Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Theoretical Computer Science 302 (2003) 257–274

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the complexity of intersecting finite state automata and $\mathcal{NL}$ versus $\mathcal{NP}$☆

George Karakostas[a], Richard J. Lipton[b,c], Anastasios Viglas[d,*]

[a] *Department of Computing and Software, McMaster University, 1280 Main St. West, Hamilton, Ont., Canada L8S 4K1*

[b] *Georgia Institute of Technology, College of Computing, 801 Atlantic Avenue, Atlanta, GA 30332, USA*

[c] *Telcordia Applied Research, USA*

[d] *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ont., Canada M5S 3G4*

Received 12 April 2002; received in revised form 22 October 2002; accepted 31 October 2002
Communicated by J. Díaz

Abstract

We consider uniform and non-uniform assumptions for the hardness of an explicit problem from finite state automata theory. First we show that a small improvement in the known straightforward algorithm for this problem can be used to design faster algorithms for subset sum and factoring, and improved deterministic simulations for non-deterministic time.

On the other hand, we can use the same improved algorithm for our FSA problem to prove complexity class separation results ($\mathcal{NL}$ is not equal to $\mathcal{P}$, or $\mathcal{NP}$ for the non-uniform case). This result can be viewed either as a hardness result for the FSA intersection problem, or as a method for separating $\mathcal{NL}$ from $\mathcal{P}$ or $\mathcal{NP}$. It is interesting to note that this approach is based on a more general method for separating two complexity classes, using algorithms rather than lower bounds.

Keywords: Complexity class separations; $\mathcal{NL}$, $\mathcal{NP}$; Finite state automata intersection

☆ A preliminary version of this work (extended abstract) appeared in the Proceedings of the 15th Annual IEEE Conference on Computational Complexity, Florence, Italy, July 2000, pp. 229–234. This work was completed while all the authors were at Princeton University, Computer Science Department, Princeton, NJ, USA.

* Corresponding author.

E-mail addresses: karakos@mcmaster.ca (G. Karakostas), rjl@cc.gatech.edu (R.J. Lipton), aviglas@cs.toronto.edu (A. Viglas).

0304-3975/03/$ - see front matter
PII: S0304-3975(02)00830-7

1. Introduction

Proving complexity class separations is a major problem in Complexity Theory which is directly connected to proving hardness results. Strong enough hardness results will imply class separations in a straightforward way. On the other hand, a strong enough "easiness" result will also imply class separations in a more indirect way, as we are going to discuss in this work. In particular, we show a connection between the separation of $\mathcal{NL}$ from other complexity classes and the hardness of an explicit problem in $\mathcal{P}$. We consider the problem of deciding whether the intersection of a collection of k deterministic finite state automata is empty. Either this problem requires large circuits or $\mathcal{NL} \neq \mathcal{NP}$. For the uniform case, either this problem does not have fast algorithms or $\mathcal{NL} \neq \mathcal{P}$. In both cases, "easiness" of an explicit problem implies a class separation. On the other hand, we also prove that if the finite state automata intersection emptiness problem has indeed fast algorithms, that is, if there exists almost any improvement to the known algorithm, then we can design faster algorithms for subset sum and integer factoring. In addition to that, we can also use these fast algorithms for the intersection emptiness problem to provide improved deterministic simulations for non-deterministic time.

Let $F_1, F_2, \ldots, F_k$ be a collection of k finite state automata of size[1] $|F_i| = \sigma$ and consider the problem of checking whether their intersection is empty:

$$\bigcap_{i=1}^{k} L(F_i) \stackrel{?}{\neq} \emptyset$$

where $L(F)$ denotes the language accepted by the automaton F.

The standard algorithm for checking the above intersection involves constructing the finite state automaton corresponding to the "Cartesian product" $F = F_1 \times F_2 \times \cdots \times F_k$, and solving the emptiness problem for $F : L(F) \neq \emptyset$. The size of the intersection automaton F is $O(\sigma^k)$.

Let $\mathcal{F}$ denote the assumption that there is a better algorithm for checking the intersection emptiness problem for a collection of a fixed number of automata: Let $F_1, F_2, \ldots, F_k$ be k FSAs of size σ. $\mathcal{F}$ denotes the assumption that there is a deterministic algorithm that can decide whether

$$\bigcap_{i=1}^{k} L(F_i) \stackrel{?}{\neq} \emptyset$$

in time $\sigma^{(k/f(k))+d}$, where $f(\cdot)$ is an unbounded function that depends only on k, and $d > 0$ is a constant.

