}$. We can suppose that the computation of $\mathcal{M}$, on input $w\#u$, starts with the input head located on the cell that contains the letter # (the #-cell) and with the automaton carrying with its pebble. We can also suppose that the first move of the automaton is to the left.

Let us first suppose that the automaton cannot cross the #-cell while it is carrying with its pebble, that is: the pebble stays at the left side of the #-cell the whole computation. Given $u \in \{0,1\}^*$, we define a function $f_u : Q \to Q$. Function f_u gives us a table with the answers that we can expect from the right side when this side is occupied by u (or a string like u), that is:

Suppose that we can see only the prefix $w\#$ and the work of $\mathcal{M}$ on this prefix. Each time the input head crosses the #-cell we loss control on the computation. However, if we know f_u, and we know that u is the suffix that comes after #, we can simulate the computation of $\mathcal{M}$ on input $w\#u$. We set

$$f_u(q) = \begin{cases} (p,1)\text{, if and only if, when the automaton crosses} \\ \text{the \#-cell going to right and being in the state } q, \\ \text{it comes back after some finite time, going to the left} \\ \text{and with its inner state being equal to } p. \\ (p,0)\text{, if and only if, when the automaton crosses} \\ \text{the \#-cell going to the right and being in the state } q, \\ \text{it does not come back anymore, because it halts} \\ \text{there being its halting state equal to } p. \end{cases}$$

The number of functions from Q to Q is equal to $|Q|^{|Q|}$. On the other hand, we have that there are infinite many binary strings. We get that there exist two strings $w \neq u$ such that $f_u = f_w$. The string $w\#u$ does not belong to Squares. However, it can be proved that either $\mathcal{M}$ accepts $w\#u$ or $\mathcal{M}$ rejects $w\#w$. The latter two alternatives imply that $\mathcal{M}$ cannot recognize Squares.

It remains to consider one-pebble automata that can work with their pebble at both sides of the #-cell. Let $\mathcal{M} = (Q, \Sigma, q_0, T, A, \delta)$ be one of those automata. The difference with the previous case is that $\mathcal{M}$ does not really distinguish between the left and the right sides of the tape. Therefore, we assign tho each string $u \in \{0,1\}^*$ two functions (in Q^Q) that we denote with the symbols L_u and D_u. Function L_u gives us a table with the answers that we can expect from u, when u is located at the left side, we are at the right side and we can see the pebble when the automaton crosses the #-cell. Function D_u is defined accordingly, this time assuming that we are at the left and u is at the right. There exist $w \neq u$ such that $(L_u, D_u) = (L_w, D_w)$. Let w, u be such a pair of strings. We claim that $w\#u$ fools $\mathcal{M}$, that is: either $\mathcal{M}$ accepts $w\#u$ or $\mathcal{M}$ rejects $w\#w$ or $\mathcal{M}$ rejects $u\#u$. Let us see:

Suppose that $\mathcal{M}$ is processing an input that is picked out from the set $\{w\#w, w\#u, u\#u\}$. Let us suppose that this is a lazy automaton that prefers to stay, as long as possible, at a single side of the tape. Let us also suppose that $\mathcal{M}$ believes that it got as input either $w\#w$ or $u\#u$. Automaton $\mathcal{M}$ is confident that it can distinguish between these two strings that belong to Squares and $w\#u$ that does not belong to Squares. The computation starts from the #-cell with a move to the left. We get that $\mathcal{M}$ will stay, as long as possible, at the left side. When the transition function forces it to cross the #-cell, leaving the pebble at the left side, it omits this command and queries the function $D_w = D_u$ about the value of $D_w(p)$, where p is the inner state of $\mathcal{M}$ at this time instant. It works, independently of the right side being occupied either by w or by u. The computation flows well, with $\mathcal{M}$ steadily increasing its faith in that it got a string in the language Squares. If the transition function forces $\mathcal{M}$ to cross the #-cell while carrying with the pebble, it has to accept this command and leave the left side for the first time, otherwise it will loss control on the computation. After $\mathcal{M}$ suffers this first alternation, between the left and the right side, it tries to stay at the right side. $\mathcal{M}$ queries the function $L_u = L_w$ when possible. Everything works well, and $\mathcal{M}$ increases its faith in the possibly wrong belief that it got a good string. The computation continues in this way, with $\mathcal{M}$ alternating between the left and the right side, without being able of finding a mismatch, and steadily increasing its faith in the belief that the input string belongs to Squares.

We have to conclude that $w\#u$ fools $\mathcal{M}$, and that $\mathcal{M}$ cannot distinguish between the three strings in the set $\{w\#w, w\#u, u\#u\}$. This disables $\mathcal{M}$ as a possible *Squares Recognizer*. The theorem is proved. ■

2.2 Two Pebbles

Two-pebble automata are like one-pebble automata, except that the former are provided with two pebbles.

Exercise 21 *Provide a precise definition of two-pebble automata.*

Does the addition of a second pebble increase the computational power? It is easy to construct a two-pebble automaton that recognizes the language Squares. Two pebbles suffice to implement a *zig-zag strategy*:

*One pebble is placed on the left end of the input string, and the second pebble is placed on the cell that is next to the #-cell. Those two pebbles are used to mark pairs of cells, one at the left and one at the right of the #-*cell. *The contents of those marked cells are checked for mismatches. Both pebbles are moved one position to the right after each checking. This ensures that if the left* *pebble is placed i positions to the right of the left-end of the input, then the right pebble is placed i positions to the right of the* #-cell. *The automaton halts and rejects when it finds a mismatch. The automaton accepts when it cannot find a mismatch.*

It is also easy to construct a two-pebble automaton that recognizes the language PM_Σ and which is based on the same type of zig-zag strategy.

Exercise 22 *Construct the aforementioned two-pebble automaton.*

We have to conclude that two pebbles suffices to solve The Pattern Matching Problem. What can be done with more than two pebbles?

2.2.1 Three and More Pebbles

Pebble automata can be used to recognize complex languages:

Exercise 23 *Prove that three pebbles suffice to recognize the language*

$$\left\{a^{2^n} : n \geq 1\right\}.$$

It seems that the addition of pebbles allows the recognition of more complex languages:

Exercise 24 *Prove that one can use three pebbles to recognize the language*

$$\{a^p : p \text{ is a prime number}\}.$$

Exercise 25 *Prove that one can use four pebbles to recognize the language*

$$\left\{a^n \# a^{2^n} : n \geq 1\right\}.$$

Conclude that one can use four pebbles to compute the function 2^n.

Exercise 26 *Prove that one can use five pebbles to recognize the language*

$$\left\{a^n \# a^k \# a^{k^n} : n \geq 1\right\},$$

conclude that five pebbles suffice to make arithmetic.

Thus, it seems that the addition of pebbles provides us with computing power. Let us ask: do there exist k and a language that can be recognized using $k+1$ pebbles but which cannot be recognized using k pebbles?

2.3 Closure Under Regular Operations and State Complexity

DFA's, 2DFA's and one-pebble automata recognize the same set of languages. Two-pebble automata recognize a larger set that includes the language Squares and The Pattern Matching Problem.

Definition 27 *We use the term regular languages to designate the languages that can be recognized by DFA's.*

The reader could think that we wasted our time studying 2DFA's and one-pebble automata. We invented a stronger hardware and we got no reward: those machines can only recognize regular languages. We have to claim that we did not waste our time. We can use this improved hardware to construct smaller automata. Recall the exercises at the end of the first lecture. We proved that the set of regular languages is closed under several operations as taking reversals or concatenations. We had to use a lot of states in order to recognize the languages L^R and $L \cdot T$, that is: we began with an automaton $\mathcal{M}$ with n states and which recognizes the language L, and we ended with an automaton $\mathcal{M}^R$ that recognizes L^R and uses something like 2^n states. This exponential blow-up cannot be avoided if we stick with DFA's. On the other hand, we have that it is fairly easy to construct a 2DFA $\mathcal{N}$ that recognizes L^R and which uses $O(n)$ states.

Exercise 28 *Prove the latter assertion.*

Exercise 29 *Use one-pebble automata to prove, once again, that the set of regular languages is closed under concatenations. How many states did you use in this construction?*

2.3.1 Star-Free Languages and One-Pebble Automata

Let us begin with a definition.

Definition 30 *The star-free languages are the regular languages that can be constructed from the empty language and the singleton languages using no more than unions, intersections, concatenations and complements.*

We can provide a recursive definition of the above set of languages:

1. $\emptyset$ is a star-free language. If $a \in \Sigma$ the language $\{a\}$ is star-free. We say that those languages, the empty language and the singleton languages, are the basic star-free languages.
2. If L and T are star-free then $L \cup T, L \cdot T$ and *co*-L are star-free.

3. Any star-free language can be constructed from the basic languages applying the star-free operations (union, concatenation and complement) a finite number of times.

The above third item suggests that we can describe star-free languages using the *construction plans* of those languages: using short strings that describe the way those languages are constructed from the basic languages with the help of the star-free operations. Let us try a recursive definition of *star-free expressions.*

Definition 31 *Let Σ be a finite alphabet, we define the set of star-free expressions over Σ, and we simultaneously define the language denoted by each one of those expressions:*

1. *ε is a star-free expression and the language denoted by ε is the empty language.*
2. *a is a star-free expression and the language denoted by a is the singleton language $\{a\}$.*
3. *Suppose that α, β are star-free expressions and suppose that L_α and L_β are the languages denoted by these expressions. We have:*
   - *$(\alpha \cup \beta)$ is a star-free expression and the language denoted by $\alpha \cup \beta$ is the language $L_\alpha \cup L_\beta$.*
   - *$(\alpha \cdot \beta)$ is a star-free expression and the language denoted by $\alpha \cdot \beta$ is the language $L_\alpha \cdot L_\beta$.*
   - *$(\overline{\alpha})$ is a star-free expression and the language denoted by $\overline{\alpha}$ is the language co-L_α.*

Suppose we are given a star-free expression α. We can use α to describe the language L_α. We cannot use α to recognize L_α simply because α is not a computing device, and hence α does not compute. However, we can use α to construct a DFA $\mathcal{M}_\alpha$ that recognizes L_α. The construction is recursive:

1. $\mathcal{M}_\varepsilon$ is the automaton with just one state that rejects all the strings.
2. $\mathcal{M}_a$ is the automaton

   $$(\{q_0, q_1, q_2\}, \Sigma, q_0, \{q_1\}, \delta),$$

   where δ is defined by

   $$\delta(q, b) = \begin{cases} q_1, \text{ if } q = q_0 \text{ and } b = a \\ q_2, \text{ otherwise} \end{cases}.$$

   Exercise 32 *Check that $\mathcal{M}_a$ recognizes the language $\{a\}$.*

3. Given $\alpha, \beta, \mathcal{M}_\alpha$ and $\mathcal{M}_\beta$, we use the aforementioned constructions to define $\mathcal{M}_{\alpha\cup\beta}, \mathcal{M}_{\alpha\cdot\beta}$ and $\mathcal{M}_{\overline{\alpha}}$.

The above construction yields a *compiler* that we denote with the symbol *Comp-SF*. This algorithm computes, on input α, a DFA $\mathcal{M}_\alpha$ that recognizes the language L_α.

References

[1] M. Rabin, D. Scott. Finite automata and their decision problems. IBM Journal of Research and Development 3 (2): 114–125, 1959.

[2] J. Shepherdson. The reduction of two-way automata to one-way automata. IBM Journal of Research and Development 3(2): 198 - 200, 1959.

[3] M. Blum, Ch. Hewitt. Automata on a 2-Dimensional Tape. Proceedings of SWAT (FOCS) 1967: 155-160.

3 Pattern Matching with Three Pebbles

We could solve the pattern matching problem using two pebbles. Now, we can consider some more sophisticated patterns. Consider the following problem

Problem 33 *Star-Free Membership Problem*

- *Input:* (α, w) *, where* α *is a star-free expression over* Σ *and* $w \in \Sigma^*$.
- *Problem: decide if* $w \in L_\alpha$.

We know how to solve this problem: given α, we compute automaton $\mathcal{M}_\alpha$, and then we run $\mathcal{M}_\alpha$ on input w. Let us ask: how do we compute $\mathcal{M}_\alpha$? What does it mean to run $\mathcal{M}_\alpha$?

3.1 Parsing of Star-Free Expressions

Let α be a star-free expression. Let us suppose that α is a very long string. It is not easy to see the construction plan of $\mathcal{M}_\alpha$ that is encoded by this long string. Then, it would be good idea to parse α : it would be a good idea to decompose α into a tree that explicitly shows the constituent parts of α and the way they are connected.

Definition 34 *A parse tree is a pair* $(\mathcal{T}, l)$ *, where:*

1. $\mathcal{T}$ *is a rooted ordered tree. The nodes of* $\mathcal{T}$ *have at most two sons. If* v *has two sons, it has a first child* $f(v)$ *and a next sibling* $n(v)$ *. We use the symbol* r_0 *to denote the root of* $\mathcal{T}$.
2. l *is a labeling function that assigns to each node of* $\mathcal{T}$ *a symbol from the set* $\Sigma \cup \{\cup, \circ, co\}$ *. Function* l *satisfies the following conditions:*
   - $l(v) \in \Sigma$, *if and only if,* v *is a leave.*
   - *If* v *is an inner node with only one son, then* v *get the label co.*

Suppose that $(\mathcal{T}, l)$ is a parse tree. We can read a star-free expression $\beta(\mathcal{T}, l)$ from this data structure. Expression $\beta(\mathcal{T}, l)$ can be easily read when one goes from the leaves to the root while using the following protocol:

- Suppose that v is an inner node and suppose that it has two children $f(v)$ and $n(v)$. Suppose also that $\beta_{f(v)}$ and $\beta_{n(v)}$ are the star-free expressions that we have read at these nodes (coming from the leaves). Let $\otimes \in \{\cup, \circ\}$ be the label of v. We have that $\left(\beta_{f(v)} \otimes \beta_{n(v)}\right)$ is the expression that we read at v.

- Suppose that v is an inner node and suppose that it has a single child $s(v)$. Suppose also that $\beta_{s(v)}$ is the star-free expression that we have read at $s(v)$. The label of v is equal to *co* and we have that $\overline{\beta_{f(v)}}$ is the expression that we read at v.

- $\beta(\mathcal{T}, l)$ is the expression that we read at the root of $\mathcal{T}$.

We can use the same *Bottom-Up* protocol to read from $(\mathcal{T}, l)$ an automaton $\mathcal{M}_{\beta(\mathcal{T},l)}$ that accepts the language $L_{\beta(\mathcal{T},l)}$. This time we assign to the leaves with label ε the automaton $\mathcal{M}_{\varepsilon}$, and we assign to the leaves with label a the automaton $\mathcal{M}_a$ (where $a \in \Sigma$). At the inner nodes we apply the constructions that allow us to get rid of the star-free operations. We get that any parse tree can be easily transformed either in a star-free expression $\beta(\mathcal{T}, l)$ or in a DFA that recognizes the language $L_{\beta(\mathcal{T},l)}$. Moreover, we have:

Theorem 35 *Given a star-free expression α, we can compute in polynomial time a parse tree $(\mathcal{T}_\alpha, l_\alpha)$ such that $\beta(\mathcal{T}_\alpha, l_\alpha)$ is equal to α.*

We can use the CYK parser to compute, on input α, the parse tree $(\mathcal{T}_\alpha, l_\alpha)$. We will study this algorithm in future lectures where we will studying context-free grammars and the parsing problem for context-free grammars. Thus, we can give for sure the existence of a good parser for star-free expressions. It is not hard to imagine such an algorithm, which can be completely based on the fact that parentheses were introduced to mark the begin and the end of the different subexpressions that occur in a star-free expression.

Exercise 36 *Write down a detailed description of the above algorithm.*

We use the symbol CYK-SF to denote the parser we use to compute parse trees of star-free expressions. Suppose we are given a regular expression α, and suppose we are asked to construct a DFA accepting this language. We can:

1. Run CYK on α and get the parse tree $(\mathcal{T}_\alpha, l_\alpha)$.

2. Run the aforementioned Bottom-Up protocol and compute an automaton $\mathcal{M}_{\beta((\mathcal{T}_\alpha, l_\alpha))}$.

How do we compute $\mathcal{M}_\alpha$ from α? Now, we have a good answer to this question. Thus, let us consider the second question: how do we run $\mathcal{M}_\alpha$.

3.2 Running DFA's Using Three Pebbles

Pebbles are computational resources that can be used to solve problems. We show that we can use three pebbles to search the transition tables of DFA's and execute the programs consigned in those tables. To do the latter we construct a three-pebble automaton $\mathcal{U}_3$ which, on input $(\mathcal{M}, w)$, simulates the computation of $\mathcal{M}$ on w and correctly decide if w belongs to $L(\mathcal{M})$.

3.2.1 Encoding of DFA's

We have to observe that $\mathcal{U}_3$ can get as inputs strings and no more than strings. Thus, we have to define an encoding of DFAs as strings over a suitable alphabet. We use alphabet $\Sigma_{bin} = \{0,1\}$.

Let $\mathcal{M} = (Q, \Sigma, q_0, A, \delta)$. We can suppose that

$$Q = \{1, ..., n\}, q_0 = 1 \text{ and } A = \{m, ..., n\} \text{ for some } m \leq n$$

We also suppose that $\Sigma = \{1, ..., l\}$. We associate to $\mathcal{M}$ a string $w_{\mathcal{M}} \in \Sigma_0^*$ that provides a full description of automaton $\mathcal{M}$. Given $i \in \mathbb{N}$, we use the symbol $\widehat{i}$ to denote the unary string $1^i = \underbrace{1 \cdots 1}_{i \text{ times}}$. String $w_{\mathcal{M}}$ is a suitable concatenation of the following four strings:

1. The string
$$w_{state} = 101101110 \cdots 01^n$$
2. The string
$$w_{alphabet} = 10110 \cdots 01^l.$$
3. The string
$$w_{accepting} = 1^m 01^{m+1} 0 \cdots 01^n.$$
4. The string w_δ that describes the transition function δ. Notice that δ can be described by its table, which is a finite list of triples of the form (p, a, q), where $p, q \in \{1, ..., n\}$ and $a \in \{1, ..., l\}$. The triple (p, a, q) can be described by the string $1^p 01^a 01^q$. If δ were equal to
$$(p_1, a_1, q_1), ..., (p_s, a_s, q_s),$$
we would set
$$w_\delta = 1^{p_1} 01^{a_1} 01^{q_1} 001^{p_2} 01^{a_2} 01^{q_2} 00 \cdots 001^{p_s} 01^{a_s} 01^{q_s}.$$

We set
$$w_{\mathcal{M}} = w_{state} 000 w_{alphabet} 000 w_{accepting} 000 w_\delta.$$

We observe that $\mathcal{M}$ can be fully reconstructed from the string $w_{\mathcal{M}}$. We have to observe also that $w_{\mathcal{M}}$ can be easily computed from $\mathcal{M}$. Thus, given a star-free expression α, we can:

1. Compute the parse tree $(\mathcal{T}_\alpha, l_\alpha)$.
2. Compute the automaton $\mathcal{M}_{\beta((\mathcal{T}_\alpha, l_\alpha))}$.
3. Compute the code $w_{\mathcal{M}_{\beta((\mathcal{T}_\alpha, l_\alpha))}}$.

3.2.2 The construction of $\mathcal{U}_3$

Notation 37 *Let $u \in \{1, ..., l\}^k$, we use the symbol $[u]$ to denote the string*

$$1^{u[1]}01^{u[2]}0 \cdots 01^{u[k]} \in \Sigma_0^*.$$

Let $\mathcal{M}$ be a DFA, and let w be an input of $\mathcal{M}$. We construct a DFA that execute the code of $\mathcal{M}$ on the code of u. Let us present a high-level description of this automaton. Thus, suppose that the input string is equal to

$$w_{state}000w_{alphabet}000w_{accepting}000w_{\delta}0000\,[u]\,.$$

The computation of $\mathcal{U}_3$, on this input, is divided in the following phases:

1. **The automaton places its pebbles.**

   - $\mathcal{U}_3$ places a pebble over the substring $10110 \cdots 01^n$. This pebble is used to keep track of the inner state of $\mathcal{M}$, and is placed at the end of this first phase over the left end of the above string. We use the term *state-pebble* to designate this pebble.
   - $\mathcal{U}_3$ places a second pebble over the substring w_δ. This pebble is used to search the table of δ. This pebble is placed on the left end of this substring. We use the term *search-pebble* to designate this pebble.
   - $\mathcal{U}_3$ places its third pebble over the substring $[u]$. This pebble is used to simulate the one-way movement of the scanning head. We use the term *scanning-pebble* to designate this pebble.

