Two-way finite automata with restricted nondeterminism and descriptional complexity

Internship report – May-August 2011

Bruno Guillon
Internship director : Giovanni Pighizzini

Università degli studi di Milano

Acknowledgements : many thanks to professor Giovanni Pighizzini for its welcome and for having directed my internship. Many thanks also to Carlo Mereghetti and Beatrice Palano and all organizing committee of NCMA workshop for having given me the possibility to assist at this workshop. Many thanks also to Viliam Geffert for his contribution to our work. Thanks to Antonio Capoduro for coffees and pleasant discussions, and more generally thanks to all persons encountered at university, for welcoming me.

Contents

Introduction

One important aim in theoritical computer science is to answer the question: *how difficult is a problem.* To approach this question, and so to "measure" the difficulty of problems, researchers have used different ways. The *descriptional complexity* area, in which the internship took place, is one of these ways. Contrary to the *complexity* theory, the question is not to know how many space or time a computation machine needs to resolve a problem, rather, what is the required size of a given model to do that. This can be viewed as the question of how difficult to explain is a problem.

This area was introduced by A. R. Meyer and M. J. Fisher in the 70s [19]. This approach is general, meaning that all calculation models can be studying with this question in aim, however the majority of works (and also this internship) have be done interesting in *finite automata* models (see Section 1.2). Finite automata model is one of the simplest calculation model, so it is a good start point for research. A lot of variant of the classical finite automata have been introduced, like two-ways (we allow the input head to change direction), nondeterministic or pushdown automata. One important aspect of these area is to compare required size of different models for a given problem.

W. J. Sakoda and M. Sipser have observed and presented [23] an analogy, by introducing descriptionnal complexity classes (see Section 1.3.2), between descriptionnal complexity theory and *complexity theory.* They also raise two main questions, still open, with an analogy with the well known $P \overset{?}{=} NP$ open problem. The links between both theories was enforced by P. Berman and A. Lingas [1] by proving a relation with the well known open problem about space bounded Turing machines: L = NL. Other relations have been found later, like in [6] or in this internship (see Theorem 3 in Section 2).

This three month internship took place at the end of the first year of master PENSUNS (from May to August 2011), at *Università degli studi di Milano*, and was directed by professor Giovanni Pighizzini. It was naturally composed of three times: (1) I started (3-4 weeks) to discover the descriptionnal complexity area, with many readings (see Bibliography), from generality to details in a more restricted area (that is *two-ways finite automata*). This first time ended with a presentation of general view of these readings. (2) Then in a second time (5-6 weeks), I start to search new results[1]. Some of these new results, which are in fact a generalisation of existing results from unary case, are good candidates for an article. (3) So, in the last time (3 weeks), I wrote an article (still a draft) presenting these results. This third time will probably be continued by e-mails

[1]One of these main found results, not presented in this report, turned out not to be new. This result was the proof of the existence of a language $\mathcal{L}$ over binary alphabet, accepted by a $O(n)$-state *sweeping automaton* but rather neither this language nor its complement was accepted by a *one-way nondeterministic automaton* with fewer than 2^{n-1} states (see Section 1.2 and 1.4 for definitions). An analog result (with other proof) was already published.

exchange, to finish the redaction of this article, and propose it for publication. At these three times, I add also the workshop *NCMA*, which took place in Milan, the 18th and 19th July, and at which I was allowed to assist (for free), thanks to the organizing committee.

The first time of my work is presented briefly in the first part of this introduction and also in Section 1 and Bibliography. The two following times are presented in Section 2 (some parts of the article are reused in this report). In Conclusion I will speak about my experiment of this stage.

1 Definitions and readings

1.1 Descriptional complexity

As said in introduction, I started the internship with many readings in order to discover the general descriptional complexity area, and then to be able to choose the restricted area, for my work, that is the case of two-way finite automata.

The first article I red [7] give me a survey of the area. It also give me the main results, especially about finite automata. The article of A.R. Meyer and M.J. Fisher [19], was the historical introduction of the area, and give me its general aim. Then I red a more recent paper [15], that was written in tribute to late Chandra Kintala, which was one of the co-founder of descriptional complexity. Therefore many of his results and general ideas are given in this article.

The most often, in descriptional complexity, we try to compare different theoretical computation machines. Given two classes $\mathcal{M}_1$ and $\mathcal{M}_2$ of computation machines and two functions *size* from $\mathcal{M}_i$ to $\mathbb{R}_+$, there exists two main family of results :

- **Upper bounds :** if we can simulate every machine of $\mathcal{M}_1$ of size n by a machine of $\mathcal{M}_2$ with size $f(n)$, we say that $f(n)$ is an upper bound for the reduction $\mathcal{M}_1 \to \mathcal{M}_2$.
- **Lower bounds :** if there exists a machine M_1 of $\mathcal{M}_1$ of size n, which cannot be simulate by a machine M_2 of $\mathcal{M}_2$ with size less than $f(n)$, then $f(n)$ is said to be a lower bound of the reduction $\mathcal{M}_1 \to \mathcal{M}_2$.

For example, if $\mathcal{M}_1$ and $\mathcal{M}_2$ are classes respectively of usual[2] deterministic finite automata (DFA), and nondeterministic finite automata (NFA), and both functions size are the number of states, then, it is known that 2^n is an upper bound for the reduction DFA$\to$ NFA (by the subset construction, see [21] for details), and it is also a lower bound 2^n (we say that 2NFAs are exponentially more succinct than 2DFAs). When an upper bound is also a lower bound, like in this example, we say that the upper bound is tight, and we speak about the best lower or upper bound.

[2]Definition of finite automata are given later, in Section 1.2.

1.2 Generalities on Finite Automata

Then I started to read articles on more precise subject. So I studied finite automata, and especially *two-way finite automata*, which are equivalent to *read-only Turing machines*.

Finite automata is an old calculation model. It is well known for its simplicity (in case *one-way*). In this subsection, we presents different variants of the usual finite automaton (i.e., the *one-way deterministic finite automaton*). We start our definition from the most general main finite automaton, which is *nondeterministic* and *two-way*, and then we will define the other usual finite automata, by restriction of it.