Based on the assumption $\mathcal{F}$ we can prove the following theorems:

(1) There is an algorithm solving sub-set sum in $2^{\varepsilon n}$ for any $\varepsilon > 0$.
(2) Integer factorization can also be solved in $2^{\varepsilon n}$ for any $\varepsilon > 0$.
(3) $\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(2^{\varepsilon t})$ for any $\varepsilon > 0$.

[1] For simplicity, in this paper the size of an automaton is the number of states. The number of bits required for the description of the automaton is the same times a poly-logarithmic factor, which does not affect our computations.

A slight modification of assumption $\mathcal{F}$ also allows us to separate $\mathcal{NL}$ from $\mathcal{P}$. These results for the uniform case use the notion of block respecting computation and apply to the multi-tape Turing Machine model.

If we consider a non-uniform version of our assumption, i.e. that there exists a "small" circuit that solves the emptiness problem for a collection of FSAs then we can prove that $\mathcal{NL} \neq \mathcal{NP}$. This result can be proved using a new lemma, that provides a general technique for proving complexity class separations and may be of independent interest.

It is interesting to note that the complexity class separation results mentioned above, are based on algorithms rather than lower bounds. To complete the separation these algorithmic techniques will be combined with known hierarchy results or counting arguments, which means that a diagonalization or a counting argument still lies in the heart of the separation. In order to separate $\mathcal{NL}$ from $\mathcal{P}$ for example, all we would need is to improve the algorithms for deciding whether the intersection of a collection of finite state automata is empty.

Note that for the intersection emptiness problem, the parameter k, the number of the finite state automata, is constant. The general problem, where this parameter can depend on the input size is much harder, known to be $\mathcal{PSPACE}$-complete [9]. If k is a constant then the problem has a polynomial time algorithm as described above. It is also known that the emptiness problem for a finite state automaton is $\mathcal{NL}$-complete [7]. The problem we consider is a parametrized version of a finite state automata intersection problem and also captures all $\mathcal{NL}$ computations.

A similar result was given by Feige and Kilian [4]. In that work the clique problem is considered and more specifically the following parameterized version: given a graph on n nodes, does it contain a clique of size k where $k < \log n$? The general clique problem is $\mathcal{NP}$-complete. But the complexity of the parameterized version mentioned above, recognizing small-$\log n$ size cliques, remains an open problem. Feige and Kilian [4] prove that if this problem is solvable in polynomial time then there is a "fast" simulation of non-deterministic computations:

$$\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(\tau^{\sqrt{\tau}}) \tag{1}$$

where $\tau = t \log t$, and $t > n$. That work is inspired from the fact that certain $\mathcal{NP}$-complete problems require only "limited" non-determinism and questions that come from the framework of *fixed parameter intractability* [2,3]. For example the $\mathcal{NP}$-complete problem Vertex Cover ("given a graph with n vertices, is there a vertex cover of size k?") is known to have algorithms with running time of the form $O(2^k \cdot n^c)$ for some fixed constant c, which implies a polynomial time algorithm for small values of the parameter $k < \log n$.

Another related result is that of Paul et al. [10]. The main result was that non-deterministic linear time is more powerful than deterministic linear time $\mathcal{NTIME}(n) \neq \mathcal{DTIME}(n)$. This is related to the non-deterministic time simulation result presented in this work.

2. Notation and definitions

We use the standard notation for the usual complexity classes and Turing Machine time and space bounds: $\mathcal{DTIME}(t)$ and $\mathcal{NTIME}(t)$ denote the classes of languages accepted by deterministic and non-deterministic multi-tape Turing Machines respectively, running in time $O(t)$. We will use both $n^{O(1)}$ and $poly(n)$ to denote functions that are polynomial in n, while $\exp(x)$ denotes the exponential 2^x. For a machine M, $L(M)$ denotes the language accepted by M. In many cases we will just use M to denote both the machine and the language accepted by the machine, to simplify the notation; for example if F_1 and F_2 are two finite state automata then $F_1 \cap F_2$ denotes the language accepted by both automata. The size of a finite state automaton for our purposes is the number of states of the automaton. The number of bits required to describe the automaton is the same times a logarithmic factor, which in any case does not affect our computations. The finite state automata considered in this work are deterministic unless stated otherwise. The "cartesian product" of two (or more) automata, denoted $F_1 \times F_2$, is the automaton whose set of states is the cartesian product of the state sets of the two automata and accepts all strings accepted by both automata F_1, F_2. This corresponds to the usual construction of the automaton that accepts the intersection of the languages accepted by the two given automata.