2. **The automaton uses its pebbles.**

   Let us suppose that

   $$\widehat{\delta}\,(1, u\,[1, ..., i]) = k \text{ and } \delta\,(k, u\,[i+1]) = r.$$

   Let us also suppose that the state-pebble of $\mathcal{U}_3$ is placed over the substring 1^k that represents state k, while the scanning-pebble is placed over the substring the encodes the i-th character u. This pebble configuration suitably represents the configuration reached by $\mathcal{M}$ after i transitions. Thus, we are assuming that the simulation has worked well until this point. We can suppose that the inner state of $\mathcal{U}_3$ indicates that the simulation of the i-th transition of $\mathcal{M}$ has just ended, and that it is time to begin with the simulation of the $(i+1)$-step. Then, the automaton moves the scanning pebble to the substring that encodes the $(i+1)$-character of u. Suppose that this character is equal to a. Notice that $\mathcal{U}_3$ can use the search-pebble and the scanning-pebble to recognize the substrings of w_δ that encode triples of the form $(-, a, -)$. Notice also that $\mathcal{U}_3$ can use the search-pebble and the state-pebble to recognize the substrings of w_δ that encode triples of the form $(q, -, -)$. Then, $\mathcal{U}_3$ can use its three pebbles to locate the substring of w_δ that encodes the unique triple in δ that looks like $(q, a, -)$. Suppose that the substring $1^p 0 1^a 0 1^q$ has been located. Notice that $\mathcal{U}_3$ can use the search-pebble and the state-pebble to place the latter pebble over the substring 1^p that represents state p. Notice that, by doing so, $\mathcal{U}_3$ has simulated a further step of the computation of $\mathcal{M}$ on the input u.

Remark 38 *The language Squares has been intensively studied along the historical development of automata theory. We claim that there are good reasons for this prominence of Squares, the most important of which is the following one:*

The ability of recognizing squares is closely related to the ability of searching tables and executing the programs that are consigned in those tables.

Note that $\mathcal{U}_3$ can search for the right triple, and then for the right state, only because this automaton can use pairs of pebbles to identify pairs of substrings that are identical. $\mathcal{U}_3$ uses those pebbles to identify the right substrings using a zig-zag strategy that is analogous to the siz-zag strategies that are used to recognize squares: recognizing squares is a prerequisite for the emergence of programmability.

Exercise 39 ***k-head automata** are two-way automata provided with k scanning heads. Prove that k-head automata can be simulated by k-pebble automata. Prove that k-pebble automata can be simulated by $k+1$-head automata. Does there exist a language L_k that can be recognized with k pebbles but which cannot be recognized with k heads?*

Exercise 40 ***Counters** are data structures that are used to save integer values encoded in unary. We can imagine a one-counter automaton as a pebble automaton provided with infinite many pebbles which are used by the automaton to build a stack (counter). Each time the automaton scans a character it also looks at the counter and then it decides what to do with its inner state, with its trajectory over the tape and with the counter. A bounded counter is a counter whose height cannot be greater than the length of the input. Bounded counters can be simulated by pebbles, and pebbles can be simulated by bounded counters. Prove that k-counter automata can be simulated by $k+1$-pebble automata, and prove that k-pebble automata can be simulated by k-counter automata. Does there exist a language T_k that can be recognized with k bounded counters but which cannot be recognized with k heads?*

References

[1] T. Koetsier. On the prehistory of programmable machines: musical automata, looms, calculators. Mechanism and Machine Theory, Elsevier, 36 (5): 589–603, 2001.

[2] M. Ben-Ari. Understanding Programming Languages. John Wiley and Sons, NY, 1996

4 A Toy Model of Computation: Star-Free as a Programming Language

We use star-free expressions to define recognition (pattern matching) problems, and we also use those regular expressions to construct the algorithmic solutions to those problems: each regular expression α encodes an automaton $\mathcal{M}_{\alpha}$ that can be constructed from α in an automatic way. We claim that the language constituted by all the star-free expressions constitutes a toy-model of programming language. We use the symbol SF to denote this language.

We cannot use SF to solve all the problems that can be solved by computers (SF is not Turing complete, see below). However, we can use SF to solve all the pattern matching problems that can be solved by aperiodic (also called counter-free) automata. We claim that SF is a programming language because it allows us to communicate with computing devices in order to solve problems. We are using SF to communicate with the tripe (CYK-SF,Bottom-Up,$\mathcal{U}_3$) . Moreover, we have:

1. SF has a precise grammar, which can be used to recognize the correct expressions in this language. We explore this grammar in next section.
2. Each expression α has a functional meaning, the algorithm $\mathcal{M}_{\alpha}$ encoded by α that solves the problem L_{α}. The meaning $\mathcal{M}_{\alpha}$ can be effectively computed from α with the aid of the formal grammar behind SF.

4.1 The Formal Grammar of Star-Free Expressions

Linguists describe the grammars of natural languages in terms of their block structure. This block structure describes how sentences are recursively built up from smaller phrases, and eventually individual words or word elements. *Context-free grammars* provide a simple and mathematically precise mechanism for describing the methods by which phrases are built from smaller blocks. The simplicity of context-free grammars makes the formalism amenable to rigorous mathematical study. On the other hand, the expressive power of context-free grammars allows them to describe the basic recursive structure of sentences, the way in which clauses nest inside other clauses, and the way in which lists of adjectives and adverbs are swallowed by nouns and verbs

Definition 41 *A context-free grammar is a tuple $G = (V, T, S, R)$, where:*

1. *V is a finite set of nonterminal symbols.*
2. *T is a finite set of terminal symbols.*
3. *$S \in V$ is the initial variable.*
4. *R is a finite set of production rules. A production rule is a expression like $X \to \alpha$, where $X \in V$ and $\alpha \in (V \cup T)^*$.*

The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky. In Chomsky's generative grammar framework the syntax of natural language was described by context-free rules combined with transformation rules.

Let us consider an artificial language as it is the language of propositional logic. Let us fix the set $\{p_1, ..., p_n\}$ as the set of propositional letters. Let $G_{PL} = (V, T, S, R)$ be the context-free grammar given by:

1. The set V is equal to $\{S\}$.
2. The set T is equal to $\{\wedge, \vee, \neg, (,), p_1, ..., p_n\}$.
3. The set R is constituted by the following production rules:

$$S \rightarrow (\neg S) \mid (S \vee S) \mid (S \wedge S) \mid p_1 \mid \cdots \mid p_n$$

Definition 42 *Let $G = (V, T, S, R)$ be a context-free grammar and let $w \in T^*$. A derivation of w is a finite sequence $\alpha_1, ..., \alpha_n$ such that:*

1. *For all $i \leq n$ it happens that $\alpha_i \in (V \cup T)^*$.*
2. *$\alpha_1 = S$ and $\alpha_n = w$.*
3. *For all $i \leq n-1$ it happens that α_i can be obtained from α_{i-1} by applying one of the production rules in R.*

Remark 43 *Let $w_1 \cdots w_n \in (V \cup T)^*$, suppose that $w_i = X$ and suppose that $X \rightarrow \alpha$ is a production rule in R. The string*

$$w_1 \cdots w_{i-1} \alpha w_{i+1} \cdots w_n$$

can be obtained from $w_1 \cdots w_n$ after applying the production rule $X \rightarrow \alpha$.

We claim that G_{PL} correctly describes the syntax of propositional logic, that is: a string $\alpha \in T^*$ can be derived in G_{PL}, if and only if, the string α corresponds to a well formed formula of propositional logic. Consider the formula $((p_1) \vee (p_2))$. The sequence:

1. S,
2. X,
3. $((X)\, con\, (X))$,
4. $((var)\, con\, (X))$,
5. $((var)\, con\, (var))$,
6. $((var) \vee (var))$,
7. $((p_1) \vee (var))$,

8. $((p_1) \vee (p_2))$.

Is a derivation of this formula.

Definition 44 *Let G be a grammar, we use the symbol $L(G)$ to denote the set of strings that can be derived in G. We say that $L(G)$ is the language generated by G.*

Exercise 45 *Let pal be the language of binary palindromes, construct a context-free grammar that generates this language.*

Exercise 46 *The elementary functions are the functions of calculus, which are constructed from the monomials, the trigonometric functions, the exponential and the logarithmic functions using sums, products, divisions and compositions. Construct a context-free grammar $G_{Calculus}$ for the language that is constituted by all those functions.*

Let $\Sigma = \{a_1, ..., a_n\}$. The set of star-free expressions over Σ can be defined by a context-free grammar that we denote with the symbol Star-Free$_\Sigma$. This context-free grammar is equal to

$$\text{Star-Free}_\Sigma = (\{S\}, S, \Sigma \cup \{\cup, \circ, , (,)\}, R),$$

and the production rules in R are the production rules in the below list

$$S \to (S \cup S) \mid (S \circ S) \mid (\overline{S}) \mid a_1 \mid \cdots \mid a_n \mid \varepsilon$$

Remark 47 *Block structure was introduced into computer programming by the Algol project (1957–1960), which, as a consequence, also featured a context-free grammar to describe the resulting Algol syntax. This became a standard feature of computer languages. The "block structure" aspect that context-free grammars capture is so fundamental to grammar that the terms syntax and grammar are often identified with context-free grammar rules, especially in computer science. Formal constraints not captured by the grammar are then considered to be part of the "semantics" of the language.*

4.1.1 Parse Trees

Generative grammars are important for linguistic analysis because those grammars allow the construction of parse trees. A parse tree decomposes a sentence into its constituent components and shows the way those components are integrated into the sentence.

Definition 48 *Let $w \in L(G)$. A parse tree for w is a labeled tree $(\mathcal{T}, r, L)$, that satisfies:*

1. *$(\mathcal{T}, r)$ is a rooted ordered tree, and r is the root of $\mathcal{T}$.*
2. *L is a labeling function that assigns to each node of $\mathcal{T}$ an element of $(V \cup T)^*$.*

3. *$L(r) = S$.*

4. *The labels assigned to the leaves of $\mathcal{T}$ belong to the set T of terminal symbols.*

5. *If we read the leaves from left to right, we read the string w.*

6. *The labels assigned to the inner nodes of $\mathcal{T}$ belong to V.*

7. *Suppose that v is a inner node, and suppose that $v_1, ..., v_n$ are its sons, ordered from left to right. Suppose also that the equalities*

$$L(v) = X, L(v_1) = X_1, ..., L(v_n) = X_n$$

hold. Then the production rule $X \rightarrow X_1 \cdots X_n$ is a production rule in R.

Exercise 49 *Let α be the propositional formula $((p_1 \wedge p_2) \vee (\neg (p_3 \wedge (\neg p_2))))$. Construct a parse tree for this formula.*

Exercise 50 *Use the grammar $G_{Calculus}$ to construct an algorithm for the (symbolic) differentiation of elementary functions (suppose that you can construct parse trees). Can you use the same grammar for symbolic integration?*

Exercise 51 *The grammar Star-Free$_\Sigma$ is an unambiguous grammar. This means that given a regular expression α there exists only one derivation tree for α in Star-Free$_\Sigma$. Prove that Star-Free is in effect an unambiguous grammar.*

Let α be a star-free expression. The grammar Star-Free$_\Sigma$ produces exactly one parse tree for α. This parse tree is equal to the parse tree that we have learnt to compute in previous lectures. We conclude that SF has a formal grammar that is analogous to the grammar of most programming languages. This grammar rules the syntax of star-free expressions, but very much more important that this:

The grammar Star-Free$_\Sigma$ dictates the functional meaning of each one of the star-free expressions over Σ. We claim this based on the following fact: Star-Free$_\Sigma$ determines the parse tree $((\mathcal{T}_\alpha, l_\alpha))$, and this parse tree encodes the automaton $\mathcal{M}_{\beta(\mathcal{T}_\alpha, l_\alpha)}$ that accepts the language L_α and solves the corresponding pattern matching problem.

4.2 The Computing System: Can It Be Done With Pebbles Alone?

We developed a toy-model of a computing system. This computing system is constituted by three algorithms that we call CYK-SF, Bottom-Up and $\mathcal{U}_3$. Algorithm CYK-SF receives as input a pair (α, u) and computes $((\mathcal{T}_\alpha, l_\alpha), u)$. Algorithm Bottom-Up receives as input the pair $((\mathcal{T}_\alpha, l_\alpha), u)$, uses the Bottom-Up protocol to compute $\mathcal{M}_{\beta(\mathcal{T}_\alpha, l_\alpha)}$, and hence it computes the code of u and the code of $\mathcal{M}_{\beta(\mathcal{T}_\alpha, l_\alpha)}$. Algorithm $\mathcal{U}_3$ receives as input those two codes and simulates the computation of $\mathcal{M}_{\beta(\mathcal{T}_\alpha, l_\alpha)}$ on input u. This simulation allows us to correctly decide whether the program α accepts the input α.

Real-world systems have an analogous structure. Those systems are constituted by a parser, a compiler and a simulator. The parser receives as input a program, the compiler computes an executable, and the simulator (an Universal Turing Machine) executes the latter program. The universal Turing Machine mentioned before can run the parser and the compiler as well as the simulator. Thus, all the execution occurs at the interior of a single machine (we do not need more tha one machine at home). We have, on the other hand, that the rustic automaton $\mathcal{U}_3$ cannot run the parser and the compiler. We ask: can the processing of star-free expressions be done by pebble automata without extra aid? The latter question is equivalent to ask whether The Membership Problem for star-free expressions can be solved by pebble automata. We study this question in the second part of this booklet. This question is a typical lower-bound question: can problem L be solved using the resources in the class X? Lower-bound questions use to be hard, and most of them force us to construct (complexity) theory.

References

[1] S. Ginsburg, S. Greibach, M. Harrison. Stack Automata and Compiling. J. ACM. 14 (1): 172 - 201, 1967.

5 Unbounded Pebbles and Turing Machines

We can solve many problems using pebbles. However, it seems that there are problems that cannot be solved with pebbles alone. We claim that we can solve most of those latter problems if we provide pebble automata with unbounded many pebbles. Those automata can use their unbounded provision of pebbles to simulate counters, pushdown stacks and Turing machines. However, we wont explore this alternative since unbounded many pebbles could be a polemic resource. Instead of this we will experience for the first time a phase transition in the evolution of hardware: we will begin to use ink, and we will begin to use the full power of writing.

5.1 Turing Machines

A *one head Turing* machine is a machine provided with one scanning head which can explore and use all the cells of the input tape. Moreover, this scanning head can write on its input tape.

Definition 52 *A one head one-tape Turing machine is a tuple*

$$\mathcal{M} = (Q, q_0, \Sigma, \Gamma, H, A, \delta),$$

where:

1. *Q is a finite set of states.*
2. *$q_0 \in Q$ is the initial state*
3. *Σ is a finite set, the input alphabet of $\mathcal{M}$.*
4. *Γ is a finite set, the work-alphabet of $\mathcal{M}$, and we have that $\Sigma \cup \{\square\} \subseteq \Gamma$.*
5. *$A \subseteq H \subset Q$, where H is the set of halting states and A is the set of accepting states.*
6. *The transition function δ is a function from the set $Q \times \Gamma$ into the set $Q \times \Gamma \times \{-1, 0, 1\}$. The equality*

   $$\delta(q, a) = (p, b, \delta,)$$

   means that: if the inner state is equal to q and the scanning head is reading the letter $a \in \Gamma$, then the machine executes the order encoded by $(p, b, \delta,)$, that is: $\mathcal{M}$ switches its inner state from q to p, deletes a and writes b, and then moves in the direction indicated by δ.

Turing machines are very similar to pebble automata. Those machines have a tape and a scanning head that can move in both directions along the tape. The main differences are the following two:

1. Turing machines can use all the cells of the tape to write. Those machines are provided with unbounded many erasable ink drops.

2. Pebble automata cannot use the cells of the tape that are to the right of the input. Turing machines can use all the cells of the input tape.

The architecture of Turing machines is completely elementary. It is natural to conjecture that there exist more powerful architectures. However, there does not exist a known model of computation more powerful than Turing machines. This motivates us to accept the following thesis.

Question 53 ***The Church-Turing Thesis***
If a problem L can be solved by algorithms, then problem L can be solved by Turing machines.

The above thesis indicates that the notions of algorithm and Turing machine are one and the same.

5.2 A Realistic Model of Computing

From now on we assume The Church-Turing Thesis. Then, if we are given an algorithmic problem L, we look for a Turing machine $\mathcal{M}_L$ that solves this problem. Suppose we can construct such a machine. What can we do with this theoretical construction? We should execute this algorithm. To do the latter we need an universal object analogous to $\mathcal{U}_3$. Turing proved that such a construction is possible, that is: Turing proved that there exists a Turing machine $\mathcal{U}$ that can simulate any other Turing machine. Machine $\mathcal{U}$ receives pairs, each pair constituted by the code of a Turing machine and the code of an input. Given one of those pairs, say $\left(\widehat{\mathcal{M}}, \widehat{u}\right)$, machine $\mathcal{U}$ simulates the computation of $\mathcal{M}$ on input u. We have that $\mathcal{U}$ is a programmable machine: we can fed $\mathcal{U}$ with program $\widehat{\mathcal{M}}$ and input $\widehat{u}$, and then execute $\widehat{\mathcal{M}}$ on $\widehat{u}$. However, we cannot ask a programmer to work this way, because the language of Turing machines is one of the most unfriendly languages on the market. If we want to hire some programmers to work with $\mathcal{U}$ we should think in creating a language that allows those programmers to communicate with the machine in a more relaxed way. This programming language could be constituted, among other things, by a formal grammar G ruling the syntax of the language. This grammar should encode a parser/compiler, which, on input α, computes a parse tree and hence the code of the Turing machine that is encoded by α. Finally, the language must be as friendly as possible. Hard work! Fortunately, there are many programming languages on the market that meet these requirements. We will have the chance of discussing some additional facts about programming languages that are Turing-complete.

5.2.1 An Universal Turing Machine

Suppose we want to construct a machine that simulates one-pebble automata. We can use a slight modification of $\mathcal{U}_3$ that includes one additional pebble. This additional marker is used to track the movement of the pebble (of the automaton that is being simulated). We use the symbol $\mathcal{U}_4$ to denote this universal object, and the symbol $\mathcal{U}_{k+3}$ to denote the automaton that we would construct this way in order to simulate k-pebble automata. We can go a step further and construct an automaton provided with infinite many pebbles that are used to track all the objects that have to be tracked along the simulation work. We use the symbol $\mathcal{U}_\infty$ to denote this machine. Observe that $\mathcal{U}_\infty$ can simulate all the pebble automata with a finite number of pebbles as well as pebble automata with infinite many pebbles. The machines that cannot be simulated by $\mathcal{U}_\infty$ are the machines that can explore and use all the cells of their input tape. Let us replace pebbles by erasable ink, and let $\mathcal{U}_\infty$ use all its tape. We get an universal Turing machine $\mathcal{U}$.

The construction of $\mathcal{U}$ is then analogous to the construction of $\mathcal{U}_3$. Turing machines can be described by binary strings similar to the binary strings that are used to describe DFA's. Machine $\mathcal{U}$ receives as input one of the former strings, say the code of $\mathcal{M}$, as well as the code of u, and then simulates the computation of $\mathcal{M}$ on u. The simulation proceeds as it proceeds the simulation of DFA's that is performed by $\mathcal{U}_3$:

Machine $\mathcal{U}$ uses its marking(overwriting) power to track the objects that must be tracked along the computation of $\mathcal{M}$. Machine $\mathcal{U}$ tracks the movement of the input head of $\mathcal{M}$, it tracks the changes in the inner state of $\mathcal{M}$, and it also registers on its tape the changes that are performed on the input tape of $\mathcal{M}$.