Definition 1

A two-way nondeterministic finite automaton *(*2NFA *for short) is a quintuple* $(Q, \Sigma, \delta, q_{start}, F)$ *where* Q *is the set of state,* Σ *is an alphabet (i.e., a finite set of symbols),* δ *is a function from* $(Q \times (\Sigma \cup \{\vdash, \dashv\}))$ *to* $\mathcal{P}(Q \times \{\triangleleft, \triangledown, \triangleright\})$*, called* transition function *where* $\vdash, \dashv \notin \Sigma$ *are respectively the* left *and* right endmarker*,* $q_{start} \in Q$ *is the initial state, and* $F \subseteq Q$ *is the set of accepting states.*

At each *step*, the automaton read an input symbol (a symbol $c \in \Sigma \cup \{\vdash, \dashv\}$ from an input tape), change it state and move its input head backward, forward or keeps it stationary, depending on whether δ return with $\triangleleft$,$\triangleright$ or $\triangledown$ respectively (we will speak about respectively $\triangleleft$-, $\triangleright$-, and $\triangledown$-*transitions*). The input head cannot go outer of the input, that means that we have no transitions of the form $(\_, \triangleleft) \in \delta(\_, \vdash)$ or $(\_, \triangleright) \in (\_, \dashv)$.

Given a word $w = w_1, w_2, \ldots, w_{|w|} \in \Sigma^*$, we extend it by $w_0 = \vdash$ and $w_{|w|+1} = \dashv$, the two endmarkers.

During a computation, the automaton can change its state and can move the head to the left, to the right or keep at the same position. So, in order to describe the situation of the automaton at a fixed time, we only have to know its head position, and its state. That is why we define configurations:

Definition 2

A configuration is a pair (q, h) *with* $q \in Q$ *and* $h \in \{0, 1, \ldots, |w| + 1\}$.

The *initial configuration* is $(q_{start}, 0)$ that means that the automaton begins in state q_{start} with the head scanning w_0, that is the left endmarker. In general, for configurations $(q, 0)$ (resp. $(q, |w| + 1)$), we will speak about *left-endmarker-configurations* (resp. *right-endmarker-configurations*) (we also will use general *endmarker-configurations* term for the union). A *successor* of a configuration (q, h) is a configuration (q', h') such that $(q', d) \in \delta(q, h)$ with d equal to $\triangleleft$, $\triangledown$ or $\triangleright$ depending on whether $h' - h$ is respectively equal to -1, 0 or $+1$. If there exists no successor, configuration (q, h) is said to be *halting*. We define symmetrically the term *predecessor*.

Definition 3

A path of $\mathcal{A}$ *on* w *is a possibly infinite sequence of configurations* $\{c_0; \ldots; c_m\}$,

$m \in (\mathbb{N} \cup \{\infty\})$ such that: for each i, $0 \leq i < m$, c_{i+1} is a successor of c_i.

Let $P = \{(q_0, h_0); \ldots; (q_m, h_m)\}$ be a finite path (we speak about *path from* (q_0, h_0) *to* (q_m, h_m)). If there is no possible *extensions* of the path, (i.e., configuration (q_m, h_m) has no successor), we speak about *maximal path* (we also say that all infinite paths are maximal).

A configuration is said *reachable* if there exists a path from the initial configuration to it (else it is said *unreachable*). We now introduce *computation*, which describe the calculation of the automaton on a word.

Definition 4

A computation *is a possibly infinite maximal path starting from the initial configuration.*

Observe that, by definition, a computation contains only reachable configurations. And reciprocally, all reachable configurations belong to at least one computation.

We say that a 2NFA $\mathcal{A}$ *accepts* a word $w \in \Sigma^*$ if there is a computation $\{(q_0, h_0); (q_1, h_1); \ldots; (q_m, h_m)\}$ of $\mathcal{A}$ on $\vdash w \dashv$ such that for some $j \leq m,\ q_j \in F$ (such a computation is said to be *accepting*). The language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ accepted by the automaton $\mathcal{A}$ is the set of all words accepted by $\mathcal{A}$. We say that two automata are *equivalent* if they accept the same language.

Observe that remove all transitions from each accepting states (so enforce accepting states to be halting) does not change the accepted language.

Lemma 1

We can always suppose that all accepting states are halting.

With this assumption, accepting computations are finite and have exactly one accepting configuration, moreover this is the last of the sequence.

In order to simplify our proofs, we prove the following lemma, which claims that we can remove $\triangledown$-transitions.

Lemma 2

For all 2NFAs we can remove $\triangledown$-transitions without changing the accepted language, only by modifying the transition function (so without increasing the number of states).

Proof: We define the subset $S_{p,c}$ of state sequences as follow: $\{p_1; p_2; \ldots; p_s\} \in S_{p,c}$ if and only if $p_1 = p$ and $\forall i,\ 1 < i \leq s,\ (p_i, \triangledown) \in \delta(p_{i-1}, c)$ and for $i \neq j,\ p_i \neq p_j$. $S_{p,c}$ can be seen as the set of all sequences of states that can be reached from (p, c), without entering any loop and using only $\triangledown$-transitions. Let $M_{p,c}$ be the set of all $\triangleleft$- and $\triangleright$-transitions of $\delta(p, c)$. We transform the transition function δ of the given automaton into a function δ' as follow:

if there is a state $q_f \in F$ that appear in a sequence of $S_{p,c}$ then $\delta'(p, c) = \{(q_f, \triangleleft)\}$ (except if $c = \vdash$, in this case we use $\triangleright$).

else $\delta(p, c) = \bigcup_{q \text{ appears in } s \in S_{p,c}} M_{q,c}$

With this transition function δ', the new automaton accepts the same language, and never uses $\triangledown$-transitions. ■

We now look at some special behaviors.

Definition 5

We say that a computation has a loop if it contains a repeated configuration.

We can easily prove the following result about accepting computations and loops by cutting sequences.

Property 1

One can simplify all accepting computations into an accepting computation without loop.

Note that, because there are a finite number of configurations, there are finitely many computations without loops. This fact is essential in our following proofs.