The notion of *block respecting* computation was introduced by Hopcroft Paul and Valiant in [6] to prove that deterministic space is strictly more powerful than deterministic time: $\mathcal{DTIME}(t) \subseteq \mathcal{DSPACE}(t/\log t)$. Block respecting Turing machines are also used in [10] to prove that non-deterministic linear time is more powerful than deterministic linear time. See also [11] for a generalization of the results from [6] for RAMs and other machine models (Fig. 1).

Definition 2.1. Let M be a machine running in time $t(n)$, where n is the length of its input x. Let the computation of M be partitioned in $a(n)$ *segments*, where each segment

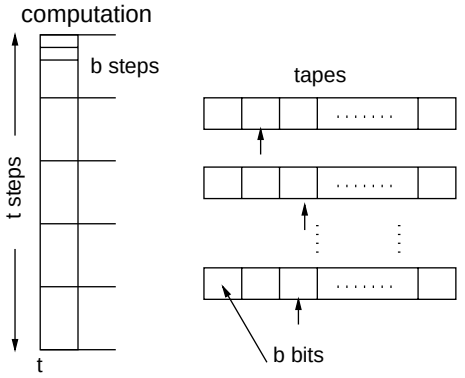

Fig. 1. Block respecting computation.

consists of $B(n)$ consecutive steps and $a(n)\cdot B(n)=t(n)$. Let also the cells of the tapes of M be partitioned into $a(n)$ *blocks* each consisting of $B(n)$ cells on each tape. We will call M *block respecting* if during each segment of its computation, each head visits only one block on each tape.

Every Turing Machine can be converted to a block respecting machine with only a constant factor slow down in its running time. The construction is simple: Let M be a deterministic Turing Machine running in time t. Break the computation steps $(1\dots t)$ in segments of size B. Break the work tapes in blocks of the same size B. If at the start of a computation segment σ the work tape head is in block b_j, then during the B computation steps of that segment, the head could only visit the adjacent blocks, b_{j-1} or b_{j+1}. Keep a copy of those two blocks along with b_j and do all the computation of segment σ reading and updating from those copies, if needed. At the end of the computation of every segment, there is a clean-up step: update the blocks b_{j-1} and b_{j+1} and move the work tape head to the appropriate block to start the computation of the next segment. This construction can be done for different block sizes B. For our purposes B will be t^c for a small constant $c<1$. For more details on this construction see [6].

The idea of block respecting computation will be used for simulating Turing machines in our proofs. The general idea is the following: in order to carry out a simulation of a Turing machine, break the computation in blocks, such that in each block the computation is "local". Then check the correctness of the computation in each block independently. The notion of block respecting computation applies to multi-tape Turing machines and therefore our results do not seem to generalize directly for random access machines.

As we mentioned in the introduction, $\mathcal{F}$ will denote the assumption that there is a slight improvement to the standard algorithm for checking the intersection emptiness problem for a collection of a fixed number k of automata:

Assumption 1 ($\mathcal{F}$). Let $F_1,F_2,\dots,F_k$ be k FSAs of size σ. There is a deterministic algorithm that can decide whether

$$\bigcap_{i=1}^{k} L(F_i) \stackrel{?}{\neq} \emptyset$$

in time $\sigma^{(k/f(k))+d}$, where $f(\cdot)$ is an unbounded function that depends only on k, and $d>0$ is a constant.

3. Subset sum and factoring

We start by showing the implications of assumption $\mathcal{F}$ for two problems that are considered hard: subset sum and factoring. If $\mathcal{F}$ is true then we can construct better algorithms for solving these problems.