Exercise 54 *It is an excellent exercise for the reader to fill into the details of this construction. The interested reader can consult the reference [2].*

5.3 Final Term Project

Implement a simulator of the computing system developed to recognize star-free languages. This simulator should allow the visualization of the whole computing process.

References

[1] A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society. 42(2): 230 - 265, 1937.

[2] R. Herken. *The Universal Turing Machine – A Half-Century Survey.* Springer Verlag, Berlin 1995.

6 The Undecidable

We finish this first part of the booklet discussing the existence of problems that cannot be solved by computers. The existence of noncomputable problems cannot come as a surprise. If we assume the Church-Turing Thesis we get that the set of available algorithms is a countable set: any Turing machine can be fully described by a binary string. On the other hand, the set of problems to be solved is equipotent with $Pow(\Sigma^*)$. We can use Cantor's diagonal argument to show that the latter set is very much larger than the former and that we cannot assign an algorithm to any problem. We can use Cantor's diagonal argument and get a more clear picture of the undecidable.

6.1 Turing's Diagonal Argument

Let $\mathcal{U}$ be an universal Turing machine. The code $\widehat{\mathcal{U}}$ is a binary string, and the inputs of $\mathcal{U}$ are binary strings. Those two facts allows us to fed $\mathcal{U}$ with its own code. This self-reference places us dangerously close to Godel's Horizon. Let us set

$$L_{acc} = \left\{ \widehat{\mathcal{M}} : \mathcal{M} \text{ rejects its own code} \right\}.$$

Language L_{acc} is a dangerous language overflowing self-reference. We claim that L cannot be recognized by Turing machines.

Theorem 55 *The language L_{acc} is undecidable*

Proof. Suppose that L_{acc} is decidable and let $\mathcal{N}$ be a Turing machine that decides L_{acc}. Let us ask what happens when we fed $\mathcal{N}$ with its own code. We get

$$\mathcal{N} \text{ accepts } \widehat{\mathcal{N}} \text{ if and only if } \mathcal{N} \text{ rejects } \widehat{\mathcal{N}}.$$

Thus, we get a contradiction that resembles the contradiction in Russell's Paradox. The theorem is proved. ∎

The above theorem gives us a concrete example of an undecidable language. We can use this language to prove that many other languages are also undecidable. To this end we use the following basic principle:

Suppose we can prove that any algorithmic solution for problem P can be transformed into an algorithmic solution for L. Then, if problem L is undecidable the problem P has to be undecidable.

We can capture this principle with the notion of algorithmic reduction.

Definition 56 *Let R, S be two languages, we say that R is reducible to S, if and only if, there exists an algorithm $\mathcal{A}$ which, on input w, computes a string $A(w)$ such that*

$$w \in R, \textit{ if and only if, } A(w) \in S.$$

We have:

Proposition 57 *Let R, S be two languages.*

1. *If L_{acc} is reducible to R the problem R is undecidable.*
2. *If R is undecidable and R is reducible to S then S is also undecidable.*

Exercise 58 *Let*

$$L_{HALT} = \left\{ \widehat{\mathcal{M}} : \mathcal{M} \textit{ halts on the empty input} \right\}.$$

Prove that L_{HALT} is undecidable.

Exercise 59 *Let*

$$L_{ACC} = \left\{ \widehat{\mathcal{M}} : \mathcal{M} \textit{ accepts at least one input} \right\}.$$

Prove that L_{ACC} is undecidable.

Exercise 60 ***Rice's Theorem*** *states that undecidability is almost everywhere. Let $\mathcal{L}$ be the set of all the binary languages that are decidable, and let P be a subset of $\mathcal{L}$. We set*

$$L_P = \left\{ \widehat{\mathcal{M}} : L(\mathcal{M}) \textit{ belongs to } P \right\}.$$

Rice's theorem asserts that L_P is decidable, if and only if, either $P = \mathcal{L}$ or $P = \emptyset$. Prove the theorem.

Exercise 61 *A tiling system is a tuple $\mathcal{T} = (\Sigma, R_H, R_V, t_0)$ such that:*

1. *Σ is a finite set of tiles.*
2. *$R_H, R_V \subset \Sigma^2$ are the horizontal and vertical constraint relations.*
3. *$t_0 \in \Sigma$ is the initial tile.*

Let $\mathcal{T}$ be a tiling system, a $\mathcal{T}$-tiling of $\mathbb{N} \times \mathbb{N}$ is a function $T : \mathbb{N} \times \mathbb{N} \to \nsucc$ such that:

1. *$T(0,0) = t_0$.*
2. *For all $n, m \in \mathbb{N}$ the conditions*

   $$(T(n,m), T(n+1,m)) \in R_H \textit{ and } (T(n,m), T(n,m+1)) \in R_V$$

   hold.

*Let Σ be a finite set of tiles (a finite alphabet). The **Tiling Problem over** Σ is the problem:*

- *Input: $\mathcal{T}$, where $\mathcal{T}$ is a tiling system whose set of tiles is equal to Σ.*
- *problem: decide if there exists a $\mathcal{T}$-tiling of $\mathbb{N} \times \mathbb{N}$.*

Prove that there exists a finite set of tiles Σ such that the tiling problem over Σ is undecidable.

Exercise 62 *Investigate about Post's Correspondence Problem.*

Exercise 63 *Provide a diagonal's argument to show that there exist problems that can be solved with $k+3$ pebbles but which cannot be solved with k pebbles.*

References

[1] M Davis (ed.). *The Undecidable.* Raven Press, Hewlett NY, 1965.

Part II

Complexity Theory: Determinism vs Nondeterminism

7 Nondeterminism I: Rabin-Scott 1959

Regular expressions came to light in 1951 when mathematician Stephen Cole Kleene used them to describe regular languages. Regular expressions became popular around 1968 mainly because of two uses: pattern matching in texts editors and lexical analysis in compilers. Regular expressions, or *regexes*, can be used to represent complex patterns. How complex? As complex as the patterns recognized by DFA's can be.

Definition 64 *Let Σ be a finite alphabet. The set of regular expressions over Σ is inductively defined by:*

1. *ε and $a \in \Sigma$ are basic regular expressions that denote the empty language and the singleton language $\{a\}$, respectively.*

2. *Let α, β be regular expressions, we have that $(\alpha \cup \beta)$ and $(\alpha \circ \beta)$ are regular expressions that denote the languages $L_\alpha \cup L_\beta$ and $L_\alpha \circ L_\beta$.*

3. *Let α be a regular expression, we have that (α^*) is also a regular expression. The language denoted by (α^*) is the language $(L_\alpha)^*$ that is equal to*

$$\{w \in \Sigma^* : \exists i \geq 0 \exists u_1, ..., u_i \, (w = u_1 \circ \cdots \circ u_i \;\&\; u_1, ..., u_i \in L)\}.$$

Theorem 65 ***Kleene's Theorem***
Language $L \subset \Sigma^*$ is regular, if and only if, there exists a regular expression α such that $L = L_\alpha$.

Proof. First, we prove that given a regular expression α the language L_α is regular. We make this proof by induction over the complexity of the regular expression.

We know that the languages denoted by the basic expressions are regular. We know what to do with $\cup$ and $\circ$. Then, it only remains to prove that the set of regular languages is closed under Kleene's star (under the operation $*$). Thus, let L be a regular language and let $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$ be a DFA that recognizes L. We set

$$\mathcal{M}^* = (\Sigma, Pow(Q), \{q_0\}, A^*, \delta^*),$$

where A^* is equal to the set

$$\{B \subset Q : A \cap B \neq \emptyset\},$$

and δ^* is the transition function defined by

$$\delta^*(B, a) = \begin{cases} \{\delta(q, a) : q \in B\}, \text{ if } B \cap A = \emptyset \\ \{\delta(q, a) : q \in B\} \cup \{\delta(q_0, a)\}, \text{ if } B \cap A \neq \emptyset \end{cases}$$

It is easy to check that $\mathcal{M}^*$ recognizes L^*. We get that the set of regular languages is closed under Kleene's star, and we get that any regular expression denotes a regular language.

Now we prove the opposite inclusion, we prove that any regular language is denoted by a regular expression.

Let $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$ be a DFA and let L be the language accepted by $\mathcal{M}$. Automaton $\mathcal{M}$ is suitably represented by its transition digraph. This transition digraph is the labeled digraph $G(\mathcal{M})$ that is defined by:

- $V(G(\mathcal{M})) = Q$.
- Let $p, q \in Q$ and let $a \in \Sigma$. There exists a directed edge (p, q) with label a, if and only if, the triple (p, a, q) belongs to δ.

Notation 66 *We use the symbol (p, α, q) to denote that there exists a directed edge from p to q and with label α.*

The strings in L are in bijective correspondence with the labeled paths that start at q_0 and ends at a node in A. The language L is this set constituted by those labeled paths. The idea is to resume all those paths, with all their combinations of labels, into a single path of length 1 that goes from q_0 to A and whose label is a regular expression. To do this we use a *shrinking* procedure that transforms a labeled digraph with n nodes into a labeled digraph with $n-1$ nodes and which accepts the same language as the former digraph. To achieve this size reduction we use larger (regex) labels. The procedure works as follows:

Let G be a labelled digraph with initial node q_0, target-set A and with edge labels in the set Regex_Σ. Let γ be a path in G, and let $\alpha_1, ..., \alpha_n$ be the ordered sequence of labels that we can read when we walk along this path. We use the symbol L_γ to denote the language

$$L_{\alpha_1} \circ \cdots \circ L_{\alpha_n}$$

We assign to G the language $\bigcup_\gamma L_\gamma$, where the index γ ranges over the paths in G that go from q_0 to A. We say that G accepts this language. It is our task to construct a smaller digraph that has assigned the same language.

Suppose that G contains a node q that does not belong to $A \cup \{q_0\}$. Let $\{(r_i, \alpha_i, q) : i \leq k\}$ be the set of edges that go into q. Let $\{(q, \beta_j, s_j) : j \leq m\}$ be the set of edges that go out of q. And let $\{(q, \gamma_t, q) : t \leq l\}$ be the set of loops that are appended to q. If we delete q, we will have to replace all those edges by a suitable set of edges connecting the nodes that were previously connected thanks those edges. The key step in the shrinking process consists in:

1. Delete q.
2. Delete the set of edges that is constituted by the union of the above three sets. Add the edges in the set

$$\left\{\left(r_i, \alpha_i \circ \gamma_t^* \circ \beta_j, s\right) : i \leq k, j \leq m, t \leq\right\}.$$

After applying this shrinking step we get a smaller digraph that accepts the same language as G. We can use this procedure to eliminate all the states that are not in $A \cup \{q_0\}$. We can use the same ideas to merge all the accepting states into a single state. At the end of the day we get a labeled digraph with two states q_0 and q_A, and a finite set of edges that go from q_0 to q_A. Let

$$\{(q_0, \psi_i, q_A) : i \leq M\}$$

be this set of edges. The regular expression $\bigcup_{i \leq M} \psi_i$ denotes the language accepted by this small digraph, and we get, because of the inductive construction, that this regular expression denotes the language that is accepted by $\mathcal{M}$. The theorem is proved ■

We would like to discuss an important feature of the above proof. Suppose we are given a regular language L, suppose we are given a DFA of size n that recognizes L and suppose that we are asked to construct a DFA that accepts L^*. The construction used in the above proof allows us to build a DFA of size 2^n that recognizes L^*. This blow-up, from n to 2^n is large (it is an exponential blow-up) but it could be even more dramatic if we think in the following: suppose we are given a regular expression α and a DFA of size $m \geq 2$ that recognizes L_α. Suppose that we are asked to compute a DFA that recognizes the language L_β with $\beta = \alpha^{\overbrace{* \cdots *}^{n\text{-times}}}$. If we recursively apply the above procedure we get a DFA whose size is greater than $T(2, n)$, where $T(2, n)$ is the exponential tower of height n. We get that the output of the inductive algorithm that is implicit in the above proof is a galactic output whose size cannot be bounded by an exponential tower. We have to conclude that we do not have, not yet, a feasible procedure for compiling DFA's from regular expressions.

How do we deal with the concatenations that occur in star-free expressions? We use a power construction similar to the power construction used to deal with $*$. The former power construction yields an exponential blow-up that becomes even more dramatic if we think in the following: suppose we are given a star-free expression α, and a DFA of size $m \geq 2$ that recognizes L_α. Suppose that we are asked to compute a DFA that recognizes the language L_β with $\beta = \underbrace{\alpha \circ \cdots \circ \alpha}_{n\text{-times}}$.

If we recursively apply the aforementioned power construction we get a DFA whose size is greater than $T(2, n)$. We get that the output of the Bottom-Up algorithm used to compile DFA's is a galactic output whose size cannot be bounded by an exponential tower. We have to conclude that we do not have, not yet, a feasible procedure for compiling star-free expressions, and we have to accept a harsh conclusion: we have not put our toy-model to work yet.

Exercise 67 *Let $L \subset \{1\}^*$, prove that L^* is a regular language.*

Exercise 68 *Prove that the set of regular languages over $\{1\}$ is equal to the set of star-free languages over $\{1\}$.*

7.1 Nondeterministic Finite State Automata

The conception of nondeterministic computation is usually attributed to Michael Rabin and Dana Scott. They defined nondeterministic finite automata in their famous 1959 paper *Finite Automata and Their Decision Problems* [2].

Definition 69 *A nondeterministic finite state automaton (NFA, for short)* $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$ *is like a DFA except that the transition function* δ *goes from* $Q \times \Sigma$ *into* $Pow\left(Q\right)$.

The differences between DFA's and NFA's seem to be few and irrelevant. However, those differences entail a deep breach with the naive notion of computation. According to this notion the actual configuration must determine a unique possible next configuration, and the initial configuration (the input) must determine a single possible output. Nondeterministic computations can have several different outcomes on the same input.

Let $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$ be a NFA and let w be an input of $\mathcal{M}$. The computation of $\mathcal{M}$, on input w, advances thanks to the sequential evaluation of the function δ. Each time $\mathcal{M}$ gets an equality like

$$\delta\left(q, a\right) = \left\{p_1, ..., p_k\right\},$$

the automaton nondeterministically chooses one state out of those k possible options. The choice of $\mathcal{M}$ determines the set of computational paths the can be continued from this time instant. Some of those computation paths lead to rejecting states, and some other (depending on w) lead to accepting states. We say that L accepts the language

$$L\left(\mathcal{M}\right) = \left\{w : \text{there exists an accepting computation of } \mathcal{M} \text{ on } w\right\}.$$

Notice that $\mathcal{M}$ unambiguously rejects the strings that do not belong to $L\left(\mathcal{M}\right)$. Notice also that $\mathcal{M}$ ambiguously accepts the strings that belong to $L\left(\mathcal{M}\right)$: suppose that $w \in L\left(\mathcal{M}\right)$, we have that there could be rejecting computations, (and hence erroneous computations), of $\mathcal{M}$ on input w. Thus, NFA's seem to be like DFA's but with the ability of making some errors. We have to notice that:

The acceptance condition of NFA's implies that those automata do not make errors. We are assuming, with this definition, that $\mathcal{M}$ *sees* all of its computations on input w. We get that $\mathcal{M}$ correctly decides whether there exists an accepting computation within the, possibly very large set, of its computations on input w. We are the ones who make errors when using a NFA, because we are the ones who can see a single computation, out of many possible, each time we run a nondeterministic automaton on one of its inputs.

7.2 NFA's Cannot Recognize Nonregular Languages

Theorem 70 *The set of languages accepted by NFA's is the set of regular languages.*

Proof. DFA's are a special type of NFA's. We get that any regular language is accepted by a suitable NFA. It remains to prove that the languages accepted by NFA's are regular languages.

Let $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$ be a NFA, we prove that we can construct a DFA $\mathcal{M}^{\varpi}$ that accepts the language $L(\mathcal{M})$. The basic idea behind the construction of $\mathcal{M}^{\varpi}$ is to build a DFA that tracks all the computations of $\mathcal{M}$ on its different inputs. This seems to be unfeasible with a finite set of states:

Suppose $\mathcal{M}^{\varpi}$ is processing input w. Suppose it is the i-th instant of the computation, and suppose that the inner state of $\mathcal{M}^{\varpi}$, say the state q_i, describes in a proper way the evolution of all the runs of $\mathcal{M}$ on the prefix $w[1,...,i]$. So far so good. Then, automaton $\mathcal{M}^{\varpi}$ scans the next character. It could happens that character $w[i+1]$ forces each run of $\mathcal{M}$ to bifurcate. We get that the number of runs of $\mathcal{M}$ on $w[1,...,i+1]$ is twice the number of its runs on $w[1,...,i]$. This doubling can keep occurring, and we get that the number of runs to be tracked grows at exponential speed with i. We have to ask: how can we use a finite set of states to track a growing sheaf of computations that grows at exponential speed? The answer is that we have to focus on the relevant information. Notice that we do not need to describe the computational paths so far evolved one by one and configuration by configuration. It suffices if we describe the set of states that are reached at instant i. This set of states is simply a subset of Q. Thus, it suffices if we use $Pow(Q)$ to track the computations of $\mathcal{M}$ on the prefixes of w. Suppose that q_i is equal to $A_i \subset Q$. Let q be a state in A_i. State q is the last state reached by a subset of the runs so far evolved. Those runs will evolve exactly the same with the scanning of $w[i+1]$ because the dynamics of DFA's is Markovian: the state that was reached in the precedent instant is the only state that matters. Then, it suffices if we track the evolution of state q, and any other state in A_i, under the action of $w[i+1]$. Set

$$\mathcal{M}^{\varpi} = (\Sigma, Pow(Q), \{q_0\}, A^{\varpi}, \delta^{\varpi}),$$

where A^{ϖ} is equal to the set

$$\{B \subset Q : A \cap B \neq \emptyset\},$$

and δ^{ϖ} is the transition function defined by

$$\delta^{*}(B, a) = \{p : \exists q (q \in B \;\&\; (q, a, p) \in \delta)\}.$$

It is easy to check that $\mathcal{M}^{\varpi}$ accepts L. The theorem is proved. ■

We have to observe that the size of $\mathcal{M}^{\varpi}$ is exponential in the size of $\mathcal{M}$. Thus, we can eliminate nondeterminism, but this elimination has an exponential cost that is unavoidable.

Exercise 71 *Let L_k be the language*

$$\left\{w \in \{0,1\}^{*} : w[|w| - k + 1] = 0\right\}.$$

Prove that L_k can be recognized by a NFA with k states. Prove that a DFA recognizing L_k has no less than 2^k states (Hint: investigate about Myhill-Nerode Theorem). Conclude that the exponential blow-up produced by the above power construction is unavoidable.

7.3 NFA's and State Complexity

It can be argued that the very idea of nondeterministic computation is a bizarre idea. We proved that NFA's cannot recognize nonregular languages. We could (wrongly) conclude that the aforementioned bizarre idea does not provide computational advantages. We have to ask: how could this seemingly useless bizarre idea survive the test of the time? The main reason behind the survival of NFA's is that those automata provide us with important computational advantages. Let us see.

Exercise 72 *Let $\mathcal{M}, \mathcal{N}$ be NFA's with n states each. Construct NFA's with no more than $2n$ states and which accept the languages*

$$L(\mathcal{M}) \cup L(\mathcal{N}), L(\mathcal{M}) \cap L(\mathcal{N}), L(\mathcal{M}) \circ L(\mathcal{N}) \text{ and } L(\mathcal{M})^{*}.$$

Exercise 73 *Let α be a regular expression. Show that it is possible to construct in linear time a NFA with $O(|\alpha|)$ states that accepts the language L_α.*

The above two exercises tells us that we can efficiently compile small NFA's from regular expressions. We cannot do the same with DFA's

Exercise 74 *Prove that the language L_k is denoted by a regular expression of length $O(k)$.Prove that L_k can be denoted with a regular expression of length*

$O(k)$.