Let us now define some restricted models. The automaton is said:

- *one-way* if neither $\triangleleft$- nor $\triangledown$-transitions are used[3],
- *deterministic* if for each symbol σ of $\Sigma \cup \{\vdash, \dashv\}$ and each q, $|\delta(q, \sigma)| \leq 1$,
- *halting* if there exists a path from each reachable configurations to an halting state,
- *unambiguous* if for each word there is at most one accepting computation.

Note that, with the given definition, an halting automaton can have infinite computation, we only require that it always has the possibility to join an halting configuration.

We also define *self-verifying automata* (SVFAs; 1SVFAs and 2SVFAs respectively one-way and two-way): a SVFA is a NFA which has three types of answer: *yes*, *no* and *I don't know*. By definition, we ensure that the automaton can always answer and never lies, that is, for each word w, exactly one of these two cases holds:

1. there is an accepting computation (i.e., a computation with answer *yes*).
2. there is a rejecting computation (i.e., a computation with answer *no*).

In fact such an automaton can recognise a language and its complement.

[3]In fact, by Lemma 1, one can easily simulate every automaton using only $\triangledown$- and $\triangleright$-, but no $\triangleleft$-transitions by a one-way automaton.

1.3 Problems raised in descriptional complexity and finite automata theory

It is known that all these variants of finite automata accept the same class of language, i.e., the regular languages (see [8]). However, in descriptional complexity, we will see that they have different succinctness, and so they are separated.

For the majority of works (and also ours), the *size* function chosen is the number of states (i.e. $|Q|$), which will be use for all the end of this report.

1.3.1 One-way is simpler

In case of one-way finite automata (1FAs), it is known, as said in Section 1.1, that upper bound for the reduction 2NFA$\to$ 2DFA is 2^n, and that this bound is tight. The upper bound is given by power set construction (see [21]), and we give here a witness language for the lower bound: the language L_n of binary ($\Sigma = \{0, 1\}$) words which have a 1 in position $l - n$ where l is the length of the word can easily be accepted by a n state 1NFA (just guess nondeterministically the position $l - n$ and check it), but no accepting 1DFA have less than 2^n states. Note that the size of the alphabet is constant in n.

1.3.2 Problems raised and analogy with complexity theory

W. Sakoda and M. Sipser have introduced in the 70s [23] classes of families of languages. They observed with this system an analogy with complexity theory.

We define the class $1D$ as the set of all family of languages $(L_h)_h \in \mathbb{N}$ such that some polynomial $p(i)$ bounds the size of the minimal accepting 1DFA of language L_i. Analogously, we define classes $2D$, $1N$, $2N$, $1SV$, $2SV$, respectively for 2DFA, 1NFA, 2NFA, 1SVFA, 2SVFA.

From these definitions, the two authors raised two main questions, still open:

Questions 1

- *Is $2N$ equal to $2D$?*
- *Is $2D$ equal to $1N$?*

Because 2DFAs and 1NFAs can be seen as particular 2NFAs, we already know that $2D \subseteq 2N$ and $1N \subseteq 2N$. The previous result, from Subsection 1.3.1, claims that $1D \subsetneq 1N$.

W. Sakoda and M. Sipser define also a reduction between two language sequences, analog to the polynomial reduction from complexity theory: given two alphabets Δ and Δ' and languages $L \subset \Delta^*$ and $L \subset \Delta'^*$, we say that L homomorphically reduces to L' (*h-reducible* for short) if there is a map $g : \Delta \to \Delta'^*$ and $i, f \in \Delta'^*$ such that for any string $s = s_1 s_2 \cdots s_k$, ($s_j \in \Delta$), $s \in L$ iff $i g(s_1) g(s_2) \cdots g(s_k) f \in L'$. Now we are able to define our reduction for language sequences: given two language sequences $(L_h)_{h \geq 1}$ and $(L'_h)_{h \geq 1}$, we say that $L \leq_h L'$ if there is polynomial p such that for each i, L_i is h-reducible to L'_j for some $j \leq p(i)$.

It easy to see that all classes $2D$, $2N\dots$ are closed under $\leq_h$.

In analogy to the complexity theory, the classes $2D$, $2N\dots$ play the role of the complexity classes P, $NP\dots$ (P and NP denote the classes of languages accepted by a Turing machine, respectively, in deterministic and nondeterministic polynomial time), and the reduction $\leq_h$ has the role of the polynomial reduction. Therefore the question $2D$ equal to $2N$ is analog to the well known open problem P $\overset{?}{=}$ NP. In their article, Sakoda and Sipser also present two family of languages complete for $2D$ and $2N$ thanks, thanks to the reduction $\leq_h$, analog to the well known SAT problem.

The analogy between complexity and descriptional complexity theories, is enforced by some theorems which give explicit relation, especially with a second well known question in complexity theory, that is L $\overset{?}{=}$ NL (L and NL denote the classes of languages accepted, respectively, in deterministic and nondeterministic logarithmic space) (see [1, 6, 14] and Theorem 3).

1.4 Different approaches of Sakoda & Sipser's main questions

In order to approach the two main questions raised by Sakoda and Sipser, known to be difficult by analogy and relation with difficult problems of complexity theory, researchers have introduced many variants of automata[4].

In 1980, M. Sipser introduced *sweeping automata* (SA) [25], which are 2DFAs but with head reversals only at the endmarkers (so with restricted bidirectionality). He proved that 1NFAs can be exponentially more succinct than SAs. However, P. Berman [2] and S. Micali [20] independently proved that this does not solve the general problem: in fact the simulation of 2DFAs by SAs requires an exponential number of states.

The result of Sipser was generalized by Hromkovic and Schnitger [9] considering *oblivious machines* and, recently, by Kapoutsis [13] considering "few reversal" 2DFAs. However, even these last restricted models have been proved to be less succinct than 2DFAs. Hence the general problem remains open.