3.1. Subset sum

We consider the following type of subset sum problem: Given m integers $a_1,\ldots,a_m$ and a number b, check if there exists a boolean vector $x=(x_1,\ldots,x_m)$ such that:

$$\sum_{i=1}^{m} a_i x_i = b.$$

Denote by n the size of the input for this problem: $a_1,\ldots,a_m,b$.

Assuming that there is an "easy" way of checking the intersection of two automata is empty, we can construct an algorithm solving subset sum in $2^{n/2}n^{O(1)}$. By choosing a collection of k automata, the resulting algorithm runs in $2^{\varepsilon n}$ for any $\varepsilon>0$.

Theorem 3.1. *Assumption $\mathcal{F}$ implies that there is an algorithm solving subset sum in $O(2^{\varepsilon\cdot n})$ for all $\varepsilon>0$.*

Proof. Pick two primes p,q of size $n/2$ (size just greater than $n/2$) and build two machines M_p and M_q testing $\sum_{i=1}^{m} a_i x_i \equiv b \pmod p$ and $\sum_{i=1}^{m} a_i x_i \equiv b \pmod q$. The construction of these automata is simple. The input to the automata is the bit vector $x=x_1x_2\ldots x_m$. The numbers $a_1,\ldots,a_m$ are part of the given subset sum problem, and are used in the construction of the automata since the transitions of each automaton depend on these a_i's. The machine M_p, reads the bits $x_1,x_2,\ldots,x_m$ and needs to find the value of the sum $\sum_{i=1}^{m} a_i x_i$ modulo p. During this computation, M_p only needs to remember the value of the partial sum modulo p, not the exact value. The automaton will have width at most p since there are p values modulo p, and length m: that is m stages of p states. At stage j, the automaton is in a state that shows the value of the partial sum $\mu=\sum_{i=1}^{j-1} a_i x_i \pmod p$, reads x_j and goes to a state in the next stage that corresponds to the value of $\mu + a_j x_j \pmod p$ and so on. The size (number of states) of these two machines is $|M_p| = O(p\cdot n^{O(1)})$, where $p=O(2^{n/2})$, and M_q has the same size.

Consider the following intersection problem:

$$M_p \cap M_q \overset{?}{\neq} \emptyset. \tag{2}$$

If there is a solution to the given subset sum problem then the intersection $M_p\cap M_q$ is non-empty, since $\sum_{i=1}^{m} a_i x_i \equiv b \pmod p$ modulo any number p. If, on the other hand, $\sum_{i=1}^{m} a_i x_i \equiv b$ modulo both primes, then $b-\sum_{i=1}^{m} a_i x_i$ is a multiple of p and q, $\sum_{i=1}^{m} a_i x_i \equiv b \pmod{p\cdot q}$. But the primes p,q where chosen so that $pq>b$ and therefore, in this case, $\sum_{i=1}^{m} a_i x_i = b$ in general.

In other words, the question "does there exist a solution x to the given subset sum problem", translates to "does there exist an input accepted by both automata". If (and only if) the intersection (2) is non-empty, then there exists a solution to the given subset sum problem.

In order to get the desired bound, $2^{\varepsilon n}$ for any positive ε, use a similar construction for k automata: pick k primes $p_1,\ldots,p_k$ of size n/k and follow the same ideas described

above to construct k automata M_{p_k} that check if $b - \Sigma_{i=1}^{m} \equiv 0 \pmod{p_k}$. Consider the emptiness problem for the intersection:

$$\bigcap_{i=1}^{k} M_{p_i}. \tag{3}$$

The size of the automata M_{p_k} is at most $\sigma = O(2^{n/k} \cdot n^{O(1)})$. Now we can use the assumption $\mathcal{F}$: for a collection of k automata of size σ the emptiness problem of their intersection can be solved in time $O(\sigma^{(k/f(k))+d})$. This will give us the following upper bound:

$$\exp\left(\frac{1}{f(k)}n + \frac{d}{k}n\right). \tag{4}$$

The time needed to construct the FSAs described above is $2^{n/k} \cdot n^{O(1)}$. For large enough k the total time required becomes

$$2^{(1/f(k))n} \cdot poly(n), \tag{5}$$

assuming that $f(k)$ grows slower than $O(k)$. Since $f(k)$ is unbounded, $1/f(k)$ can become less than any constant $\varepsilon > 0$ by choosing k appropriately. □

3.2. Integer factoring

Using the same ideas as in the previous section, we can prove that integer factoring of an n-bit number is solvable in $n^{O(1)} \cdot 2^{\varepsilon n}$, for any $\varepsilon > 0$, provided that the assumption $\mathcal{F}$ is valid. Finding deterministic algorithms for factoring is a major open problem: The best known deterministic algorithm runs in time $2^{(1/4)n}$. With the Extended Riemann Hypothesis this bound only improves to $2^{(1/5)n}$ (see [1]).