We argued before that the complete elimination of nondeterminism is very much expensive than just exponential. If we insist in the complete elimination of nondeterminism we get an algorithm whose running time cannot be bounded by an elementary function. On the other hand, if we allow us to use NFA's when we compile regular expressions we can get small automata: we get NFA's of linear size or, by applying the power construction just at the very end of the compiling process, we get DFA's of exponential size. The former option seems to be very much better than the latter, and very much better than building DFA's of galactic size, as are the automata we end up building when we insist on eliminating nondeterminism from the very beginning. It remains to ask if we can run(simulate) NFA's efficiently. We discuss this issue, and some related topics, in the last lecture of this booklet.

References

[1] S. Kleene. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*, pages 3 - 42, Princeton University Press, Princenton NJ 1951.

[2] M. Rabin, D. Scott. Finite automata and their decision problems. IBM Journal of Research and Development 3 (2): 114–125, 1959.

8 Nondeterminism II: Does Nondeterminism Provide Computational Power?

Does nondeterminism provide computational power? If we think in DFA's the answer could be NOT as NFA's cannot recognize nonregular languages. If we think in regular expressions the answer might be YES as NFA's allow us to efficientlly process regular expressions, and it seems that we cannot do this processing without the help of NFA's. Thus, the match seems to be undecided.

If we focus on star-free languages the answer is NOT:

We can efficiently process star-free expressions with the help of deterministic two-way automata with nested pebbles. We cannot efficiently process star-free expressions using NFA's as those automata are ill suited to deal with complementation.

8.1 On the Compilation of Star-Free Expressions

Let us go back to star-free expressions. The set of star-free expressions is somewhat orthogonal to the set of regular expressions: we renounce to use Kleene star, but instead of this we use complementation. The set of star-free expressions determines an important proper subset of the regular languages: the star-free languages are the languages that can be defined by first order formulas and by linear temporal formulas.

Exercise 75 *Let $EVEN$ be the language $\left\{a^{2n} : n \geq 0\right\}$. Prove that $EVEN$ is a regular language, and prove that $EVEN$ cannot be defined by a star-free expression.*

Exercise 76 *Investigate about the theorem of McNaughton-Papert.*

We build a computing system around the language constituted by the star-free expressions. Let α be one SF-program (let α be a star-free expression). The first step in the compilation of α consists in computing the parse tree $(\mathcal{T}_\alpha, l_\alpha)$. The second step consists in computing a DFA from $(\mathcal{T}_\alpha, l_\alpha)$. We can use two-wayness in this second stage to deal with $\cup$, and control, to some extent, state explosion. We can also use one pebble to deal with the outermost concatenation occurring in α. However, if α contains more than one occurrence of $\circ$, we get forced to use the aforediscussed power construction. If we stick with DFA's we end the compiling process with automata of galactic size (actually we are uncapable of finishing with the compiling). Thus, we have to conclude that our simulator does not work with expressions of moderate size that have a moderate large number of occurrences of the symbol $\circ$. What can be done? We could try to use NFA's. We know that those automata are well suited to deal with joins and concatenations. However, we have to take into account that in this scenario we also have to deal with complementations.

8.1.1 Complementing NFA's

Let L be a regular language and let $\mathcal{M} = (\Sigma, Q, q_0, A, \delta)$ be a DFA that recognizes L. It is very easy to complement $\mathcal{M}$ and construct a DFA with $|Q|$ states that accepts *co-*L. It suffices if we set

$$\mathcal{M}^{co} = (\Sigma, Q, q_0, Q \backslash A, \delta).$$

It is a pity that this trick does not work for NFA's. Let $\mathcal{N} = (\Sigma, Q, q_0, A, \delta)$ be a NFA. Let w be an input of $\mathcal{N}$, and suppose that there exist an accepting computation as well as a rejecting computation of $\mathcal{N}$ on w. We have that $\mathcal{N}$ and $\mathcal{N}^{co}$ both accepts w.

How can we complement $\mathcal{N}$? We can use the power construction, construct $\mathcal{N}^{\varpi}$, and then complement $\mathcal{N}^{\varpi}$. This construction works, but it has an exponential cost: we get, after applying the construction, a NFA of exponential size. Can we do better? Unfortunately the answer is not (see [3]):

Theorem 77 *There exists a sequences of NFA's, say the sequence* $\{\mathcal{M}_i\}_{i \geq 1}$, *such that the size of* $\mathcal{M}_i$ *is* $O(i)$ *and such that the size of the minimal NFA recognizing co-*$L(\mathcal{M}_i)$ *is* $\Omega(2^i)$.

Let us discuss the proof of this theorem. Let $n \geq 1$. We use the symbol Γ_n to denote the set of all digraphs that are made of $2n$ vertices arranged into two columns (the left and the right columns) each of height n, and such that all the edges are directed from the left column to the right column.

Let $G, H \in \Gamma_n$, we catenate those two graphs by identifying the right column of G with the left column of H. We can use this associative operation to catenate finite many of those graphs: Γ_n is a finite alphabet, and the finite strings over this alphabet are words as well as digraphs.

Let B_n be the language

$$\left\{ \begin{array}{c} g_1 \cdots g_m \in \Gamma_n^* : m \geq 1 \;\&\; \text{the digraph } g_1 \cdot \cdots \cdot g_m \text{ contains a path} \\ \text{from the leftmost column to the rightmost column} \end{array} \right\}.$$

Theorem 78 *The language* B_n *can be recognized by a NFA with* $O(n)$ *states. On the other hand, we have that co-*B_n *requires* 2^n *states to be accepted by a NFA.*

Exercise 79 *It is an excellent exercise for the reader to prove that* B_n *can be recognized with* $O(n)$ *states.*

Let us discuss the proof of the opposite direction. We use the following lemma.

Lemma 80 *Let* L *be a regular language and let* X *be a set of indices. Suppose that there exist two sets of strings, say* $\{v_x : x \in X\}$ *and* $\{w_x : x \notin X\}$, *such that:*

1. *For all x the string $v_x \circ u_x$ belongs to L.*

2. *For all $x \neq y$ either $v_x \circ u_y \notin L$ or $v_y \circ u_x \notin L$.*

Let $\mathcal{M}$ be a NFA that recognizes L, automaton $\mathcal{M}$ cannot have less than $|X|$ states.

Proof. Suppose that L is accepted by a NFA $\mathcal{M}$. Given $x \in X$, we assign to x a state q_x such that $q_x \in \widehat{\delta}(q_0, v_x)$ and $F \cap \widehat{\delta}(q_x, u_x) \neq \emptyset$. We have that for all $x \neq y$ the condition $q_x \neq q_y$ holds. We get that $|Q| \geq |X|$. The lemma is proved ■

Let $G \in \Gamma_n$. We can assume that the nodes of G are called $1, ..., n, 1^*, ..., n^*$. Let $A_1, ..., A_{2^n}$ be an enumeration of the subsets of $\{1, ..., n\}$. Given $i, j \leq 2^n$ we can construct a string w_{ij} with the following property: the nodes of the rightmost column that can be reached from the leftmost column are the nodes in A_j, and the nodes of the leftmost column that reach some of the former nodes are the nodes in A_i.

Given $i \leq 2^n$ we set $u_i = w_{ii}$ and $v_i = w_{k(i)k(i)}$, where $k(i)$ is the index of *co*-A_i. We get that:

1. $u_i \circ v_i \in$ *co*-B_n

2. Given $i \neq j$ we have that either $u_i \circ v_j \in B_n$ or $u_j \circ v_i \in B_n$.

We get that a NFA recognizing *co*-B_n cannot have less than 2^n states, and we meet a surprising conclusion: deterministic automata are suited better than nondeterministic automata to deal with complementations. We also meet a worrying conclusion: we cannot put our programming language to work using nondeterminism in the naive way we are using it. What can be done? There exists a deterministic architecture that is well suited to deal with joins, concatenations and complementations (any deterministic architecture is well suited to deal with complementations)

8.1.2 Automata with Nested Pebbles

An automaton with nested pebbles is a 2DFA provided with an unbounded amount of labelled pebbles which have to be used in a LIFO manner, that is:

There is a bijective labeling of the pebbles that assigns to each pebble a positive integer. Moreover, the labels of the pebbles that are placed on the tape at a given instant of the computation have to constitute an initial segment of the natural numbers. The latter means that the first pebble to be placed on the tape has to be pebble 1. Then, we can place pebble 2 and then pebble 3. If we want to pick up pebble 1 we have to pick up first pebbles 3 and 2.

Exercise 81 *Recall the two-pebble automaton that recognizes the language Squares. Does this automaton use its two pebbles in a LIFO manner? Prove that it is not possible to recognize Squares using two pebbles in a LIFO manner.*

The reader should solve the above exercise without invoking the theorem below. The exercise is a warm up for this theorem (for a proof see [1]).

Theorem 82 *Automata with nested pebbles cannot recognize nonregular languages.*

We can recognize all the regular languages without using pebbles, or using at most one pebble. We have to ask: what do we gain with all those nested pebbles? Notice that we can use our provision of pebbles to deal with concatenations in a proper way.

Exercise 83 *Let α be a star-free expression of length n. Show that we can use the parse tree of α and the Bottom-Up protocol to construct an automaton with nested pebbles that recognizes the language L_α and which uses no more than n pebbles and no more than $3n$ states.*

Exercise 84 *How many states are required to recognize co-B_n using automata with nested pebbles.*

8.1.3 The Toy Model Version 2.0

We can process star-free expressions efficiently and without using a single bit of nondeterminism. Given α, and given the parse tree $(\mathcal{T}_\alpha, l_\alpha)$, we use the Bottom-Up protocol to compute a 2DFA with nested pebbles that recognizes the language L_α. We use the symbol $\mathcal{M}_\alpha$ to denote this automaton.

Observe that we cannot run $\mathcal{M}_\alpha$ on $\mathcal{U}_3$. This is so because $\mathcal{M}_\alpha$ might use very much more than three pebbles. Therefore, we have to use a simulator that is equipped with infinite many pebbles. We can construct a machine $\mathcal{U}_\infty$ analogous to $\mathcal{U}_3$ but which is equipped with the required pebbles. We can use $\mathcal{U}_\infty$ to simulate any 2DFA with nested pebbles. We can also use all those facts an ideas to construct a reloaded computing system that can efficiently compile and execute all the SF-programs. This new system is constituted by the triple (CYK-SF*,Comp-NP*, $\mathcal{U}_\infty$) .

1. CYK-SF* is a slightly modified parsing algorithm. CYK-SF* computes, on input α and as it is done by CYK-SF, the parse tree of α. The main difference between those two parsers is that CYK-SF* traverses the computed tree using depth first search in order to assign to the nodes with label $\circ$ a second label. This second label is a positive integer that determines the label of the pebble that will be assigned to the corresponding concatenation. We use the symbol $(\mathcal{T}_\alpha, l_\alpha)^*$ to denote the parse tree with integer labels that is computed by CYK-SF*.

2. Comp-NP* is a compiler that computes, from $(\mathcal{T}_\alpha, l_\alpha)^*$, a small 2DFA with nested pebbles that recognizes the language L_α.

Exercise 85 *Fill into the details: write down a detailed description of Comp-NP*.*

3. $\mathcal{U}_\infty$ is the aforementioned simulator.

Exercise 86 *Can you incorporate all those new facts into your final project?*

References

[1] N. Globermann, D. Harel. Complexity Results for Two-Way and Multi-Pebble Automata and their Logics. Theor. Comput. Sci. 169(2): 161-184, 1996.

[2] R. McNaughton, S. Papert. *Counter-Free Automata.* Cambridge University Press, Cambridge Mass., 1971.

[3] W. Sakoda, M. Sipser. Non-determinism and the size of two-way finite automata. Proceedings 10-th Symposium on the Theory of Computing: 275 - 286, 1978.

9 Nondeterminism III: Chomsky-Schutzenberger 1959

Pushdown automata (PDA's, for short) are nondeterministic one-way automata provided with an external-memory called a pushdown stack. Let $\mathcal{M}$ be a PDA, automaton $\mathcal{M}$ can write on its pushdown stack, but the access to this special tape is restricted by the following constraint: the scanning-writing head of the stack has to be placed, all the time, on the occupied cell that is most to the left.

Remark 87 *Notice that a PDA $\mathcal{M}$ can do no more than two things with its stack: $\mathcal{M}$ can either push a new symbol in its stack or pop the last symbol in it. Therefore the given name of pushdown automata.*

Definition 88 *A pushdown automaton is a tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, A, \delta)$ such that:*

1. *Q is a finite set of states.*
2. *$q_0 \in Q$ is the initial state.*
3. *Σ is a finite alphabet called the input alphabet.*
4. *Γ is a finite alphabet called the stack alphabet. We suppose that the special symbol $\square$ belongs to Γ.*
5. *The transition relation is a finite set of tuples. There are three types of tuples in δ :*
   - *Tuples of the form $(q, a, A, p, \_)$. This tuple indicates that if the automaton is in state q scanning on the tape the character a and scanning on the stack the character A, then the machine pops the last character in the stack.*
   - *Tuples of the form $(q, \_, A, p, BC)$. This tuples are called ε-transitions. A tuple like $(q, \_, A, p, BC)$ indicates that if the automaton is is scanning A on the stack, then the machine can nondeterministically decide to push the characters B and C in the top of the stack without consuming a character from the tape.*

Let $\mathcal{M}$ be a PDA, let w be an input of $\mathcal{M}$ and let c be a computation of $\mathcal{M}$ on input w. We say that c is accepting, if and only if, in the last configuration reached by c the stack is empty (we say that $\mathcal{M}$ accepts w by *empty stack*). We set

$$L(\mathcal{M}) = \{w : \text{there exists a computation of } \mathcal{M} \text{ that accepts } w \text{ by empty stack}\}.$$

We say that $L(\mathcal{M})$ is the language accepted by $\mathcal{M}$.

9.1 Context-free grammars and Pushdown Automata

N. Chomsky introduced the model of nondeterministic pushdown automata (PDA's, for short) three years before the publication of Rabin-Scott paper [1]. PDA's were introduced by Chomsky to provide a machine characterization of context-free grammars.

Definition 89 *Let $G = (V, T, S, R)$ be a context-free grammar. We say that G is a Chomsky grammar, if and only if, any production rule in G has either the form $A \to \alpha$ or the form $A \to BC$, where $\alpha \in T$ and $B, C \in V$.*

Let G, H be two grammars. We say that those grammars are equivalent if and only if the equality $L(G) = L(H)$ holds. It is not hard to prove that any context-free grammar G that does not produce the empty string is equivalent to a Chomsky grammar. Moreover, given G as above, one can compute a grammar G^* such that G^* is in Chomsky normal form and G is equivalent to G^*. From now on we suppose that all our grammars do not produce the empty string.

Theorem 90 ***Chomsky's Theorem***
The language L is a context-free language, if and only if, there exists a PDA $\mathcal{M}_G$ such that $L(\mathcal{M}_G) = L$.

Proof. Let $G = (V, T, S, R)$ be a context-free grammar. We suppose that G is a Chomsky grammar. We construct a PDA $\mathcal{M}_G$ that accepts $L(G)$. The construction of $\mathcal{M}_G$ presupposes that we will encode the computations that can performed using G as computations of a pushdown automaton.

Let $w \in T^*$. Whant are the computations of G on this string? The computations performed using G, and related to w, are the G-derivations that produce w. We can think of those derivations as finite sequences of configurations. One of those derivations is a sequence

$$\underline{S}, W_1, W_2, ..., w$$

where each element of the sequence is a string $W \in (V \cup T)^*$ with one marked position. The marked position is always occupied by the variable that will be transformed in the next step. We can try to see those sequences as computations of a PDA. It is not easy to figure out such a PDA given that the above sequence can evolve in an unordered way. Notice that we can choose any variable occurring in W_i to work on it and produce W_{i+1}. The latter suggests that our PDA has to go from left to right and from right to left visiting different regions of the input tape in an unordered way. PDA's do not work this way, PDA's work from left to right. Therefore, we have to impose some order on the computations that we perform with G. We say that a derivation is a left-derivation, if and only if, the transition from W_i to W_{i+1} proceeds by applying a production rule to the leftmost variable of W_i. Let

$$\underline{S}, W_1, W_2, ...W_k, w$$

be a left-derivation and let $i \leq k$. Notice that W_i is equal to $u_i X U_i$, where $u_i \in T^*$, X is the leftmost variable of W_i and $U_i \in V^*$. Those configurations that are made of two discernible strings resembles the configurations of PDA's. Let $\mathcal{M}$ be a PDA and let v be an input of $\mathcal{M}$. suppose we run $\mathcal{M}$ on input u. After i time units we reach a configuration that is perfectly described by two strings: the prefix $u[1, ..., j]$ that has been scanned so far and the string $X_i \in \Gamma^*$ that is written on the pushdown stack. It is fairly easy to figure out a PDA that accepts $L(G)$. Let $\mathcal{M}_G = (T, \Gamma, Q, q_0, \delta)$, where:

- $\Gamma = V$.
- $Q = \{q_0\}$.
- The tuple $(q_0, a, X, q_0, \varepsilon)$ belongs to δ, if and only if, the rule $X \rightarrow a$ belongs to R.
- The ε-transition $(q_0, , X, q_0, YZ)$ belongs to δ, if and only if, the rule $X \rightarrow YZ$ belongs to R.

It is not hard to check that $\mathcal{M}_G$ accepts $L(G)$. It is also easy to follow this same path, but in the opposite direction, and build a Chomsky grammar from a PDA. We leave the remaining details to the reader and we claim that the theorem is proved. ■

PDA's, with all their nondeterministic transitions including the misterious ε-transitions, are a little bit bizarre. We have to take into account that nondeterminism naturally occur in context-free grammars, where several production rules can be applied to the same nonterminal symbol. It seems that the nondeterministic character of PDA's naturally arise from their connection with the mathematical modeling of natural languages. However, if we think of those automata as computing devices its nondeterministic nature becomes problematic. Recall that NFAs solve the same problems that are solved by DFAs. It is natural to ask: is nondeterminism essential for capturing context-free grammars and context-free languages using the architecture of pushdown automata?

Exercise 91 *Let $L \subset \{1\}^*$, suppose that L is context-free, prove that L is regular (Hint: investigate about Parikh's Theorem).*

9.2 Deterministic Pushdown Automata

M. Schutzenberger studied the set of languages that are accepted by deterministic PDAs, the so called deterministic context-free languages [2]. We use the symbol DCFL to denote this set of languages. Schutzenberger proved that DCFL is different to CFL.

Exercise 92 *Let L be a language, and let*

$$\begin{aligned} \min(L) &= \{w \in L : \text{no proper prefix of } w \text{ belongs to } L\}, \\ \max(L) &= \{w \in L : \text{no proper suffix of } w \text{ belongs to } L\}. \end{aligned}$$

If L is a deterministic context-free language then $\min(L)$ *and* $\max(L)$ *are deterministic context-free languages. Use this to prove that the language of binary palindrome cannot be recognized by a deterministic pushdown automata (Hint. you will have to prove that either* $\min(pal)$ *is not context-free or* $\max(pal)$ *is not context-free, to prove the latter you can use the pumping lemma for context-free languages). Conclude that nondeterminism provides computational power to pushdown automata.*

Let L be a nondeterministic context-free language generated by a grammar G. Suppose we construct a PDA $\mathcal{M}$ that accepts L. Automaton $\mathcal{M}$ is nondeterministic and it implies that we cannot use it to recognize the language L : recall that $\mathcal{M}$ does not make errors, but we make errors when we use nondeterministic machines. Let w be an input of $\mathcal{M}$. We could try a deterministic simulation of all the computations of $\mathcal{M}$ on input w. However, this simulation takes exponential time. Does this means that the unique *feasible* solution for L is the nondeterministic automaton $\mathcal{M}$? If this were the case we would have to accept that nondeterminism provides computational power. We have to notice that $\mathcal{M}$ provides us with a real-time solution for the problem of recognizing L. We would conform ourselves with an algorithmic solution that runs in deterministic polynomial time. Is this possible?

9.3 Parsing Context-Free Grammars in Deterministic Polynomial Time

Let G be a context-free grammar. Consider the problem:

Problem 93 ***Computing Derivations for*** $L(G)$.