A different kind of restriction has been investigated in the literature, starting in 2003 with a paper by Geffert *et al.* [4]. They considered the unary case, namely the case of automata with a one letter input alphabet. Under this strong restriction, the problem of Sakoda and Sipser looks also difficult. Furthermore, it is connected with the open question L $\overset{?}{=}$ NL in complexity theory. More precisely, in [4] the authors proved that each n-state unary 2NFA can be simulated by an equivalent 2DFA with $O(n^{\lceil \log_2(n+1)+3 \rceil}) = n^{O(\log n)}$ states, hence obtaining a subexponential but still superpolynomial upper bound. We do not

[4]We do not present all results seen in readings during the internship. In fact, we don't speak about [3, 17, 18, 9, 10, 12], because the results are not interesting for our work presented in Section 2, but the reading of these articles was a important time at the beginning of the internship.

know whether or not that simulation is tight. However, a positive answer to this question would imply the separation between the classes L and NL. In fact, in [6] it was shown that if L = NL then each unary 2NFA with n states can be simulated by a deterministic automaton with a number of states polynomial in n. (For further connections between the question of Sakoda and Sipser and the problem $L \overset{?}{=} NL$ we address the reader to [1, 14]).

Using similar techniques, in [6] the authors also proved that each unary 2NFA can be made unambiguous with a polynomial increasing in the number of the states.

Along these lines of investigation, Geffert *et al.* considered in [5] the problem of the complementation of unary 2NFAs. They proved that for each n-state 2NFA accepting a unary language L there exists a 2NFA with $O(n^8)$ states accepting the complement of L. They also discuss the relationships between the problem of the complementation of 2NFAs and the problem of Sakoda and Sipser. The proof of this result was given using inductive counting arguments and a kind of normal form for unary 2NFAs which allows to restrict nondeterministic choices and head reversals only at the endmarkers.

Kapoutsis [11] considered the problem of the complementation in the case of machines with general input alphabets, but with restrictions on the head reversals. He proved that the complementation of *sweeping* 2NFAs (namely, 2NFAs with input head reversal *only* at the endmarkers) requires exponentially many states, thus emphasizing an important difference with the unary case.[5]

1.5 Two-way restricted finite automata

In the internship we further consider the case of general alphabets, but under a restriction which is different from those investigated so far. From one hand, we do not put any constraint on the head movement, *i.e.*, we allow head reversals not only at the endmarkers, but at any input position. On the other hand, we permit nondeterministic choices only when the input head is scanning one of the endmarkers. We call these models *two-way restricted nondeterministic finite automata* (2RNFAs).

Definition 6

A 2RNFA *is a* 2NFA $(Q, \Sigma, \delta, q_{start}, F)$ *such that, for each* $q \in Q$*:*

$$c \neq \vdash, \dashv \quad \Rightarrow |\delta(q, c)| \leq 1$$

We show that several results obtained in the case of unary 2NFAs can be extended to 2RNFAs with *any* input alphabet. These results are presented in this section.

[5]Due to the above mentioned normal form, with a *linear increasing* in the number of the states, in unary 2NFAs it is always possible to restrict nondeterministic choices and head reversals only at the endmarkers. Hence from the result on the complementation of unary 2NFAs in [5] follows that the complementation of unary sweeping 2NFAs can be done with a polynomial number of states.

In order to simplify our proofs, we start to prove the following lemma, which give us a normal form for 2RNFA.

Lemma 3

For every n-state 2RNFA $\mathcal{A}$ there exists an equivalent $(3n)$-state 2RNFA $\mathcal{A}'$ with the following properties:

1. *There is a unique accepting state q_{accept} which is also halting,*
2. *q_{accept} is reachable only from the left endmarker by $\triangledown$-transition,*
3. *nondeterministic choices can occur only at the left endmarker,*
4. *$\triangledown$-transitions can occur only at the left endmarker, to reach q_{accept}.*

Proof: Let $\mathcal{A} = (Q, \Sigma, \delta, q_{start}, F)$ be a n-state 2RNFA. We suppose F nonempty, otherwise, we are in the trivial case of empty language.

First, observe that we can remove all $\triangledown$-transitions by Lemma 2, without increasing the number of state. So assume that $\mathcal{A}$ does not have such a transition. We define an automaton $\mathcal{A}'$ equivalent to $\mathcal{A}$ which has properties (1)-(3), and property (4) will be implied by this previous observation. Let $q_{accept} \notin Q$ be a new state, and let $\overleftarrow{Q} = \{\overleftarrow{q},\ q \in Q\}$ and $\overrightarrow{Q} = \{\overrightarrow{q},\ q \in Q \backslash F\}$ be two new state sets. We define the transition function δ' of $\mathcal{A}''$ as follow: $\forall c \in \Sigma,\ \forall q \in Q$

- if $q \in F$ then $\delta'(q, c) = \{(\overleftarrow{q}, \triangleleft)\}$
  else $\delta'(q, c) = \delta(q, c)$
- $\delta'(q, \dashv) = \{(\overleftarrow{q}, \triangleleft)\}$
- if $q \in F$ then $\delta'(q, \vdash) = \{(q_{accept}, \triangledown)\}$
  else $\delta'(q, \vdash) = \delta(q, \vdash)$
- $\delta'(\overleftarrow{q}, c) = \{(\overleftarrow{q}, \triangleleft)\}$
- $\delta'(\overrightarrow{q}, c) = \{(\overrightarrow{q}, \triangleright)\}$
- $\delta'(\overrightarrow{q}, \dashv) = \{(q, \triangleleft)\}$
- if $q \in F$ then $\delta'(\overleftarrow{q}, \vdash) = \{(q_{accept}, \triangledown)\}$
  else $\delta'(\overleftarrow{q}, \vdash) = \{(\overrightarrow{p}, \triangleright),\ (p, \triangleleft) \in \delta(q, \dashv)\}$

All other transitions, not given in the previous list of rules, are set to be empty (for completeness). In this construction, we use the set $\overleftarrow{Q}$ and $\overrightarrow{Q}$ to transfer state information to the left endmarker. If the state is accepting, then the left endmarker goes to q_{accept}, if not, that means that the transfer comes from the right endmarker, and the automaton simulates, possibly using nondeterminism, with this state information, the initial transition at the right endmarker, and then transfer the new state information, using $\overrightarrow{Q}$. By construction, the automaton $\mathcal{A}' = ((Q \cup \overleftarrow{Q} \cup \overrightarrow{Q} \cup \{q_{accept}\}), \Sigma, \delta', q_0, q_{accept})$ proves the lemma. ■

Let us now observe a computation of such an automaton on word $w = w_1 w_2 \dots w_l$, with convention $w_0 = \vdash$ and $w_{l+1} = \dashv$. The previous lemma ensures us to have in each computation, a sequence of deterministic parts (when the head don't cross the left endmarker). That is why we define *segments*, which correspond to such a part.