- *Input:* $w \in T^*$.
- *Problem: decide if* $w \in L(G)$, *in that case compute a derivation tree for w with respect to the grammar G.*

The Cocke–Younger–Kasami Algorithm (CYK algorithm, for short) is a parsing algorithm for context-free grammars that employs bottom-up parsing and dynamic programming [3]. This algorithm receives as input a Chomsky grammar G and a string w, and it outputs either a parse tree for w or a message indicating that $w \notin L(G)$. CYK is a dynamic programming algorithm. This means that CYK computes an array, of which some entries can be easily computed at the beginning of the computation, while the remaining entries are computed in a dynamic way propagating the data that are encoded into the previously computed entries. The computed array is equal to $[P_{l,s,v}]_{l \leq s \leq |w|, v \in V}$, where

$P_{l,s,v} = 1,$ if and only if, the string $w[l,s]$ can be derived from the variable v.

Note that it is easy to compute all the entries of the form $P_{l,l,v}$. To do this, one simply has to check if G contains the production rule $v \rightarrow w[l]$. To propagate the previously computed values one uses the following rule

$$P_{l,s,v} = 1 \text{ and } P_{s+1,r,w} = 1 \text{ and } (X \rightarrow vw) \in R \text{ imply that } P_{l,r,X} = 1.$$

It is easy to decide, after computing the whole array, wether $w \in L(G)$: we only have to check whether the equality $P_{1,|w|,S} = 1$ holds.

Exercise 94 *Analyze the running time of the CYK algorithm. Prove that the CYK algorithm runs in time* $O\left(|w|^3\right)$.

Generating a parse tree The above algorithm can be used to decide whether the string w belongs to $L(G)$. It is fairly easy to transform this algorithm into a parser that constructs a parse tree for w provided that $w \in L(G)$. This parsing algorithm stores parse trees into the entries of the array instead of Boolean values.

Exercise 95 *Write down the a description of this latter algorithm.*

We can use deterministic algorithms to solve problems that are harder and more general than the problems solved by PDA's. However, we have to observe that PDA's are more efficient than those deterministic algorithms that run in cubic time. Can we recognize all the context-free languages in deterministic real-time? It seems that the answer to the latter question is negative. This would represent more evidence in favor of the power of nondeterminism. We have already found some evidences in favour of nondeterminism. We have also found evidences against nondeterminism. It could be said that the question about the power of nondeterminism remains undecided. It can also be said that it would remain undecided if we stick with restricted models of computation like finite and pushdown automata. Perhaps it is time to approach the question from the point of view of Turing machines.

References

[1] N. Chomsky. Three models for the description of language. IEEE Transactions on Information Theory, 2(3): 113–124, 1956.

[2] M. Schutzenberger. On context-free languages and pushdown automata. Inf. and Control, 6(3): 246 - 264, 1963.

[3] T. Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages- Technical report AFCRL. 65-758, 1965.

10 Can It Be Done With Pebbles Alone?

The main question is: does nondeterminism provide computational power?

If we consider unrestricted Turing machines we get a negative answer: the problems that can be solved by deterministic machines are the same as the problems that can solved by nondeterministic machines.

Nondeterminism does not allow us to solve new problems, but nondeterminism might allow us to solve some problems in a more efficient way. We were able to see glimpses of this possibility in previous lectures. In those lectures we were considering the same question bu from the point of view of restricted architectures like finite one-way automata and pushdown automata. We were told about the existence of many problems that can be solved by nondeterministic automata and which cannot be solved by their deterministic counterparts. It could be argued that the existence of those problems fully certifies the supremacy of nondeterminism. We claim that the aforementioned facts are no more than just glimpses of this possible supremacy. We claim this because of the following fact: all the aforementioned advantages (supposedly) provided by nondeterminism can also be provided by suitable deterministic architectures. Think for a moment in PDA's. The context-free languages that cannot be recognized by DPDA's can be recognized by polynomial time deterministic Turing machines. We would accept the claimed supremacy of nondeterminism if someone can exhibit a proper separation between complexity classes defined by Turing machines.

10.1 Logarithmic Space

We begin in this lecture the study of complexity classes of Turing machines (of complexity theory). We begin with the study of the class L.

Definition 96 *Let $\mathcal{M}$ be a nondeterministic Turing machine which is provided with a read-only input tape and a finite number of work tapes, one head per tape. We say that $\mathcal{M}$ is a multitape Turing machine. Let w be an input of $\mathcal{M}$, we define $s_{\mathcal{M}}(w)$ as*

$$\max\{\#cells(c) : c \text{ is a computation of } \mathcal{M} \text{ on } w\},$$

where $\#cells(c)$ is equal to the maximum number of work-cells that are simultaneously occupied along the execution of c.

We define $S_{\mathcal{M}} : \mathbb{N} \to \mathbb{N}$ by the equation

$$S_{\mathcal{M}}(n) = \max\{s_{\mathcal{M}}(w) : w \in \Sigma^n\}.$$

Function $S_{\mathcal{M}}$ measures the memory requirements of the algorithm embodied by $\mathcal{M}$.

Exercise 97 *Let $C > 0$, and let $\mathcal{M}$ be a multitapeTuring machine that satisfies the inequality $S_{\mathcal{M}}(n) \leq Cn$. Prove that there exists a machine $\mathcal{M}^*$ that recognizes the language $L(\mathcal{M})$ and which satisfies the inequality $S_{\mathcal{M}}(n) \leq n$.*

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a non-decreasing function. We define two complexity classes:

$$\begin{aligned} NSPACE(f) &= \left\{ \begin{array}{c} L : L \text{ can be recognized by a nondeterministic multitape} \\ \text{Turing machine } \mathcal{M} \text{ such that } S_{\mathcal{M}}(n) \in O(f) \end{array} \right\}, \\ DSPACE(f) &= \left\{ \begin{array}{c} L : L \text{ can be recognized by a multitape deterministic} \\ \text{Turing machine } \mathcal{M} \text{ such that } S_{\mathcal{M}}(n) \in O(f) \end{array} \right\} \end{aligned}$$

The class L (logspace, DSPACE($\log(n)$)) is the class of problems that can be solved using a logarithmic amount of workspace. Observe that we are talking about machines that use a sublinear amount of workspace. This forced us to separate the input tape, which always has linear size, and the tapes where the computing work is done.

Definition 98 *A logspace Turing machine is a machine with $k+1$ tapes, one head per tape. We can suppose that those tapes are numbered, from 1 to $k+1$, and we suppose that tape 1 is a read only input tape. We suppose that the remaining k tapes are work tapes that are empty at the beginning of the computation. Moreover, there exists a constant c such that the machine uses at most $c\log(n)$ cells of each work tape along its computations on inputs of size n.*

There are nontrivial problems that can be solved with logspace. Let α be a first order sentence over the vocabulary τ. Consider the problem:

Problem 99 ***MC(α): Model Checking of*** α.

- *Input: $\mathcal{U}$, where $\mathcal{U}$ is a finite τ-structure.*
- *Problem: decide if $\mathcal{U}$ satisfies α.*

Exercise 100 *Prove that MC(α) can be solved using a logspace Turing machine.*

Remark 101 *One of the most important results of the last two decades is the result of Omer Reingold asserting that the graph accessibility problem for undirected graphs is in L [1].*

Exercise 102 ***The class PTIME.*** *The class PTIME (P, for short) is constituted by all the problems that can be solved in deterministic polynomial time. Prove that L is included in P.*

10.2 Pebble Automata and the Class L

Let us bring back our old good friends the pebbles.

Definition 103 *We use the symbol $\mathcal{REG}_k$ to denote the set of languages that can be recognized by k-pebble automata.*

We have:

Theorem 104 *The class L is equal to* $\bigcup_{k\geq 0} \mathcal{REG}_k$.

Proof. It is easy to simulate pebble automata using logspace machines. This is so because it is very easy to track a pebble using a tape of logarithmic size that is used to save the location of the pebble. We get that $\bigcup_{k\geq 0} \mathcal{REG}_k$ is contained in L.

It is not hard to simulate a logspace machine using pebble automata. This is so because it is not hard to simulate a tape of logarithmic size using some few pebbles:

Let $\mathcal{M}$ be a logspace machine. We can suppose that the work alphabet of $\mathcal{M}$ is equal to $\{0,1\}$. Let $w_1 \cdots w_l$ be the content of one of the worktapes of $\mathcal{M}$. This configuration is well represented by the natural numbers a and b whose binary codes are equal to $1w_1 \cdots w_{i-1}, 1w_i \cdots w_l$. Those two numbers can be represented by the positions on the input tape of two specially assigned pebbles. It is not hard to use those two pebbles, and some few additional pebbles, to track the evolution of the corresponding tape. ■

10.3 The class NL: Nondeterministic Pebbles

The class NL is equal to NSPACE($\log(n)$). We have that L $\subseteq$ NL. It is not known whether this containment is strict. The question about the relation between the classes L and NL is one of the most important questions of complexity theory.

Definition 105 *Let* $\mathcal{NREG}_k$ *be the set the languages that can be recognized by* k*-pebble nondeterministic automata.*

It is easy to prove that NL is equal to $\bigcup_{k\geq 0} \mathcal{NREG}_k$. We claim that this representation of NL can help us to understand this complexity class. Let us ask: what are the problems that can be solved using nondeterministic logarithmic space? Consider the following problem

Problem 106 ***GA: Graph Accessibility***

- *Input:* (G,s,t), *where* G *is a digraph and* $s,t \in V(G)$.
- *Problem: decide if there exists a path in* G *that goes from* s *to* t.

The above problem is an ubiquitous problem that occurs in all the areas of combinatorial optimization. This problem can be easily solved using nondeterministic logarithmic space. Let us see this using pebbles.

Let G be a digraph with nodes in the set $\{1, ..., n\}$. We encode G by means of the string

$$w_G = a0aa0 \cdots 0a^n00\varepsilon_{11} \cdots \varepsilon_{1n}00 \cdots 00\varepsilon_{n1} \cdots \varepsilon_{nn},$$

where

$$\varepsilon_{ij} = \begin{cases} 1, \text{ if there exists an edge from } i \text{ to } j \\ 0, \text{ otherwise} \end{cases}$$

Given a triple (G, s, t), we encode this triple by means of the string

$$w_{Gst} = 1^s01^t0w_G.$$

The problem GA is exactly the same as the recognition problem for the language

$$L_{GA} = \{w_{Gst} : (G, s, t) \text{ belongs to GA}\}.$$

Theorem 107 *Language L_{GA} can be recognized with $2 + O(1)$ pebbles.*

Proof. We use pebble 1 and 2 to walk over the graph. We use the remaining pebbles to check whether the steps that we try can be done.

We begin searching the substring a^s. After locating this substring we place pebble 1 on it. Then, we guess a node v_1 and we place pebble 2 on the substring a^{v_1}. We use the remaining pebbles to search for the letter ε_{sv_1}, after locating this letter we check whether it is equal to 1. If this is the case we check whether v_1 is equal to t, and if v_1 is in effect equal to t we accept w. If v_1 is different of t we move pebble 1 to a^{v_1} and we guess a node v_2.

If ε_{sv_1} is 0 we keep pebble 1 on a^s and we guess once again the node v_1. We continue in the same way guessing and checking. We halt and accept if we reach node t. ■

References

[1] O. Reingold. Undirected ST-connectivity in log-space. Proceedings of the 37th Annual ACM Symposium on Theory of Computing STOC, pp 376–385, 2005.

11 NL-Completeness

The question L vs NL is, as we said before, one of the most important questions of complexity theory. We study this question in the next few lectures.

11.1 Logspace Reductions

Let $\mathcal{C} \subseteq \mathcal{D}$ be two complexity classes. The question about the equality of those two classes can be attacked using a suitable notion of algorithmic reducibility.

Definition 108 *A k-pebble (logspace) transducer is a k-pebble automaton provided with an additional write-once output tape.*

Definition 109 *Let $L \subset \Sigma^*, T \subset \Gamma^*$ be two decision problems, a logspace reduction from L into T is a function $f : \Sigma^* \rightarrow \Gamma^*$ such that:*

1. *Given $w \in \Sigma^*$, we have that $x \in L$, if and only if, $f(x) \in T$.*
2. *Function f can be computed by a pebble transducer.*

We use the symbol $L \leqslant_{\log space} T$ to indicate that L is logspace reducible to T.

Exercise 110 *Prove that:*

1. *The class L is closed under logspace reductions, that is: if $L \leqslant_{\log space} T$ and T belongs to L, then the language L also belongs to L.*
2. *The class NL is closed under logspace reductions.*

Definition 111 *Let L be a language. We say that L is NL-hard, if and only if, any problem in NL is logspace reducible to L. We say that L is NL-complete, if and only if, L is NL-hard and L belongs to NL.*

Exercise 112 *Suppose that there exist problems that are NL-complete. Prove that L is equal to NL, if and only if, there exists a NL-complete problem that belongs to L.*

We prove in next section that there exist problems that are NL-complete. This fact, and the above exercise, indicate that the notion of NL-completeness is appropriate to deal with the L vs NL question.

11.2 Graph Accessibility

Do there exist problems that are complete for NL?

Theorem 113 *Problem GA is NL-complete.*

Proof. Let L be a language in NL. We know that there exists a nondeterministic k-pebble automaton $\mathcal{M}$ that recognizes L. Let w be an instance of L and let s_w be the initial configuration of $\mathcal{M}$ that is determined by this input. The set of configurations of $\mathcal{M}$ that can be accessed from w has polynomial size with respect to $|w|$, and this implies that all its elements can be described using a logarithmic number of bits. Moreover, the whole graph can be constructed using logspace. Let $G_{\mathcal{M},w}$ be this graph. Let T_w be the set of accepting configurations that are contained in the set $V(G_{\mathcal{M},w})$. We have that $w \in L$, if and only if, $(G_{\mathcal{M},w}, s_w, T_w)$ is a YES-instance of GA. The function that sends w in the triple $(G_{\mathcal{M},w}, s_w, T_w)$ can be computed using a pebble transducer. The theorem is proved. ■

Exercise 114 *Prove that NL is contained in P.*

11.3 Monien's Language

Let G be a digraph with nodes $1, ..., n$. We say that G is a *well sorted digraph,* if and only if, for all $(i, j) \in E(G)$ the condition $i < j$ holds.

Exercise 115 *It is easy to see that graph accessibility remains complete for NL when restricted to well sorted digraphs. Prove this assertion.*

Let G be a well sorted digraph. We can describe digraph G using many different strings. We choose to use a string $w_G \in \{1, \#\}^*$ that we construct as follows:

We concatenate descriptions of the different pairs in $E(G)$. We begin listing in ascending order the edges that are incident with 1, we continue with the edges that are incident with $2, 3$ and so on. A pair (i, j) is simply encoded by the string $1^i\#1^j$. Consecutive pairs are glued together using the following rule:

- Let $W1^i\#1^j$ be the constructed prefix, and let $1^i\#1^k$ be the next pair $(j < k)$. Then, we append to $W1^i\#1^j$ the string $\#\#1^i\#1^k$.
- Let $W1^i\#1^j$ be the constructed prefix, and let $1^k\#1^l$ be the next pair $(i < k)$. Then, we append to $W1^i\#1^j$ the string $\#\#\#1^k\#1^l$.

Let AG (*Adjacency of Graphs*) be the language constituted by all the strings

$$1^i\#\#\#w_G\#\#\#1^{i+1}\#\#\#w_G\#\#\# \cdots \#\#\#1^{j-1}\#\#\#w_G\#\#\#$$

such that G is well sorted and $(i, j) \in E(G)$. Language AG captures the problem of recognizing the adjacencies of well sorted digraphs, while the language AG* captures the accessibility relations of those digraphs.

Exercise 116 *It is easy to check that AG can be recognized with two pebbles. It is also easy to see that AG^* is complete for NL [2]. Prove those two facts.*

Exercise 117 *Prove that NL is closed under the Kleene star, if and only if, L is equal to NL.*

We get that the language p-AG, defined below, is complete for NL.

Definition 118 *Let p-AG be the language*

$$\left\{c^k * 1^i\#\#\#w_G\#\#\#1^{i+1}\#\#\#w_G\#\#\#\cdots\#\#\#1^{j-1}\#\#\#w_G\#\#\# : \Psi\right\},$$

where Ψ is the condition asserting that the string

$$1^i\#\#\#w_G\#\#\#1^{i+1}\#\#\#w_G\#\#\#\cdots\#\#\#1^{j-1}\#\#\#w_G\#\#\#$$

belongs to the k-th power of the language AG (Ψ asserts that there exists a path of length k connecting the node i and the node j in the well sorted digraph G).

Remark 119 *The language p-AG is a concrete language representation of graph accessibility. We can use, and we will use, this concrete representation of GA to visualize some different constructions and some different algorithms that are related to GA.*

11.4 It Cannot Be done With Pebbles Alone: The Regex Membership Problem

In this section we prove that the membership problem for star-free expressions is NL-hard. This means that, unless L equal NL, this problem cannot be solved using pebble automata provided with a bounded number of pebbles. This implies that our toy model cannot be implemented using pebbles alone: we need the parser and the compiler, and those algorithms cannot be implemented with bounded pebbles.

Theorem 120 *The Star-Free Membership Problem is NL-hard.*

Proof. Proving that a certain problem L is NL-hard reduces to prove that GA is logspace reducible to L.

Let (G, s, t) be an instance of GA and suppose that $V(G) = \{1, ..., n\}$. We can add a loop to each node of G and get a triple (G^*, s, t) that holds the following property:

There exists a path in G from s to t, if and only if, there exists a path in G^* of length $n-1$ and which goes from s to t.

Let us suppose that we get an instance like (G^*, s, t), and that we are asked to decide whether there exists a path of length $n-1$ that goes from s to t. We suppose that $V(G)$ is equal to $\{1, ..., n\}$. The code of a node i is the string 0^i

that we denote with the symbol $c(i)$. The symbol $\overline{c(i)}$ denotes the string 0^{n-i}. Set

$$\begin{aligned} R_0 &= \left\{c(s)\,1\overline{c(v)} : (s,v) \in E(G)\right\} \\ R_{n-1} &= \left\{c(v)\,1\overline{c(t)} : (v,t) \in E(G)\right\} \\ R_i &= \left\{c(v)\,1\overline{c(w)} : (v,w) \in E(G)\right\}, \text{ where } i = 1, ..., n-2. \end{aligned}$$

Set

$$R = R_0 \cdot R_1 \cdot \cdots \cdot R_{n-2} \cdot R_{n-1}$$

and set $w = c(s) \cdot (10^n)^{n-2} \cdot 1 \cdot \overline{c(t)}$. Notice that R is star-free. Moreover, we have that $w \in L(R)$, if and only if, (G^*, s, t) is a positive instance of GAP. This constitutes a logspace reduction of GAP into The Star-Free Membership Problem. The theorem is proved. ■

References

[1] R. Floyd. Nondeterministic Algorithms. Journal of the ACM. 14 (4): 636 - 644, 1967.

[2] B. Monien. About the deterministic simulation of nondeterministic (log n)-tape bounded Turing machines. Automata Theory and Formal Languages 1975: 118-126.

[3] H. Petersen. The Equivalence of Pebbles and Sensing Heads for Finite Automata. FCT 1997: 400-410.

12 L, NL and co-NL

We continues with the study of the L vs NL question. This time we invite to the stage the class co-NL.

12.1 co-NL and The Immerman-Szelepcsényi Theorem

The Immerman–Szelepcsényi theorem states that nondeterministic space complexity classes are closed under complementation. It was proven independently by Neil Immerman and Róbert Szelepcsényi in 1987, for which they shared the 1995 Gödel Prize (see [2] and [3]). In its general form the theorem states that NSPACE($s(n)$) = co-NSPACE($s(n)$) for any function $s(n) \geq \log(n)$. We are interested in the special case $s(n) = \log(n)$. We prove that *co*-(p-AG) (the complement of the language p-AG) can be recognized by a nondeterministic automaton provided with a bounded number of pebbles. This suffices for our purposes since *co*-(p-AG) is co-NL complete (see below).

12.1.1 The Class co-NL.

We know that nondeterministic machines are ill suited to deal with complementations. This can cause nondeterministic classes to be not closed under this operation. On the other hand, we know that deterministic classes are closed under complementations. This provides us with a possible strategy to prove separations between deterministic and nondeterministic complexity classes. Recall that we proved that CFL is different to DCFL by proving that CFL is not closed under complementation.

Exercise 121 *Suppose that NL is not closed under complementation. Conclude that L is different of NL.*

Definition 122 *The class co-NL is the class*

$$\{\textit{co-L} : \textit{L belongs to NL}\}$$

Exercise 123 *Prove that co-NL is closed under logspace reductions.*

Exercise 124 *Prove that co-(p-AG) is co-NL complete.*

Exercise 125 *Prove that L is equal to co-NL, if and only if, co-(p-AG) belongs to NL.*

12.1.2 The Theorem

What does it means that *co*-(p-AG) belongs to NL? This means that we can use a bounded number of pebbles to certify that there does not exist a path of length k from s to t whenever the input string

$$1^k\$1^sW1^{t-1}\#\#\#w_G\#\#\#$$

belongs to *co*-(p-AG) .

Lemma 126 *There exists a* 4*-pebble automaton* $\mathcal{M}$ *which can certify, on an input*

$$1^k\$1^sW1^{t-1}\#\#\#w_G\#\#\#$$

that belongs to p-AG, the existence of a path of length k from s to t.

Proof. The automaton uses two pebbles to guess a path from s to t. Suppose $\mathcal{M}$ has guessed a path of length i that ends at the node l. This means that the automaton has its first pebble placed on the l-th block of

$$1^k\$1^sW1^{t-1}\#\#\#w_G\#\#\#.$$

Then, the automaton guesses the next node, say the node $v > l$. It means that $\mathcal{M}$ places its second pebble on the v-th block. Automaton had a third pebble that is placed on the i-th cell of the block 1^k. After moving the second pebble, the automaton moves this counter pebble one cell to the right. It remains to be checked that v is a heir of l. To this end, the automaton uses the first two pebbles and a fourth pebble that searches the substring w_G included in the l-th block.

Suppose that $\mathcal{M}$ reaches a configuration such that the first pebble is placed on the r-th block, the second pebble is placed on the t-th block and the third pebble is placed on the k-th cell. If after reaching this configuration, automaton $\mathcal{M}$ certifies that t is a son of r, then it has certified that there exists a path of length k from s to t. ■

Let us suppose that $\mathcal{M}$ is asked to certify that

$$1^k\$1^sW1^{t-1}\#\#\#w_G\#\#\#$$

belongs to *co*-(p-AG) . Let $\#_{s,k}$ be the number of nodes that can be reached from s in k steps, and let us suppose that a fifth pebble was magically placed on the $\#_{s,k}$-th cell.

Lemma 127 *Automaton* $\mathcal{M}$ *can, given the current hypothesis, use its four pebbles and two additional pebbles to certify that a string*

$$1^k\$1^sW1^{t-1}\#\#\#w_G\#\#\#$$

belongs to co-(p-AG) .

Proof. Let us use the symbol $\mathcal{M}^*$ to denote the modified automaton that uses six pebbles, and the magical pebble, to certify that a given string

$$1^k\$1^sW1^{t-1}\#\#\#w_G\#\#\#$$

belongs to *co*-(p-AG) . We proceed to describe the program of $\mathcal{M}^*$.

$\mathcal{M}^*$ uses the first four pebbles to certify, given $r > s$, that there exists a path of length k from s to r. Automaton $\mathcal{M}^*$ uses the fifth pebble to search the set $\{s+1, ..., n\} \setminus \{t\}$, and it uses the sixth pebble to count the number of nodes in this latter set that got certified by $\mathcal{M}^*$. Thus, when the fifth pebble reaches the n-th block, and after the node n is analyzed, the position of the sixth pebble becomes equal to number of nodes that were certified, along the process, as being reachable from s in k steps. If this pebble is located at position $\#_{s,k}$ (at the same position that the magical pebble), then $\mathcal{M}^*$ has certified that s is none of the $\#_{s,k}$ nodes that can be reached from s in k steps. This means that $\mathcal{M}^*$ has provided a convincing proof (its computation) that t cannot be reached from s in k steps. The lemma is proved. ■

It remains to be proved that we do not need the aid of magic to locate the $\#_{s,k}$-th cell, and that this cell can be located with a bounded number of pebbles.

Lemma 128 *Automaton $\mathcal{M}^*$ can use its six pebbles and five additional pebbles to locate the cell $\#_{s,k}$.*

Proof. Let us use the symbol $\mathcal{M}^{\#}$ to denote the modified automaton that, using no more than 11 pebbles, computes, on input

$$1^k\$1^sW1^{t-1}\#\#\#w_G\#\#\#,$$

the cell $\#_{s,k}$.

We use a method of inductive counting that was envisaged by Szelepcsényi. We compute the sequence $\#_{s,1}, ..., \#_{s,k}$. We begin with $\#_{s,1}$ and we save this count using one out of the 11 available pebbles. Suppose that we have already computed $\#_{s,i}$. This means that we have pebble 1 placed on the $\#_{s,i}$-th cell, and pebble 2 placed on the i-th cell. We proceed to compute $\#_{s,i+1}$. First, we move pebble 2 to the $i+1$-th cell indicating that this is the length of the paths that have to be counted. The key point is that a node r can be reached from s in $i+1$ steps, if and only if, node r can be reached in one step from one of the $\#_{s,i+1}$ nodes that can be reached from s in i steps.

Suppose pebble 3 is located on the r-th block. This means that $\mathcal{M}^{\#}$ is checking if r can be reached from s in $i+1$ steps. Notice that we have three pebbles in use:

1. Pebble 1. The position of this pebble, say position $i+1$, indicates the length of the paths that have to be counted.

2. Pebble 2. The position of pebble 2, which was inductively (not magically) computed, indicates the exact number of nodes that can be reached from s in i steps.

3. Pebble 3. The position of this pebble on the r-th block indicates what is the node that is being analyzed.

Automaton $\mathcal{M}^{\#}$ can use six out of the remaining 11 pebbles, and the (magical) second pebble, to certify the nodes in $\{s+1,...,r-1\}$ that can be reached from s in i steps. When one of those nodes gets certified, the automaton uses pebble number 10 to check whether this node is an ancestor of r. If the checking is positive node r gets certified as being reachable from s in $i+1$ steps. Then, the automaton moves the pebble number 11 one position to the right. $\mathcal{M}^{\#}$ uses pebble number 11 to count the number of nodes that can be reached from s in $i+1$ steps. If node r gets certified, and pebble number 11 is moved to the right, then pebble number 3 is placed on the $r+1$-th block. On the other hand, if $\mathcal{M}^{\#}$ cannot certify the existence of an ancestor of r that can be reached from s in i steps, then the position of pebble number 11 remains unchanged, and pebble number 3 is moved to the $r+1$-th block.

Suppose that pebble number 1 reaches the k-th cell. Suppose also that after a certain number of steps pebble number 3 reaches the n-th block, and suppose that the analysis of node n (wrt length k) is successful, then the final position reached by pebble number 11 is equal to $\#_{s,k}$. The lemma is proved. ■

We get as an easy corollary of the above lemmata that *co*-(p-AG) can be recognized by a nondeterministic 11-pebble automaton.

We also get the following corollary

Theorem 129 *The following assertions hold true.*

1. *NL is equal to co-NL.*
2. *Given $s(n) \in \Omega(\log(n))$, the class NSPACE($s(n)$) is equal to the class co-NSPACE($s(n)$).*

12.2 Savitch´s Theorem

In this section we prove that the nondeterministic class NL is included in $DSPACE\left(\log^2(n)\right)$. The proof is based on two key facts:

1. Let L be a problem that is complete for NL under logspace reductions. It suffices if we prove that L belongs to $DSPACE\left(\log^2(n)\right)$.
2. The language p-AG is complete for NL.

Lemma 130 *Suppose that language L can be recognized by a pebble automaton provided with $f(n)$ pebbles. We have that L belongs to $DSPACE(f(n)\log(n))$.*

Proof. We only have to recall that each pebble can be tracked using a tape of logarithmic size. ■

We can use the above lemma, and the versatility of pebble automata, to prove that certain languages belong to $DSPACE\left(\log^2(n)\right)$. Let us illustrate this fact using the language *p-pal* that is defined below.

Definition 131 *The language p-pal is the language*

$$\left\{c^k\#w : k \geq 1 \ \& \ w \in \{0,1\}^* \ \& \ w \textit{ is the concatenation of } k \textit{ palindrome}\right\}.$$

We have:

Proposition 132 *The language p-pal belongs to* $O\left(\log^2(n)\right)$.

Proof. We prove that *p-pal* can be recognized with $O(\log(n))$ pebbles.

Let P_k be equal to the set

$$\left\{c^k\#w : w \in \{0,1\}^*\right\} \cap p\text{-}pal$$

and let $p(k)$ be the number of pebbles that are required to recognize the strings in P_k. Note that it suffices if we prove that for all $k \geq 1$ the inequality

$$p(k) \leq \left(p\left(\left\lceil\frac{k}{2}\right\rceil\right) + 1\right)$$

holds.

It is easy to recognize the strings in P_k using $k-1$ pebbles. However, we can use a divide and conquer strategy to recognize P_k employing very much less pebbles, no more than a logarithmic number of them. Let us use this strategy to prove the above inequality. Thus, suppose we have to check wether $c^k\#w$ belongs to *p-pal*. We show that we can do this using no more than $\left(p\left(\left\lceil\frac{k}{2}\right\rceil\right)+1\right)$ pebbles. To this end, we place the $\left(p\left(\left\lceil\frac{k}{2}\right\rceil\right)+1\right)$-th pebble at position j, and we use the remaining $p\left(\left\lceil\frac{k}{2}\right\rceil\right)$ pebbles to check whether $w[1,...,j]$ belongs $P_{\left\lceil\frac{k}{2}\right\rceil}$. If this checking is positive, we pick up those $p\left(\left\lceil\frac{k}{2}\right\rceil\right)$ pebbles (we recycle those pebbles), and we use them to check whether $w[j+1,...,] \in P_{k-\left\lceil\frac{k}{2}\right\rceil}$. If this second checking is positive we accept, otherwise we pick up all the pebbles, we place the $\left(p\left(\left\lceil\frac{k}{2}\right\rceil\right)+1\right)$-th pebble at position $j+1$ and we execute the above procedure once again. We begin with $j=1$. We reject, if and only if, we place the $\left(p\left(\frac{k}{2}\right)+1\right)$-th pebble on the right end marker. ■

Corollary 133 *p-pal* $\in DSPACE\left(\log^2(n)\right)$.

It is time to prove that p-AG can be recognized using $O(\log(n))$ pebbles.

Theorem 134 *The containment*

$$NSPACE(\log(n)) \subset DSPACE\left(\log^2(n)\right)$$

holds.

Proof. We have to prove that p-AG belongs to $DSPACE\left(\log^2(n)\right)$. It is enough if we prove that p-AG can be recognized using a logarithmic number of pebbles.

Let

$$1^k\$1^s W 1^{t-1}\#\#\# w_G \#\#\#$$

be an instance of p-AG, and let us suppose that k is even. We use the same divide and conquer strategy used with pal^k. We can do this since:

There exists a path of length k between s and t, if and only if, there exists an intermediate node v such that s is connected to v by a path of length $\frac{k}{2}$, and v is connected to t by a path of length $\frac{k}{2}$.

Thus, we focus on locating the intermediate node v. Suppose we believe that $v = u$. We place a pebble on u (on the block $\#\#\#1^u\#\#\#$ that occurs once in W), and we use the remaining pebbles to check if there exists the necessary two paths of length $\frac{k}{2}$. We are not obligated to make those two checks simultaneously. Then, we use the remaining pebbles to look for the existence of a path of length $\frac{k}{2}$ connecting s and u, and hence, if such a path is detected, we pick up all the pebbles but the pebble that is placed on u, and we use those recycled pebbles to check for the existence of a path of length $\frac{k}{2}$ connecting u and t. We get that an instance

$$1^k\$1^s W 1^{t-1}\#\#\# w_G \#\#\#$$

requires just one more pebble than an instance

$$1^{\frac{k}{2}}\$1^x U 1^y \#\#\# w_G \#\#\#.$$

Altogether, we get that

$$1^k\$1^s W 1^{t-1}\#\#\# w_G \#\#\#$$

can be processed using $O(\log(k))$ pebbles, and we get that the full language p-AG can be processed using $O(\log(n))$ pebbles. The theorem is proved ■

References

[1] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. Journal of Computer and System Sciences, 4(2): 177–192, 1970.

[2] N. Immerman. Nondeterministic space is closed under complementation. SIAM Journal on Computing, 17(5): 935 - 938, 1987.

[3] R. Szelepcsényi. The method of forcing for nondeterministic automata. Bulletin of the EATCS, 33: 96 - 100, 1987.

13 P vs NP

With this lecture we arrive to one of the most important problems of theoretical computer science the so called P vs NP problem.

13.1 The Complexity Class P

The complexity class P is the set constituted by all the problems that can be solved by polynomial time deterministic Turing machines. It is said that P is constituted by the problems that can be solved efficiently. Thus, most people think that the notion of polynomial time deterministic Turing machine and the notion of efficient algorithm are almost the same notions. Why do we say that P is constituted by all the tractable problems? This is off course a convention, but this a convention is based on solid grounds:

1. Polynomial functions are functions of slow growth (though this cannot be said of a polynomial of degree 100 grows slowly).
2. The set of polynomial functions is closed under sum, multiplication and composition, and this implies that the set of deterministic Turing machines that run in polynomial time is closed under different forms of composition. The latter implies that P is closed under polynomial time reductions (see below).
3. Cobham noted in 1964 that the class P remained invariant when one chose a different feasible model of computation, that is: let $\mathcal{C}$ be one of the universal models of computation that was known at this time, and let $P_{\mathcal{C}}$ be the set of problems that can be solved by $\mathcal{C}$-machines that run in polynomial time, Cobham observed that for all those $\mathcal{C}$'s the equality $P=P_{C}$ holds (see [1]).

We will assume the above point of view in the remainder of this work, that is: we will assume that the complexity class P is constituted by all the tractable problems.

13.2 The Complexity Class NP

The reader should observe that most problems in mathematics, and most theorems in mathematics have an existential character: those problems ask for the construction of a structure with some special properties. We can cope with those problems using the following strategy:

- **Guess** a candidate.
- **Check** that this candidates holds the intended property.

If the set of candidates is infinite the above strategy gives place to nonhalting algorithms. If the set of candidates is a large finite set, this strategy gives place to inefficient brute force algorithms, and if the set of candidates is small (can be made small) we get an efficient algorithm.

Suppose that the input is a finite structure, and suppose that we are asked to compute a substructure exhibiting some special property. Observe that this scenario corresponds to the above second category: the set of substructures has exponential size. Most problems of combinatorial optimization have this kind of structure: most of those problems ask for the construction of substructures exhibiting some special properties. Most of those problems also have a second special feature: the structural property the defines the problem is easy to check. Consider the following examples:

- The Hamiltonian cycle problem.
- The Eulerian trail problem.
- The traveling salesman problem.
- The k-coloring problem.
- The clique problem.

We can solve any of those problems by guessing a substructure and then checking that this substructure holds the property that defines the problem. If the candidate does not satisfy this property, then we try a second candidate, and then a third, and so on. We halt if: either we find a substructure that satisfies the property, or we try all the possible candidates without finding a solution to the problem. By doing so, we are solving the existential problem by performing exponential many checks, that can be done, each one of them, in polynomial time. We can work in a sequential fashion and use exponential time, or we can use an exponential number of processors and solve the problem in polynomial time, or we can work nondeterministically with a machine that always do the correct guesses.

Definition 135 *The complexity class NP is the set of languages that can be recognized by nondeterministic polynomial time Turing machines.*

Exercise 136 *The complexity calls NP can be presented in terms of the notion of certificate. Let $R \subset \Sigma^* \times \Sigma^*$ be a binary relation. We say that R is p-balanced, if and only if, the following two conditions are satisfied:*

- *Let # be a character that does not belong to Σ. The language*

  $$L_R = \{x\#y : (x,y) \in R\}$$

  can be recognized in polynomial time.

- *There exists a polynomial function $p(n)$, such that given $x \in \Sigma^*$ there exists y such that $(x,y) \in R$, if and only if there exists y such that $(x,y) \in R$ and $|y| \leq p(n)$.*

Let $L_{\exists R}$ be the language $\{x : \exists y\,((x, y) \in R)\}$. Prove that L belongs to NP, if and only if, there exists a p-balanced relation R such that $L = L_{\exists R}$.

Let $x \in L_{\exists R}$, and let y be a string such that $(x, y) \in R$, we say that y is a certificate for x.

13.3 Cook's Theorem

Suppose we are given a language in NP, say L, and suppose that we are asked to decide if x belongs to L.

- We can choose to work deterministically, look hard for a certificate, say y, and then check that $(x, y) \in R_L$.
- Or, we can choose to work nondeterministically, skip the search for y, guess such a y (y is given by a friendly advisor), and then do the easy check: we **guess** a certificate, and we **check** that this is a correct certificate.

Remark 137 *We cannot implement the second option, what we do in practice is to simulate this strategy listing all the possible certificates and checking, one by one, whether there is one that works.*

It is very much harder to construct a mathematical proof than to check a mathematical proof. This suggests that the deterministic procedure takes very much longer than the nondeterministic one, and this suggests that NP contains problems that do not belong to P. Which problems? An easy answer to this question would be: the hardest problems in NP.

Definition 138 *Let $L \subset \Sigma^*$, and let $T \subset \Omega^*$, we say that L is polynomial time reducible to T, if and only if, there exists a polynomial time algorithm $\mathcal{M}$, which, on input $x \in \Sigma^*$, computes $r_x \in \Omega^*$ such that:*

$$x \in L, \textit{ if and only if, } r_x \in T.$$

Exercise 139 *Prove that P and NP are closed under polynomial time reductions.*

Exercise 140 *Suppose that L is polynomial time reducible T ($L \preccurlyeq_{pt} T$). It can be argued that T is at least as hard as L : if T belongs to P, the language L also belongs to P.*

We would say that L is the hardest problem in NP, if and only if, any problem in NP can be polynomial time reduced to L.

Exercise 141 *Choose one of your favorite problems in NP, and reduce it to SAT.*

It seems that many people did the same exercise around 1970, and it seems that most of them arrived to the same conclusion we arrived: it is easy to reduce a problem in NP to the problem SAT. This suggests that SAT is one of the hardest problems in NP.

Definition 142 *Let L be a language, we say that L is NP-hard, if and only if, any problem in NP can be polynomial time reduced to L. We say that L is NP-complete, if and only if, the problem is NP-hard and belongs to NP.*

Theorem 143 ***Cook's Theorem***
SAT is NP-complete.

Remark 144 *Leonid Levin independently proved an equivalent result [2].*

Remark 145 *Before Cook's paper there was not much interest in problems computable in polynomial time by nondeterministic Turing machines. The class NP only became an interesting class after Cook's Theorem.*

Exercise 146 *Suppose that L is NP-hard, and also suppose that $L \preccurlyeq_{pt} T$. Prove that T is NP-hard.*

Exercise 147 *Prove that P is equal to NP, if and only if, there exists a NP-complete problem in P.*

13.4 NP and co-NP

A language L belongs to co-NP, if and only, the language *co*-L belongs to NP.

Exercise 148 *Prove that co-NP is closed under polynomial time reductions.*

The complement of any language in NP is a problem in co-NP. A language L is co-NP complete, if and only if, L is in co-NP and for any problem in co-NP there exists a polynomial-time reduction from that problem into L. An example of such a problem is that of determining if a formula in propositional logic is a tautology: that is, if the formula evaluates to true under every possible assignment to its variables.

Exercise 149 *Prove that co-SAT is co-NP-complete.*

Consider the following problem:

Problem 150 ***Integer Factorization***

- *Input: (m, n), such that m and n are positive integers.*
- *Problem: decide if m has a factor k such that $1 < k \leq n$.*

This problem is one of the few problems that is known to belong to both NP and co-NP (but not known to be in P). It is easy to check that the problem belongs to NP: if m does have such a factor then the factor itself is a certificate. Membership in co-NP is also straightforward: one can just list the prime factors of m, all greater or equal to n, which the verifier can confirm to be valid by multiplication and by using the AKS primality test.