Definition 7

We call segment, a subsequence $(q_k, h_k), \dots, (q_{k+j}, h_{k+j})$ of a computation, such that: $w_{h_k} = w_{h_{k+j}} = \vdash$ and for every i, $k < i < (k+l) \Rightarrow w_{h_j} \neq \dashv$.

2 Results

2.1 Reach

One crucial point of all our results is, given two states s and t, to deterministically answer the question of whether we can reach t from s in exactly one segment. Starting from an n-state 2RNFA $\mathcal{A}$ in the normal form given by Lemma 3, we will now construct a family $(REACH(\mathcal{A})_{s,t})_{(s,t)\in Q\times Q}$ of 2DFAs such that: $REACH(\mathcal{A})_{s,t}$ accepts a word w if and only if there is a one segment path of $\mathcal{A}$ from s to t reading w.

A first idea, in order to construct such an automaton, is to simulate directly and deterministically, using the initial automaton, all segments starting from s, and verify whether t is encountered at the left endmarker or not. But, our initial automaton is allowed to enter in a deterministic loop, and so never answer (never cross the left endmarker again). To avoid this problem, we shall now present the construction of the halting 2DFA $REACH(\mathcal{A})_{s,t}$, by suitably refining Sipser's construction for space bounded Turing machines [24]. We suppose to have a linear order on Q. We give here only the main idea by recalling the original Sipser's construction.

Recall of the original Sipser's construction

We want to transform a 2DFA $\mathcal{A}$ into an equivalent halting 2DFA $\mathcal{A}'$. Assume that $\mathcal{A}$ starts with the initial state q_0 at the left endmarker, and accepts in a unique accepting state q_f only at the left endmarker. For each $w \in \Sigma^*$, $\mathcal{A}$ accepts w if and only if there is a "backward" path, following the history of the accepting computation in reverse, from the unique accepting configuration $(q_f, 0)$ to the unique initial configuration $(q_0, 0)$.

Consider the graph whose nodes represent configurations and edges computation steps. Since $\mathcal{A}$ is deterministic, the component of the graph containing $(q_f, 0)$ is a tree rooted at this configuration, with backward paths branching to all possible predecessors of $(q_f, 0)$. In addition, no backward path starting from $(q_f, 0)$ can cycle (hence, it is of finite length), because the halting configuration $(q_f, 0)$ cannot be reached by a forward path from a cycle.

Thus, the equivalent halting machine can perform a depth-first search of this tree in order to detect whether the initial configuration $(q_0, 0)$ belongs to the predecessors of $(q_f, 0)$. If this is the case, the simulator accepts. On the other hand, if all the tree has been examined without reaching $(q_0, 0)$, this means that there are no path from $(q_0, 0)$ to $(q_f, 0)$ and so the w is not in the language, so the simulator rejects.

The depth-first search strategy visits the first (in some fixed lexicographic order) immediate predecessor of the current configuration that has not been visited yet. If there are no such predecessors, the machine travels along the edge towards the unique immediate successor. (Traveling forward along an edge is simulated by executing a single computation step of $\mathcal{A}$, traveling backward is a corresponding "undo" operation).

For this search, the simulator has only to keep, in its finite control, the state q related to the currently visited configuration (q, i) (the input head position i is represented by its own input head), together with the information about the previously visited configuration (its state and input head position relative to the current head position, i.e., a number ± 1). Hence, the simulator uses $O(n^2)$ states.

Reach

Now we briefly present our improvements to this procedure and problems encountered. Here, t plays the role of q_f, and we suppose $t \neq q_{accept}$. Hence, predecessors of t are right predecessors (by Lemma 3, configuration $(t, \vdash)$ is not reachable by $\triangledown$-transitions). Therefore, until reaching the next left endmarker configuration, the sequence of predecessors is a tree rooted in $(t, \vdash)$, because of determinism in such parts.

Because we want to check segments, we don't have to look at predecessors of left endmarker configuration (because our search stops at left endmarker). So when we reach a left endmarker configuration, we just test whether this configuration is $(s, \vdash)$ or not: if it is, we directly accept, else we cut the subtree rooted in it, and we continue the search.

Our automaton examines at each time a current configuration in two modes: (1) examination of the left predecessors, (2) examination of the right predecessors. As explain in the recall of Sipser's construction, this can be done by remembering only the state of q of the current examined configuration.

For each $q \in Q$, we create states $q\nwarrow$, $q\downarrow_1$, $q\nearrow$ and $q\downarrow_2$, and we also add a new accepting state q_f. So the state set of our automaton is

$$Q' = \{q\nwarrow,\, q\downarrow_1,\, q\nearrow,\, q\downarrow_2 : q \in Q\} \cup \{q_f\}.$$

The meaning of these states is:

$q\nwarrow$ Starting state for Mode 1: examination of left predecessors of (q, i). The input head is positioned in $i - 1$. If $i - 1$ is the position of the right endmarker, then we just check, as said before, if $(q, \triangleright) \in \delta(s, \vdash)$.

$q\downarrow_1$ Finishing state of Mode 1: all left predecessors of (q, i) have been examined. The input head is positioned in i. The transition function will start the Mode 2, except if i is the position of the right endmarker (if it is, it directly finishes the Mode 2 in $q\downarrow_2$).

$q_{\nearrow}$ Starting state of Mode 2: examination of right successors of (q, i). The input head is positioned in $i + 1$.

$q_{\downarrow_2}$ Finishing state of Mode 2: all predecessors (in at most one segment) of (q, i) have been examined (and $(s, \vdash)$ didn't appears). The transition function will start the examination of the next (according to linear order) predecessor of direct successor of (q, i). This examination will start in state Mode 1. If no such configuration exists, depending of whether the successor of (q, i) is left or right, the automaton finishes Mode 1 or 2 of examination of this successor.

q_f Accepting state, which is reached only if configuration $(s, 0)$ is encountered.