References

[1] A. Cobham. The Recognition Problem for the Set of Perfect Squares. SWAT (FOCS) 1966: 78 - 87, 1973.

[2] L. Levin. Universal search problems. Problems of Information Transmission (in Russian) 9 (3): 115116.

[3] N. Chomsky. On Certain Formal Properties of Grammars. Inf. Control. 2(2): 137 - 167, 1959.

[4] N. Chomsky. A Note on Phrase Structure Grammars. Inf. Control. 2(4): 393 - 395, 1959.

14 Automata Synchronization

Let L be a problem. Suppose that we suspect that L is NP-hard. This conjecture leads us to search for a NP-hardness proof for L. It follows from the definition that L is NP-hard, if and only if, SAT is polynomial time reducible to L. It also follows from the definition of NP-hardness that problem SAT is as reducible to L as any other NP-complete problem, that is: theory tells us that it is the same looking for a reduction of SAT into L as looking for a reduction of any other NP-hard problem into L. Theory tells us one thing and praxis tells us a completely different thing: if we want to prove that L is NP-hard, we should look into the enormous list of problems that are known to be NP-complete, and then choose those problems that exhibit structural analogies with problem L. Sometimes one can easily figure out a reduction of SAT (CNF-SAT) into L. Sometimes it is very much easier to figure out a reduction of a suitable NP-complete problem into L than a reduction from SAT.

We will to study two algorithmic problems related to automata. Those two problems are NP-hard, and they represent the two faces of this coin. The first problem is the *shortest synchronizing string problem* (SSS, for short), and the second one is the *separating words problem* (SW, for short).

It can be easily proved that SSS is NP-hard. The NP-hardness proof for this problem is a straightforward reduction of CNF-SAT into it (this reduction is one the most crystal-clear reduction this author knows). It can also be proved that SW is NP-hard. However, this second result is a recent result, and the NP-hardness proof is a reduction of a technical problem about finite semigroups into SW. It is true that one can use this reduction, and other auxiliary reductions, to construct a reduction of SAT into SW. We can search the literature and reconstruct the path that goes from SAT to SW. However, if we do the latter we will have to compose no less than four nontrivial polynomial time reductions.

14.1 Automata Synchronization

Let $\mathcal{M} = (Q, \Sigma, \delta)$ be a triple such that δ is a function from $Q \times \Sigma$ into Q. Observe that $\mathcal{M}$ is a DFA without initial and accepting states. Observe also that we can think of $\mathcal{M}$ as a finite labelled digraph, the labelled digraph $G_{\mathcal{M}}$ that is defined by:

- $V(G_{\mathcal{M}}) = Q$.
- Given $p, q \in Q$ there exists an edge (p, q) with label a, if and only if, the equality $\delta(p, a) = q$ holds.

We can think of $G_{\mathcal{M}}$ as a territory that is navigated by agents, whose software is equal to δ, and which interpret programs that belong to Σ^* : suppose agent x is placed at node q, and suppose that it has to execute the program $w \in \Sigma^*$, then after i time units this agent will be placed at node $\widehat{\delta}(p, w[1, ..., i])$. Now suppose that we have several agents scattered over $G_{\mathcal{M}}$, suppose that we do not know the specific position of each agent, and suppose that we want to send a message $w \in \Sigma^*$, the same message for all the agents, that leads all them to the same node q. What we are looking for is a synchronizing string for$\mathcal{M}$.

Definition 151 *Let $\mathcal{M} = (Q, \Sigma, \delta)$ be a DFA (without initial and accepting states), and let $w \in \Sigma^*$. We say that w is a synchronizing string for $\mathcal{M}$, if and only if, for all $p, q \in Q$ the equality $\widehat{\delta}(p, w) = \widehat{\delta}(q, w)$ holds. We say that $\mathcal{M}$ is a synchronizing automaton, if and only if, there exists a synchronizing string for $\mathcal{M}$.*

There are DFAs that are not synchronizing.

Exercise 152 *Show that any unary automaton without a sink state is not synchronizing. Prove also that any automaton with more than one sink is not synchronizing.*

Consider the following problem:

Problem 153 ***Recognition of Synchronizing Automata***

- *Input: $\mathcal{M}$, where $\mathcal{M}$ is a DFA.*
- *Problem: decide if $\mathcal{M}$ is a synchronizing automaton.*

It is easy to recognize the automata that are synchronizing, we have:

Theorem 154 *The problem RSA (recognition of synchronizing automata) belongs to P.*

Proof. Let $\mathcal{M}$ be a DFA. We say that $\mathcal{M}$ is pair-synchronizing, if and only if, for all $p, q \in Q$ there exists a string $w_{p,q}$ such that $\widehat{\delta}(p, w_{pq}) = \widehat{\delta}(q, w_{pq})$. It is easy to prove that $\mathcal{M}$ is synchronizing, if and only if, $\mathcal{M}$ is pair-synchronizing. Then, the problem RSA reduces to the problem of recognizing the automata that are pair synchronizing. Let $\mathcal{M}^{(2)}$ be the pair-automaton that is defined by:

- $Q^{(2)} = \{S \subset Q : |S| \leq 2\}$.
- $\delta^{(2)}$ is defined by the equation
$$\delta^{(2)}(S, a) = \{\delta(p, a) : p \in S\}.$$
  We have that $\mathcal{M}$ is pair-synchronizing, if and only if, the set
$$\left\{S \in Q^{(2)} : |S| = 1\right\}$$
  can be accessed from any other state of $\mathcal{M}^{(2)}$. Thus, the problem of recognizing the automata that can be synchronized is polynomial time reducible to an accessibility problem about digraphs. We know that this latter problem belongs to NL and we know that NL is included in P. The theorem is proved.

■

Let $\mathcal{M}$ be a synchronizing automaton. We define the reset treshold of $\mathcal{M}$ as

$$rt(\mathcal{M}) = \min\{|w| : w \text{ is synchronizing for } \mathcal{M}\}.$$

We have:

Exercise 155 *There is a naive strategy for the construction of synchronizing strings. This strategy goes as follows: we synchronize a pair p, q using a string w_{pq}. After that we synchronize a pair of the states that belong to the image of Q under the action of w. We continue in this way, working on pairs, until we synchronize all the states of $\mathcal{M}$. Prove that this naive strategy gives place to synchronizing strings of length $O\left(|Q|^3\right)$.*

Let us set

$$rt(n) = \max\{rt(\mathcal{M}) : \mathcal{M} \text{ is a } n\text{-state synchronizing DFA}\},$$

we have that $r(n) \in O\left(n^3\right)$. The famous Cerný Conjecture estates that $rt(n)$ is equal to $(n-1)^2$ (see [1]). Then, if this conjecture is true, there is a large gap between the length of the shortest synchronizing strings, and the length of the strings that can be constructed using the above sketched strategy. One of the ubiquitous questions of computer science is the question about the existence of algorithms that beat the most naive algorithmic solutions. Thus, it is natural to ask whether there exists a polynomial time algorithm for the construction of shortest synchronizing strings. Consider the problem:

Problem 156 ***Shortest Synchronizing Strings***

- *Input: $(\mathcal{M}, k)$, where $\mathcal{M}$ is a synchronizing automaton and k is a positive integer.*
- *Problem: decide if $rt(\mathcal{M}) \leq k$.*

We will prove that this problem is NP-hard.

14.2 Eppstein Reduction

D. Eppstein proved that SSS (shortest synchronizing strings) is NP-hard [**?**, **?**]. The proof is a reduction from CNF-SAT, and this is one of the most crystalline reductions this author knows. The reduction consists in representing the CNF α, given as input, as a labelled sink-digraph.

Theorem 157 *The problem SSS is NP-hard.*

Proof. Consider the propositional formula

$$(X_1 \wedge \neg X_2 \wedge \neg X_3) \vee (X_1 \wedge X_3)\,,$$

and consider the automaton

Let α be a CNF (like the above formula) with k clauses and n variables. The reduction consists in the drawing of a labeled digraph G_α (like the above automaton) which resembles the structure of α. The digraph G_α is made of k directed paths of length $n+1$ that are appended to a sink node s. The i-th path represents the i-th clause, and the node q_{ij} represents the way in which the variable X_i occurs in c_j :

- If X_i occurs positively in the clause c_j, there is a shortcut-edge from q_{ji} to s with label 1.
- If X_i occurs negatively in the clause c_j, there is a shortcut-edge from q_{ji} to s with label 0.
- If X_i does not occur in c_j, then there is not shorcut at all, and the two edges that go out from q_{ji} are directed toward the next node.

Observe that the digraph G_α is synchronizing. However, if formula α is not satisfiable the reset threshold of G_α is equal to $n+1$, while if α is satisfiable, the reset threshold is at most n. We get that the function

$$\alpha \mapsto (G_\alpha, n)$$

is a polynomial time reduction of CNF-SAT into SSS. The theorem is proved. ■

Exercise 158 *Write down a detailed description of the reduction used in the proof of the above theorem.*

References

[1] J. Cerný, A. Pirická, B. Rosenauerová. On directable automata. Kybernetika 7(4): 289 - 298, 1971.

[2] D. Eppstein. Reset Sequences for Monotonic Automata. SIAM J. Comput. 19(3): 500 - 510, 1990.

15 Separating Strings with Automata and Semigroups

We study the problem of separating strings with finite automata. This problem can be regarded as "the simplest computational problem you could ask to solve".

To begin with we suppose that there are two parties, say Alice and Bob, which can communicate with each other using a noiseless communication channel. We also suppose that those two parties have unlimited computational power.

First, we suppose that Alice has a string $w \in \{0,1\}^n$, and we suppose that she wants to communicate this string to Bob. What is the minimum number of bits that must be transmitted by Alice in order to achieve her goal? This is the question that is addressed by *Kolmogorov Complexity* [1]. We know that the answer is $O(K(w))$, where $K(w)$ is the Kolmogorov complexity of string w. Recall that $K(w)$ is equal to the length of the smartest (shortest) program that *separates* the string w.

Definition 159 *We say that a program (machine) separates the string w, if and only if, program $\mathcal{M}$ accepts w and rejects any other input string.*

It is known that there are strings whose Kolmogorov complexity is (almost) equal to its length, and it is also known that there are strings (like the string 1^n) whose Kolmogorov complexity is just logarithmic in the length of those strings. We have that K is a *string measure* that takes many different values when restricted to $\{0,1\}^n$. The latter is a feature that must be exhibited by any non-trivial measure.

Exercise 160 *Suppose for an instant that Alice is not allowed to compute smart programs separating the string w (she cannot compute unrestricted Turing machines), and suppose that she is restricted to compute deterministic finite state automata. Alice is obligated to communicate $\Omega(n)$ bits: prove that a minimal DFA separating the string w has $n+2$ states.*

Let us consider a second scenario. In this second scenario, Alice has a string $w \in \{0,1\}^n$, Bob has a string $u \in \{0,1\}^n$, and they must decide whether those two strings are equal. What is the minimum number of bits that must be communicated by Alice and Bob in order to achieve their common goal? This is the question that is addressed by *Communication Complexity* [2]. *To begin with we* notice that there is a naive solution to the problem that is being faced by Alice and Bob: Alice communicates to Bob $K(w)$ bits describing the string w. However, we know that there exists nondeterministic communication protocols that allow Alice and Bob to solve their problem by communicating a very much smaller amount of bits. Consider the following protocol:

1. *Alice guesses a position $i \in \{1, ..., n\}$.*

2. *Alice sends to Bob the string* $\widehat{i}w[i]$*, where* $\widehat{i}$ *is the binary code of* i*.*

3. *Bob checks if the equality* $w[i] = u[i]$ *holds.*

Notice that, if the equality $w = u$ holds, Alice and Bob will agree that those two strings are equal. On the other hand, if $w \neq u$ Alice can guess a position $i \leq n$ for which $w[i] \neq u[i]$. In the latter case Alice and Bob will agree that those two strings are different. Thus, we can conclude that the above nondeterministic protocol computes the equality function using $O(\log(n))$ communication bits. It is known that the latter problem requires $\Omega(\log(n))$ communication bits to be correctly solved in the nondeterministic sense [2].

Let us consider a third scenario. Suppose that Alice and Bob have an *unordered* pair of strings, say the strings $w, u \in \{0,1\}^n$, and suppose that Alice chooses one of those two strings. What is the minimum number of bits that must be communicated to Bob in order to let he knows which is the string chosen by Alice? It happens that the latter question is closely related to the former two questions:

If Alice wants to solve the latter problem efficiently, then she has to compute the smartest program *separating* u *from* w, and then she has to communicate a succinct description of this program to Bob.

Remark 161 *We say that program* $\mathcal{M}$ *separates* u *from* w*, if and only if, the final states of the computations of* $\mathcal{M}$ *on those two inputs are different states.*

Observe that the above solution to the *discerning problem,* as we call our problem, can be turned into a nondeterministic communication protocol that computes the equality function using (roughly speaking) the same number of communication bits. Consider the following protocol:

1. *Alice guesses a smart program* $\mathcal{M}$ separating u from w.

2. Alice runs $\mathcal{M}$, on input w.

3. *Alice sends to Bob the string* $\widehat{\mathcal{M}}o(\mathcal{M}, w)$*, where* $\widehat{\mathcal{M}}$ *is a short description of program* $\mathcal{M}$ *and* $o(\mathcal{M}, w)$ *is the final state of the computation of* $\mathcal{M}$ *on input* w*.*

4. *Bob runs* $\mathcal{M}$*, on input* u*, and he checks if the equality* $o(\mathcal{M}, w) = o(\mathcal{M}, u)$ *holds.*

It is easy to transform a nondeterministic protocol for the equality function into a deterministic protocol that solves the discerning problem using (roughly speaking) the same number of communication bits.

Let $u, w \in \{0,1\}^n$, and let $S(w, u)$ be the minimum number of bits that must be communicated by Alice in order to solve the discerning problem for the pair u, w. We get from the latter observation that $S(u, v) \in O(\log(n))$. Moreover, we have that there are strings $w, u \in \{0,1\}^n$ such that $S(u, v) \in \nleq(\log(n))$.

15.1 Separating Strings with Finite Automata

Suppose for an instant that Alice is not allowed to compute smart programs, and that she is restricted to compute DFA's. We use the symbol $S_{DFA}(u,v)$ to denote the number of bits that must be communicated by Alice in this latter scenario. We are interested in the asymptotic behavior of function S_{DFA}.

Notice that a n-state DFA can be fully described using $O(n\log(n))$ bits. We can forget the logarithmic factor and focus our attention on the functions $Sep : \Sigma^* \times \Sigma^* \rightarrow \mathbb{N}$ and $sep : \mathbb{N} \rightarrow \mathbb{N}$ that are defined by

$$\begin{aligned} Sep(u,w) &= \text{the number of states of a minimal DFA separating } u \text{ and } w, \\ sep(n) &= \max\{Sep(u,w) : u,w \in \{0,1\}^n\}. \end{aligned}$$

We are interested in upper and lower bounds for function *sep*. Let us begin with two naive bounds.

Lemma 162 *Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a function such that $f(n)\log(f(n))$ belongs to $\Theta(\log(n))$, we have that:*

1. $sep(n) \in \Omega(f(n))$.
2. $sep(n) \leq n+2$.

Proof. The lower bound comes from the connections between the discerning problem and nondeterministic communication complexity that were discussed at the end of the previous section. The upper bound is obvious. ■

Observe that the gap between those two bounds is exponential. Thus, there is a lot of room for improvements. It seems that the first published reference on this subject is an old paper of Goralcik and Koubek [3]. This paper includes the first non-trivial upper bound for function *sep* : Goralcik and Koubek proved that $sep(n) \in o(n)$. Robson proved, shortly after, that $sep(n) \in O\left(n^{\frac{2}{5}}\log^{\frac{3}{5}}(n)\right)$, and it is the best known upper bound for function *sep* [4].

Notice that the gap between those two bounds is still exponential, and notice that those bounds have not been improved after more than thirty years. We heard that Jeffrey Shallit offered 100 pounds to the first person that improves one of those bounds.

15.1.1 The Goralcik-Koubek Bound

It is easy to prove that $sep(n) \in \Omega(\log(n))$. To do the latter one can consider the set of strings

$$GK = \left\{\left\{1^n 0^{n+\mathrm{lcm}(1,\ldots,n)}, 1^{n+\mathrm{lcm}(1,\ldots,n)}0^n\right\} : n \geq 1\right\}.$$

Notice that the differences between those strings are somewhat obvious. Therefore, we are tempted to conjecture that there must be pairs of strings that are very much harder to separate. It is a little bit frustrating to know that

$\Omega(\log(n))$ is the best known lower bound for function *sep*, and that this bound is achieved by the above set of pairs that we call GK pairs.

Can we shorten the gap between upper and lower bounds for function *sep*? It could happens that one of the two bounds is sharp, either the lower bound or the upper bound. Thus, if we want to shorten the gap between those two bounds, we will have to carefully decide which of those two bounds can be improved. Can complexity theory help us to take a suitable decision?

15.2 The Problem SW

Consider the algorithmic problem

Problem 163 ***SW***

- *Input:* (u, v, k)*, where* u, v *are strings of the same length and* k *is a positive integer.*
- *Problem: decide if* $sep(u, v) \leq k$.

What is the algorithmic complexity of problem SW? The latter problem is closely related to the *dfa consistency problem*, which asks for the size of a minimal automaton separating two finite sets of strings. Notice that SW is the restriction of the latter problem to sets of size 1. It is known that the dfa consistency problem is NP-hard [5]. Thus, it is reasonable to conjecture that SW is also NP-hard problem. Let us suppose that SEP is NP-hard.

15.2.1 NP-hardness of SEP

We prove that SEP is NP-hard. We use an important connection between finite automata and finite transformation semigroups.

Definition 164 *Let* $k > 0$. *A transformation semigroup over the set* $\{1, ..., k\}$ *is a subset of* $\{1, ..., k\}^{\{1,...,k\}}$ *that is closed under composition.*

The set $\{1, ..., k\}^{\{1,...,k\}}$ is itself a transformation semigroup, and it is called the *full transformation semigroup* of order k. We use the symbol $\mathcal{T}_k$ to denote this semigroup.

Definition 165 *Let* Σ *be a finite alphabet, an interpretation of* Σ *in the semigroup* $\mathcal{T}_k$ *is a function* $\alpha : \Sigma \to \mathcal{T}_k$ *that assigns to each letter in* Σ *an element of* $\mathcal{T}_k$. *Given* $w \in \Sigma^*$, *and given* α, *we interpret* w *as the function*

$$f_{w,\alpha} = \alpha(w[n]) \circ \cdots \circ \alpha(w[1]).$$

Definition 166 *We say that a pair of strings, say the pair (u, v), is an identity for $\mathcal{T}_k$, if and only if, for all interpretation*

$$\alpha : \{0, 1\} \rightarrow \mathcal{T}_k$$

the equality $f_{u,\alpha} = f_{v,\alpha}$ holds.

Suppose that (u, v) is and identity for $\mathcal{T}_k$. We have:

- Given $i \leq k$, the pair (u, v) is an identity for $\mathcal{T}_i$.
- If $u \neq v$, there exists k such that (u, v) is not an identity for $\mathcal{T}_k$.

Thus, it makes sense to define the function

$$I(u, v) = \max \{k : (u, v) \text{ is not an identity for } \mathcal{T}_k\},$$

and it also makes sense to consider the following algorithmic problem

Problem 167 ***IDENT***

- *Input: (u, v, k), where u, v are strings and k is a positive integer.*
- *Problem: decides if $I(u, v) \geq k$.*

Exercise 168 *Let L be a language in NP. Prove that L is NP-hard, if and only if, L is co-NP hard.*

Volkov et al proved that IDENT is co-NP hard [6]. We use this result to prove that SW is NP-hard.