By Lemma 3 we know that we can reach q_{accept} only by stationary move. So $REACH_{s,q_{accept}}$ just have to test whether $(q_{accept}, \triangledown) \in \delta(s, \vdash)$. (We can do this for each $s \in Q$ with 2DFA with only 1 states.)

We have given here general ideas to obtain our family $(REACH(\mathcal{A})_{s,t})_{s,t \in Q \times Q}$ which has, by construction, the following wanted property.

Property 2

$\forall w \in \Sigma^*$ and $\forall s, t \in Q$, $REACH(\mathcal{A})_{s,t}$ accepts w if and only if we can reach t from s in exactly one segment by using the transition function of $\mathcal{A}$ reading w. Moreover $REACH(\mathcal{A})_{s,t}$ has $O(n)$ states.

Starting from the normal form of Lemma 3, and using the $REACH$ automata family, we are now able to prove results of following subsections (we will only give some general ideas).

In all this subsections, we will start from a n-state 2RNFA $\mathcal{A} = (Q, \Sigma, \delta, q_{start}, \{q_{accept}\})$ in normal form.

2.2 Polynomial reduction from 2rnfa to 2svfa

By Property 1, we know that for all words of accepted language, there is an accepting computation without loop. Such a computation has at most $|Q|$ segments (otherwise two left configurations are equal, and so we have a loop).

So the main idea is to use inductive counting[6], and simulate recursively a finite number (maximum |Q|) of segments, checking at each recursive call, if we can reach the accepting state.

To reader's ease of mind, we prefer to present the simulating 2SVFA in the form of algorithm, written in high-level code. We will then informally discuss the actual implementation, evaluating the number of states required. In the following code, we use two subroutines:

[6]The inductive counting technique is based on the following idea: instead of remembering many objects (that can have exponential cost), we only remembering their number, and try to recover them using nondeterminism and some linear order.

- *simulation*(k): a nondeterministic function, returning a nondeterministically chosen state q that is reachable by a computation path of $\mathcal{A}$ in exactly k segments from the initial configuration. The call of this function may also abort the entire simulation by halting in a "don't-know" state $q_?$, due to a wrong sequence of nondeterministic guesses, if the chosen path halts too early, not having completed k segments. This subroutine can be simulated using a direct simulation of our initial automaton $\mathcal{A}$ (so using $O(n)$ states).

- *reach*(s,t): a deterministic function. It returns *true*/*false*, depending on whether the state t can be reached from the state s in exactly one segment. This subfunction does exactly the same work as the automaton $REACH_{s,t}$ presented in Section 2.1 (so by Property 2 we already know that we can do this work with a $O(n)$-state 2DFA for each pair of state (s,t)).

The nondeterministic simulation algorithm, based on the well-known inductive counting technique, is displayed here:

Algorithm 1: *Main simulation*

```
$m' \leftarrow 1$;
for $k \leftarrow 1$ to $|Q| - 1$ do
    $m \leftarrow m'$; $m' \leftarrow 0$;
    for $each\ t \in Q$ do
        for $i \leftarrow 1$ to $m$ do
            $s \leftarrow simulation(k)$;
            if $(i > 1\ and\ s \leq s_{prev})$ then halt in $q_?$;
            $s_{prev} \leftarrow s$;
            if $reach(s,t)$ then
                if $t = q_{accept}$ then halt in $q_{\text{yes}}$;
                $m' \leftarrow m' + 1$;
                $break$;
            end
        end
    end
end
halt in $q_{\text{no}}$;
```

Basically, the algorithm proceeds by counting, for $k = 0, \ldots, |Q| - 1$, the number of states reachable by $\mathcal{A}$ at the left endmarker by all computation paths starting from the initial configuration and composed by exactly k segments. As a side effect of this counting, the algorithm generates all states reachable at the left endmarker, and hence it can correctly decide whether to accept or reject the given input (so the algorithm is *self-verifying*).

Because the main algorithm has 7 variables (of Q), and because both subroutines need only one more variable (by a good implementation of subroutine *reach* reusing variables s and t of the main algorithm), the implementation of the algorithm by a finite automaton requires only $O(n^8)$ states (that is polynomial).

Theorem 1

For all n-state 2RNFA $\mathcal{A}$ *there exists a* $O(n^8)$*-state* 2SVFA $\mathcal{A}'$, *that accepts the same language as* $\mathcal{A}$.

2.3 Subexponential reduction from 2rnfa to 2dfa

Again using Property 2, we prove now that we can simulate each 2RNFA by a 2DFA, which has a subexponential number of states.

We also give the 2DFA in form of an algorithm written in high-level code. This algorithm, called *reachable*, is based on the divide(and-conquer technique. Hence it is recursive and it reuses the same deterministic subroutine *reach*(s,t) as previously, for its base case ($k = 1$). *reachable*(s, t, k) answers the question of whether there is an at most k segments path from s to t.

Algorithm 2: *reachable*(s, t, k)

```
18 if $k$=1 then
19     if $p$=$q$ then return true
20     else return reach(s,t)
21 end
22 else
23     for each state $r \in Q$ do
24         if reachable($s, r, \lceil k/2 \rceil$) then
25             if reachable($r, t, \lceil k/2 \rceil$) then return true
26         end
27     end
28     return $false$
29 end
```

Observe first that this algorithm is deterministic, and because of Property 1, the call of *reachable*$(q_{start}, q_{accept}, |Q|)$ will answer the question of reachability of $(q_{accept}, 0)$ that is the question of whether w is in the accepted language or not. So *reachable* "accepts" the same language as $\mathcal{A}$.

To explain how implement this algorithm with a 2DFA, we first implement it with a *pushdown automaton*. At each recursive call, the pushdown automaton save on the top of its pushdown the pair of states s, t and the integer $k \leq |Q|$.

In fact, if we allow the automaton to read all the pushdown[7], only one state has to been saved (one don't change) with a binary information (*prefix* or *suffix*), and the integer can be recover from the size of the pushdown.

The maximal push down high is $\lceil \log_2(n-1) \rceil$, not counting the activation of the "main program", i.e., the predicate *reachable*$(q_{start}, q_{accept}, n-1)$. And because, unlike in classical divide-and-conquer technique, the problem of size k is not divided into two subproblems of size $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$, but, rather, both the outer and inner if statements (line 24 and 25) use the same parameter

[7]Such a device, which is less restrictive than a pushdown, is sometimes denoted by the term *stack* in the literature.

$\lceil k/2 \rceil$. This ensures that, whenever a bottom level of the recursion is reached,the pushdown is of the same height.

To implement it with a 2DFA, we store the stack information into the state. Because of the logarithmic maximal pushdown high, we obtain a subexponential size:

Theorem 2

For each n-state 2RNFA $\mathcal{A}$, there exists an equivalent 2DFA $\mathcal{A}'$, which needs only $O(n^{\lceil \log_2(n-1) \rceil -1})$ states.

So we have prove here an upper bound for the reduction 2RNFA $\rightarrow$ 2DFA, which is subexponential but not polynomial.

2.4 Polynomial reduction from 2rnfa to 2dfa, under hypothesis L= NL

First we define *computation graphs.* Recall that $\mathcal{A}$ is a 2RNFA (in normal form), and w is a word of Σ^*. Let $G(w) = (V, E)$ be an oriented graph with:

- $V = Q$
- $E = \{(p, q) \backslash REACH_{p,q} \textit{ accepts } w\}$

In other words, (p, q) is an edge of G if we can reach q from p in exactly one segment.

Remark 1

The size of the graph is independent on the length of the input word w. This size is always n^2 (size of the adjacency matrix).

By definition, the word w is accepted by $\mathcal{A}$ if and only if there exists an accepting computation of $\mathcal{A}$ on w. That means that there exists a sequence of connected segments, which starts with state q_{start} and ends with state q_{accept}. So, in other words $(G(w), q_{start}, q_{accept})$ is an instance of the well known *graph accessibility problem* (GAP).

Assuming that L= NL, there exists D_{GAP} a deterministic logarithmic space bounded Turing machine accepting GAP. We can assume that D_{GAP} has a two-way read-only input tape which contains a representation of a graph with N vertices, given by its adjacency matrix.

Now, our goal is to simulate this Turing machine on all graphs $G(w)$, by a 2DFA $\mathcal{A}'$ with a polynomial number of states. The unique difference between a 2DFA and a deterministic Turing machine, is that the first model is not allowed to write on a tape.

The deterministic Turing machine D_{GAP} uses only logarithmic space. So our main idea is to simulate this logarithmic space in the finite control of our automaton. So assuming that the size of the output alphabet of D_{GAP} is m and the size of the input graph is k, our automaton needs $O(m^{\log_2(k)})$ states to simulate this output tape.

In our case, with $G(w)$ as input graph, Remark 1 claims that, this number does not depend on the size of the input word w, but only of n. Thus this number is $m^{\log_2(n^2)} = n^{2\log_2(m)} < n^K$, for a constant K.

Now our automaton has also to keep in its finite control the state of D_{GAP}, which is also a constant.

With all this previous remark, we are able to simulate D_{GAP} with a 2DFA which has a number of state polynomial in n, on input $G(w)$. Let us now explain how obtain this input from w.

Recall that we have supposed that the input graph is given by its adjacency matrix. So each cell of the input correspond to a couple p, q of states of Q (i.e., p and q are respectively the indices of row and column of the cell of the adjacency matrix of $G(w)$, that D_{GAP} is currently reading), and its value is a boolean depending on whether (p, q) is an edge of G. This question is, by definition, equivalent to the question of whether we can reach q from p in exactly one segment.

We can answer these questions deterministically, using $O(n)$ states for each pair p, q, by using the $(REACH_{s,t})_{s,t \in Q\times Q}$ automata family, presented in Section 2.1. So our automaton is now able, reading w, to simulate D_{GAP} on $G(w)$: at each step, the automaton instead of reading the read tape, perform the subautomaton $REACH_{p,q}$, supposing that (p, q) informations are saved in its current state. The return of this subautomaton give it the value of the corresponding cell of the adjacency matrix of $G(w)$.

Observe that the current configuration (input head position and state) of D_{GAP} is completely stored in state informations of our automaton, using the previous pair of states p, q to encode the position of the input head and a state r of D_{GAP} to encode itself. So the input head of our automaton moves only when subautomata $REACH$s are called. The move of the D_{GAP} input head are replaced, in our simulation, by changing the p, q components of the state, not changing the position of our input head. That ensure us to not have problems to start and stop $REACH$s subautomata with the input head at the good position (it always starts and halts, both at the left endmarker).

Our automaton uses so less than $n^2 \times j \times n^K$ states, where j is the number of states of D_{GAP}. So this is less than $n^{K'}$ for a constant K'. This conclude the proof of the following theorem.

Theorem 3

If L = NL, *for each n-state* 2RNFA, *there exists an equivalent* 2DFA *with a number of states polynomial in n.*

In other words, under hypothesis L = NL, the class $2RN$ and $2D$ are equal. This result may be seen in its counterpart form: we can prove that L $\neq$ NL, by finding a language L in $2RN$ which is not in $2D$. Observe that, unlike to the relation with L = NL problem presented in [1], this relation does not depend on the size of the input word.

2.5 Other results

With analog proofs, we became able to prove polynomial reduction from 2RNFA to unambiguous 2NFA(2UFA). Compared with Theorem 3, 2UFAs are more powerful devices than 2DFAs, but this simulation does not require any additional assumptions, such as L= NL.

Reinhardt and Allender [22] proved that, in the context of nonuniform complexity, nondeterministic logarithmic space bounded computations can be made unambiguous. Our simulation combines this result with the reduction from a 2RNFA language to GAP.

Given a complexity class C , let us denote [16] by C/poly the class of languages $\mathcal{L}$ for which there exist a sequence of binary "advice" strings $\{\alpha(n)|n \geq 0\}$ of polynomial length and a language $\mathcal{B} \in C$ such that $\mathcal{L} = \{x|(x, \alpha(|x|)) \in \mathcal{B}\}$.