Theorem 169 *SW is NP-hard.*

Proof. Let (u, v, k) be an instance of IDENT. We only have to prove that the latter pair is an identity of $\mathcal{T}_k$, if and only if, this pair cannot be separated using k states. The latter equivalence follows from the well studied relation between finite state automata an transformation semigroups [6]. Let Σ be the alphabet of u and v. Suppose that $\Sigma = \{X_1, ..., X_n\}$. Given

$$f_1, ..., f_n : \{1, ..., k\} \rightarrow \{1, ..., k\},$$

those two functions determine a DFA with input alphabet Σ. We use the symbol $\mathcal{M}_{f_1,...,f_n}$ to denote this automaton which is defined by

$$\mathcal{M}_{f_1,...,f_n} = (\{1, ..., k\}, \{0, 1\}, \delta),$$

where the transition function δ is defined by

$$\delta(i, a) = f_i(a).$$

Notice that $\mathcal{M}_{f_1,...,f_n}$ separates the pair (u, v), if and only if, the equality $f_{u,\alpha} = f_{v,\alpha}$ does not hold, with α the interpretation given by the equalities

$$\alpha(i) = f_i, \text{ for all } i \in \{1, ..., n\}$$

We get the first implication: if (u, v) is an identity for $\mathcal{T}_k$, then the pair (u, v) cannot be separated with k states. On the other hand, given a k-state automaton $\mathcal{M}$ with input alphabet $\{X_1, ..., X_n\}$, we suppose that the set of states Q is equal to $\{1, ..., k\}$. If we do the latter, we get that automaton $\mathcal{M}$ determines an interpretation $\alpha_{\mathcal{M}} : \{X_1, ..., X_n\} \to \mathcal{T}_k$ which is given by:

$$\alpha_{\mathcal{M}}(i)(s) = \delta_{\mathcal{M}}(s, i).$$

Notice that $\mathcal{M}$ separates the pair (u, v), if and only if, the equality $f_{u,\alpha_{\mathcal{M}}} = f_{v,\alpha_{\mathcal{M}}}$ does not hold. We get the second implication, and the theorem is proved. ■

The NP hardness of SEP can also be obtained from a reduction of the problem SAT into the former problem. However, it seems that the most natural of those reductions is the composition of no less than three reductions that involve some technical problems about finite groups and finite transformation semigroups.

References

[1] M. Li, P. Vitanyi. An introduction to Kolmogorov Complexity and Its Applications. Springer Verlag, Berlin, 2008.

[2] E. Kushilevitz, N. Nisan. *Communication Complexity.* Cambridge University Press, 1996.

[3] P. Goralcik, V. Koubek. On discerning words by automata. In L. Kott, editor, Proc. 13th Int'l Conf. on Automata, Languages, and Programming (ICALP), volume 226 of Lecture Notes in Computer Science, pages 116-122. Springer-Verlag, 1986.

[4] J. Robson. Separating strings with small automata. Inform. Process. Lett, 30: 209-214, 1989.

[5] M. Gold. Complexity of automata identification from given data. Information and control, 378(3): 302-320, 1978.

[6] J. Almeida, M. Volkov, S. Goldberg. Complexity of the identity checking problem for nite semigroups. J. Math. Sciences, 158(5): 605 - 614, 2009.

[7] A. Bulatov, O. Karpova, A. Shur, K. Startsev. Lower Bounds on Words Separation: Are There Short Identities in Transformation Semigroups? The Electronic Journal of Combinatorics 24(3), 2017.

16 Matrix Semigroups and Quantum Automata

Suppose we are compiling small finite automata from regexes. We enter a regex α and we get a small NFA $\mathcal{M}_\alpha$ with 1000 states. We try to construct a DFA for L_α and we realize that all our attempts are leading us to automata with something like 2^{1000} states. We have to take into account that this big number of states could be greater than the number of particles in the galaxy. Thus, we get obligated to accept (understand) that all those deterministic automata solving problem L_α cannot take place in our universe. However, we really want to solve problem L_α, and it seems that $\mathcal{M}_\alpha$ is our single possible choice. We get obligated to run $\mathcal{M}_\alpha$ in despite of its nondeterministic nature. We ask: how can we simulate DFA's and NFA's in digital computers?

16.1 Simulations of One-Way Automata and Matrix Semigroups

Let $\mathcal{M} = (Q, q_0, A, \Sigma, \delta)$ be a finite state recognizer. Automaton $\mathcal{M}$ is a solution to the algorithmic problem encoded by language $L(\mathcal{M})$. Let us suppose that $\mathcal{M}$ is our solution. We want to use this solution to recognize the strings in $L(\mathcal{M})$. The existence of $\mathcal{M}$ is not the same as $\mathcal{M}$ processing a long list of strings $w_1, ..., w_n$. We have to run $\mathcal{M}$ on those strings, and this means that we have to simulate $\mathcal{M}$ using our technological means. The architecture of $\mathcal{M}$, with its tape and its working head, tempts us to construct a small golem that walks along the tape taking annotations in a small blackboard (the golem might carry some pebbles!). Notice that we need a semi-infinite tape to carry over our construction. It is better if we use a compact representation of this golem that uses the transition digraph of $\mathcal{M}$. This time the golem walks over the transition digraph using as travel route the input string it is processing. Thus, if $\mathcal{M}$ has 1000 states, we construct a small digraph of size 1000 and we let our golem walk on this digraph. Suppose that $\mathcal{M}$ is a NFA. If we apply the power construction we end with a digraph of size 2^{1000} that wont find room in our galaxy. We can try a different approach. This time we have a divisible golem walking on $G(\mathcal{M})$. Suppose a copy of this golem meets a transition

$$\delta(q, a) = \{q_1, ..., q_k\}.$$

Then, the copy makes k copies of itself, and the first copy go to q_1 the second to q_2 and so on. This works for short time. After a moderate long period of time we ends with 2^{1000} golems trying to walk over the hypercongested digraph. If we learned something with the power construction we have a way to solve the latter problem: each time several copies of a golem meet at a node, we can fuse all those copies into a single copy. If we apply the fusion rule we ensure that there will be no more than 1000 golems at each instant of the computation. Thus, we conclude that the deterministic simulation of NFA's can be done with few nodes, few space and few golems.

We have to notice that all the above simulations are analogical simulations of the dynamics encoded by $\mathcal{M}$. Analogical simulations are expensive and unreliable. Thus, it is better if we think in use our digital technology to run symbolic simulations of the aforementioned dynamics.

16.1.1 An Old Theorem of Cayley

Let $\mathcal{M} = (Q, q_0, A, \Sigma, \delta)$ be a DFA and suppose that $Q = \{1, ..., n\}$. The connection between automata and semigroups tells us that we can represent the dynamics of $\mathcal{M}$ as the action of a semigroup of transformations over a set of size n. An old theorem of Cayley tells us that we can represent those actions as the action that is exerted by a suitable set of Boolean matrices over the canonical basis of $\mathbb{R}^n$. The latter means that:

1. Given $i \leq n$, we use the canonical vector

$$e_i = \left(0, 0, ...0, \underbrace{1}_{i\text{-th entry}}, 0, ..., 0\right)$$

to represent the state i.

2. Let $a \in \Sigma$, letter a defines a function T_a that goes from $\{e_1, ..., e_n\}$ to $\{e_1, ..., e_n\}$. This finite function can be though as the restriction of a linear function T_a to the canonical basis. This linear map has a matrix representation that we denote with the symbol M_a. Matrix $M_a = [a_{ij}]_{i,j \leq n}$ is the Boolean matrix

$$a_{ij} = \begin{cases} 1, \text{ if } \delta(j, a) = i \\ 0, \text{ otherwise} \end{cases} .$$

We use matrix M_a as the representation of a.

3. Let $a_1 \cdots a_m \in \Sigma^*$. We represent the action of this string over the state q_1 (let us suppose that q_1 is the initial state) by

$$\left(M_{a_n} M_{a_{n-1}} \cdots M_{a_2} M_{a_1}\right) \cdot e_1.$$

The equivalence

$$\widehat{\delta}(q_1, a_1 \cdots a_m) = j, \text{ if and only if, } \left(M_{a_n} M_{a_{n-1}} \cdots M_{a_2} M_{a_1}\right) \cdot e_1 = e_j$$

holds for any string in Σ^*.

We get, from the above three items, a symbolic representation of the dynamics of $\mathcal{M}$. We can use this representation to simulate on a digital computer the computations of DFA's.

We can work with NFA's in a similar way. Suppose that $\mathcal{M}$ is nondeterministic and suppose that $\delta(i, a) = \{i_1, ..., i_k\}$. We represent the image of i under

a with the vector $e_{i_1} + \cdots + e_{i_k}$. The action of a on Q gets represented by the matrix M_a^{ND} that is defined by:

$$a_{ij} = \begin{cases} 1, \text{ if } i \in \delta(j, a) \\ 0, \text{ otherwise} \end{cases} .$$

Exercise 170 *Let $a_1 \cdots a_m \in \Sigma^*$. Prove that the equivalence*

$$j \in \widehat{\delta}(q_1, a_1 \cdots a_m), \text{ if and only if, } \pi_j \left(M_{a_n}^{ND} M_{a_{n-1}}^{ND} \cdots M_{a_2}^{ND} M_{a_1}^{ND} \cdot e_1 \right) = 1,$$

where π_j is the j-th projection that sends the vector $(v_1, ..., v_n)$ in v_j.

Let $\mathcal{M} = (Q, q_0, A, \Sigma, \delta)$ be a NFA with 1000 states, and suppose that the size of Σ is equal to m. We represent $\mathcal{M}$ with m small Boolean matrices of order 1000 which can be quickly composed on a digital computer. We use those matrices and basic matrix algebra to simulate the nondeterministic computations of $\mathcal{M}$.

Remark 171 *If we begin with a regex of length n, we end with a tuple $(M_{a_1}, ..., M_{a_m})$, where each one of those M's is a Boolean matrix of order $O(|\alpha|)$.*

16.2 Probabilistic One-Way Automata

What does the vector e_i with all its 0's and its isolated 1 represent? It could represent that automaton $\mathcal{M}$ is in state i with probability 1. If we assume this point of view, we would be able to accept that $\mathcal{M}$ reaches a *meta-state* like $(p_1, ..., p_n)$, with:

1. $p_i \geq 0$.
2. $\sum_{i \leq n} p_i = 1$.

Now, if the states reached by $\mathcal{M}$ are stochastic vectors, then the action of the letters on those stochastic vectors must send them onto vectors that are also stochastic.

Exercise 172 *Let M be a $n \times n$ matrix with nonnegative entries. We say that M is stochastic if all the columns of M sum up to 1. Prove that stochastic matrices send stochastic vectors onto stochastic vectors.*

Definition 173 *A n-state one-way probabilistic automaton over Σ (PFA, for short) is a tuple*

$$\mathcal{P} = (\{1, ..., n\}, A, \{P_a : a \in \Sigma\})$$

such that A is a subset of $\{1, ..., n\}$ and for all $a \in \Sigma$ the matrix P_a is stochastic of order n.

Let w be a string of length k. The action of w on the initial state is given by

$$P_{w[k]} \cdot \cdots \cdot P_{w[1]} \cdot e_1.$$

We say that $\mathcal{P}$ accepts w with probability δ, if and only if, the inequality

$$\sum_{i \in A} \pi_i \left(P_{w[k]} \cdot \cdots \cdot P_{w[1]} \cdot e_1 \right) \geq \delta$$

holds. Let $\delta < \frac{1}{2}$, we say that $\mathcal{P}$ accepts the language L with error δ, if and only if, the following two conditions hold:

1. For all $w \in L$ the automaton $\mathcal{P}$ accepts w with probability at least $1 - \delta$.
2. For all $w \notin L$ the automaton $\mathcal{P}$ accepts w with probability at most δ.

Does randomness provide computational power to one-way finite state automata? The answer to this question is negative. There does not exist a nonregular language L a PFA $\mathcal{P}$ and a $\delta < \frac{1}{2}$ such that $\mathcal{P}$ accepts L with error δ.

PFA's can use very small gaps to distinguish between a language L and its complement *co*-L. This supposed ability of PFA's could lead us to think that those automata might separate a nonregular language from its complement. However, we have to observe that this ability of PFA's is just apparent

Exercise 174 ***Probability amplification.*** *Let L be a language that is accepted by a PFA $\mathcal{P}$ with error δ. Let $\varepsilon < \frac{1}{2}$, prove that there exists a PFA $\mathcal{P}_\varepsilon$ that accepts L with error ε.*

Exercise 175 *Investigate about* ***Myhill-Nerode Theorem.***

Theorem 176 *PFA's cannot accept nonregular languages.*

Proof. Let L be a nonregular language and suppose that it is accepted by a PFA $\mathcal{P}$. We can suppose that L is accepted with error $\delta < \frac{1}{4}$. We invoke Myhill-Nerode's Theorem to ensure the existence of two infinite sets of strings $\{w_i : i \geq 1\}$ and $\{l_{ij} : i, j \geq 1\}$ such that for all $i, j \geq 1$ it happens that either $w_i l_{ij} \in L$ and $w_j l_{ij} \notin L$, or $w_j l_{ij} \in L$ and $w_i l_{ij} \notin L$. Let us pick $i, j \geq 1$. String l_{ij} defines a function $L_{ij}^{\mathcal{R}}$over the set of n-dimensional stochastic vectors. Function $L_{ij}^{\mathcal{R}}$ is defined by

$$L_{ij}^{\mathcal{R}}(X) = \mathcal{R}_{l_{ij}[|l_{ij}|]} \cdot \cdots \cdot \mathcal{R}_{l_{ij}[|l_{ij}|]} \cdot X.$$

Notice that $L_{ij}^{\mathcal{R}}$ is continuous. String w_i defines an analogous function $L_{w_i}^{\mathcal{R}}$ that is also continuous.

Given $\varepsilon > 0$ there exists η such that if $\|X - Y\| < \eta$ then $\left\| L_{ij}^{\mathcal{R}}(X) - L_{ij}^{\mathcal{R}}(Y) \right\| < \varepsilon$. We can pick $\eta < \frac{1}{4}$ such that for infinite many pairs the following two conditions hold:

1. For all X, Y stochastic vectors, if $\|X - Y\| < \eta$ then $\left\| L_{ij}^{\mathcal{R}}(X) - L_{ij}^{\mathcal{R}}(Y) \right\| < \frac{1}{2}$.

2. $\left\| L_{w_i}^{\mathcal{R}} \cdot e_1 - L_{w_j}^{\mathcal{R}} \cdot e_1 \right\| < \eta$.

Let us pick one of those infinite many pairs, say the pair i, j; we get that:

1. $\left\| L_{w_i}^{\mathcal{R}} \cdot e_1 - L_{w_j}^{\mathcal{R}} \cdot e_1 \right\| < \eta$ and $\left\| L_{ij}^{\mathcal{R}} \left(L_{w_i}^{\mathcal{R}} \cdot e_1 \right) - L_{ij}^{\mathcal{R}} \left(L_{w_j}^{\mathcal{R}} \cdot e_1 \right) \right\| < \frac{1}{2}$.

2. $L_{ij}^{\mathcal{R}} \cdot \left(L_{w_i}^{\mathcal{R}} \cdot e_1 \right) \in L$ and $L_{ij}^{\mathcal{R}} \cdot \left(L_{w_j}^{\mathcal{R}} \cdot e_1 \right) \notin L$.

Notice that the last two assertions contradict each other. The theorem is proved. ■

16.3 Nonregular Language and Quantum Supremacy

If we think in terms of probabilities we get that stochastic vectors are well suited to represent states. If we think in terms of quantum amplitudes we get that unitary vectors are well suited to to represent states (quantum states). We can use this idea to define many different models of quantum automata. All those models use unitary matrices to define the transitions. This is so because the unitary matrices of order n are the matrix representations of the n-dimensional linear maps that leave invariant the set of n-dimensional unitary vectors. Thus, the dynamics of a quantum automaton with n-states is driven by a finite set of unitary matrices of order n. The action of those unitary operators on the set of initial configurations determines the set of outcomes of the automaton. We have to measure (project) those n-dimensional vectors in order to take a decision. Quantum automata are equipped with unitary maps that lead the dynamics and with projections that are used to measure the outcomes of those unitary dynamics. Let us introduce one the simplest quantum architectures.

Definition 177 *A n-state one-way quantum automaton over Σ (QFA, for short) is a tuple*

$$\mathcal{U} = \left(\{1, ..., n\}, \{U_a : a \in \Sigma\}, \pi \right).$$

Given $a \in \Sigma$ the matrix U_a is unitary of order n. The symbol π denotes a projection of $\mathbb{R}^n$ onto $\mathbb{R}$.

Let w be a string of length k. The action of w on the initial state is given by

$$U_{w[k]} \cdot \cdots \cdot U_{w[1]} \cdot e_1.$$

Definition 178 *Let $\mathcal{U}$ be a QFA and let w be a string. We say that $\mathcal{U}$ accepts w, if and only if, the condition*

$$\pi \left(U_{w[k]} \cdot \cdots \cdot U_{w[1]} \cdot e_1 \right) = 0$$

holds.

We say that $\mathcal{U}$ accepts the language

$$L(\mathcal{U}) = \{w : \mathcal{U} \text{ accepts } w\}.$$

PFA's cannot accept nonregular languages. We could have great expectations regarding PFA's, but those expectations came to nothing because PFA's cannot accept nonregular languages. We noticed that PFA's cannot recognize nonregular languages because those automata cannot really operate with small gaps. QFA's can distinguish between strings based on the existence of negligible gaps. We prove that there exist nonregular language that are accepted by QFA's.

Let BAL be the language

$$\{w \in \{0,1\}^n : \#\ 0\text{'s in } w \text{ is equal to the number of } 1's\}$$

Exercise 179 *Prove that BAL is nonregular.*

Theorem 180 *There exists a QFA that accepts the language BAL with zero error.*

Proof. We can use a two-state QFA over the real numbers to recognize the language BAL. Let

$$\mathcal{U} = (\{1,2\}, \{U_a : a \in \{0,1\}\}, \pi)$$

be the QFA that is defined by:

- Let $\alpha \in (0,1)$ be an irrational number. Let U_0 be the unitary operator that corresponds to a clockwise rotation of $\mathbb{R}^2$ by an angle of $\alpha\frac{\pi}{2}$ radians.
- Let U_1 be the unitary operator that corresponds to an anticlockwise rotation of $\mathbb{R}^2$ by the same angle as U_0.
- π is the projection of $\mathbb{R}^2$ onto $\langle e_2 \rangle$ (onto the y-axis).

Let w be a string that belongs to BAL. Suppose that the number of 0's in w is equal to m. We get that

$$U_w \cdot e_1 = U_{w[2m]} \cdot \cdots \cdot U_{w[1]} \cdot e_1 = U_0^m \cdot U_1^m \cdot e_1.$$

The equality on the right follows from the fact that two rotations in $\mathbb{R}^2$ commute. Moreover, we have that

$$\pi(U_w \cdot e_1) = \pi(U_0^m \cdot U_1^m \cdot e_1) = \pi(I \cdot e_1) = 0,$$

and we get that $\mathcal{U}$ accepts w.

Let w be a string in *co*-BAL. Let m be the number of 0's and let n be the number of 1's. Let us suppose that $m > n$. We have that

$$U_w \cdot e_1 = U_0^{m-n} \cdot e_1,$$

and we have that $U_0^{m-n} \cdot e_1$ is the vector obtained after applying to e_1 a rotation of $(m-n)\,\alpha\frac{\pi}{2}$ radians. Recall that α is irrational. We get that $(m-n)\,\alpha\frac{\pi}{2}$ cannot be a integer multiple of 2π, and we get that $U_w \cdot e_1$ cannot be equal to e_1. We get as a consequence that $\pi\,(U_w \cdot e_1) \neq 0$. $\blacksquare$

The irrationality of α is a crucial factor in the construction of $\mathcal{U}$. If we consider noisy versions of $\mathcal{U}$ that work with rational numbers that are close to α, we get automata that accepts some strings in *co*-BAL. The power, and the possible supremacy, of quantum automata strongly depends on the accuracy with which those automata are prepared. This could be a serious obstacle for quantum computing. We have to observe that perfect accuracy can be achieved in some cases.

Exercise 181 *Construct an irrational angle like $\alpha\alpha\frac{\pi}{2}$ using ruler and compass alone.*

References

[1] I. Chuang, M. Nielsen. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge UK 2010.