Theorem 4 (*see [22]*)

NL $\subseteq$ UL/poly

As a consequence of this theorem, there exists a nondeterministic Turing machine U_{GAP} such that:

- U_{GAP} works in logarithmic space and has at most one accepting path on each input string,
- there exists a sequence of binary strings $\{\alpha(n) \mid n \geq 0\}$ and a polynomial q, such that $|\alpha(n)| \leq q(n)$ for each $n \geq 0$, and
- for each graph G with N vertices, encoded in the form of the binary adjacency matrix, U_{GAP} accepts the string $G \sharp \alpha(N^2)$ if and only if $G \in$ GAP. Here $\sharp \notin \{0, 1\}$ denotes a new separator symbol.

We are now ready to prove this following theorem:

Theorem 5

Every n-state 2RNFA *can be simulated by a* 2UFA *which has a number of states polynomial in n.*

Proof: The idea of this proof is the same as the proof of Theorem 3. We will also simulate the given Turing machine (now we use U_{GAP} instead of D_{GAP}), by an automaton (now this automaton is just unambiguous and not necessary deterministic).

The unique difference with the previous construction is that U_{GAP} can use $\alpha(size(G(w))$ information. But, by Remark 1, this size of $G(w)$ is n^2, and so does not depend on the length of the input word w. Hence the value of $\alpha(n^2)$ is the same for all input word, it can be calculated before the construction of our automaton and encoded in its transition function. ■

In fact, the proof of Theorem 3 give us a general method to simulate a Turing machine. So using other known results about GAP or its variants, we can obtain by this method new theorem to simulate 2RNFA by automata with the some properties of the simulated Turing machine.

We also find a family of language which is complete for the class $2RN$, which is a restriction of complete language family of Sakoda & Sipser for $2N$.

Conclusion

This internship give me the motivation to research, by showing me all main aspects of researcher work:

- Readings: in the first part I discover the area by many readings, from generalities to details
- Presentation: then I presented these readings, using slides
- Research and proof: in a second time, I start to search (and find), alone or by exchange with G. Pighizzini
- Explanation and writing: in the third time, I also write (still a draft) of article, in aim of publication (and also this report)
- Attend conferences: I was allowed to assist at the NCMA workshop, and so discover the community of researchers and other subjects of research about automata.

It also give me the interest of descriptional complexity. I probably will continue to work on the article, by e-mail exchange.

This internship was also the occasion to discover Italian research, and meet (particularly at the NCMA workshop) others researchers. More generally, I discover another (close) culture, by learning Italian and by meeting Italians.

References

[1] J. Berman and A. Lingas. On the complexity of regular languages in terms of finite automata. Technical Report 304, Polish Academy of Sciences, 1977.

[2] Piotr Berman. A note on sweeping automata. In *ICALP*, pages 91–97, 1980.

[3] Pavol Duris, Juraj Hromkovic, José D. P. Rolim, and Georg Schnitger. Las vegas versus determinism for one-way communication complexity, finite automata, and polynomial-time computations. In Rüdiger Reischuk and Michel Morvan, editors, *STACS*, volume 1200 of *Lecture Notes in Computer Science*, pages 117–128. Springer, 1997.

[4] V. Geffert, C. Mereghetti, and G. Pighizzini. Converting two-way nondeterministic automata into simpler automata. *Theoret. Comput. Sci.*, 295:189–203, 2003.

[5] Viliam Geffert, Carlo Mereghetti, and Giovanni Pighizzini. Complementing two-way finite automata. *Inf. Comput.*, 205(8):1173–1187, 2007.

[6] Viliam Geffert and Giovanni Pighizzini. Two-way unary automata versus logarithmic space. *Inf. Comput.*, 209(7):1016–1025, 2011.

[7] Jonathan Goldstine, Martin Kappes, Chandra M.R̃. Kintala, Hing Leung, Andreas Malcher, and Detlef Wotschke. Descriptional complexity of machines with limited resources. *J. UCS*, 8(2):193–234, 2002.

[8] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, MA, 1979.

[9] Juraj Hromkovic and Georg Schnitger. Nondeterminism versus determinism for two-way finite automata: Generalizations of sipser's separation. In Jos C.M̃. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 439–451. Springer, 2003.

[10] Christos A. Kapoutsis. Removing bidirectionality from nondeterministic finite automata. In *MFCS*, pages 544–555, 2005.

[11] Christos A. Kapoutsis. Small sweeping 2nfas are not closed under complement. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (1)*, volume 4051 of *Lecture Notes in Computer Science*, pages 144–156. Springer, 2006.

[12] Christos A. Kapoutsis. Size complexity of two-way finite automata. In Volker Diekert and Dirk Nowotka, editors, *Developments in Language Theory*, volume 5583 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 2009.

[13] Christos A. Kapoutsis. Nondeterminism is essential in small 2fas with few reversals. In *ICALP (2)*, pages 198–209, 2011.

[14] Christos A. Kapoutsis. Two-way automata versus logarithmic space. In *CSR*, pages 359–372, 2011.

[15] Martin Kappes, Andreas Malcher, and Detlef Wotschke. Remembering chandra kintala. In *DCFS*, pages 15–26, 2010.

[16] R. Karp and R. Lipton. Turing machines that take advice. *Enseign. Math.*, 28:191–209, 1982.

[17] Hing Leung. Separating exponentially ambiguous finite automata from polynomially ambiguous finite automata. *SIAM J. Comput.*, 27(4):1073–1082, 1998.

[18] Hing Leung. Tight lower bounds on the size of sweeping automata. *J. Comput. Syst. Sci.*, 63(3):384–393, 2001.

[19] Albert R. Meyer and Michael J. Fischer. Economy of description by automata, grammars, and formal systems. In *FOCS*, pages 188–191. IEEE, 1971.

[20] Silvio Micali. Two-way deterministic finite automata are exponentially more succinct than sweeping automata. *Inf. Process. Lett.*, 12(2):103–105, 1981.

[21] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Develop.*, 3:114–125, 1959.

[22] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM J. Comput.*, 29(4):1118–1131, 2000.

[23] William J. Sakoda and Michael Sipser. Nondeterminism and the size of two way finite automata. In *STOC*, pages 275–286. ACM, 1978.

[24] Michael Sipser. Halting space-bounded computations. *Theor. Comput. Sci.*, 10:335–338, 1980.

[25] Michael Sipser. Lower bounds on the size of sweeping automata. *J. Comput. Syst. Sci.*, 21(2):195–202, 1980.