The problem is the following: Given any integer z of size n, find x, y such that $x \cdot y = z$.

Theorem 3.2. *The assumption $\mathcal{F}$ implies that factoring can be solved in time $2^{\varepsilon n}$ for any $\varepsilon > 0$.*

Proof. We show how to build a fixed number of finite state automata to check if $x \cdot y = z$. Exactly as in the case of the subset sum problem, pick two primes p and q of size $n/2$ and consider the corresponding FSAs M_p, M_q, checking whether $x \cdot y \equiv z \pmod{p}$ and $x \cdot y \equiv z \pmod{q}$, respectively. The following emptiness problem solves the factoring problem:

$$M_p \cap M_q \overset{?}{\neq} \emptyset. \tag{6}$$

The input of the finite state machines is the string xy. Since M_p and M_q are finite state automata, we need to know the length of x and y in advance; we need to know where the string x stops and y starts. Recall that only z is given to us as input, and our algorithm will try to determine whether there exist any x, y such that $x \cdot y = z$.

Since the length of the possible factors x, y is not known in advance, we simply check all possible lengths $|x| = n/2,\ |x| = n/2 - 1, \ldots$. The size of M_p is $p \cdot n^{O(1)}$ and for $M_q\, q \cdot n^{O(1)}$. Based on the assumption $\mathcal{F}$, we can check the intersection (6) for emptiness in $2^{n/2} \cdot n^{O(1)}$.

Now consider k primes $p_1, \ldots, p_k$ of size n/k each, build the corresponding collection of FSAs $M_{p_1}, \ldots, M_{p_k}$. The size of each automaton is $|M_{p_i}| \equiv \sigma = p_i \cdot n^{O(1)} = 2^{n/k} \cdot n^{O(1)}$. The factoring problem can be solved by checking the following intersection:

$$\bigcap_{i=1}^{k} M_{p_i} \stackrel{?}{\neq} \emptyset. \tag{7}$$

By our assumption $\mathcal{F}$ the intersection from Eq. (7) can be solved in time $\sigma^{(k/f(k))+d}$. This yields the following upper bound:

$$n^{O(1)} \exp\left(\frac{1}{f(k)} n + \frac{d}{k} n\right). \tag{8}$$

In order to factor a given number z proceed as follows: check whether there exist x, y such that $xy = z$, trying all possible lengths for x and using the automata intersection technique presented above. In order to find the actual number x, compute its bits one by one by solving the following problem: is there a factorization of $z = xy$ where the first bit of x is 1? If we repeat this $n/2$ times, we can find all the bits of the number x. □

4. Deterministic simulation of non-deterministic computation

In this section we show how to build a collection of automata to simulate deterministically the computation of a non-deterministic time-bounded Turing machine. Under the assumption $\mathcal{F}$ the time required for constructing the automata and checking their intersection for emptiness will give an improvement in the required time for the simulation. These results apply for the multi-tape Turing machine model.

A *trace of the computation* on input x of a machine M is a string of computation steps. Each step contains the current contents of the working tapes, the position of the heads, the state of M, the input symbol read, the position of the head on the input tape, and the non-deterministic choice of M at this step. This description of the computation of a Turing machine M on a certain input is also referred to as a *tableau* of computation, the computation string, or just "the computation" of M.

The main idea for the simulation follows roughly these steps: Start from any non-deterministic Turing machine computation and on input x simulate the computation deterministically as follows:

(1) Break the non-deterministic computation in blocks.
(2) Make sure the computation is "local" in each block.
(3) Build finite state automata that will check the correctness of the computation in each block. Each block can be checked independently.
(4) Check if all the automata accept the computation, which would mean that every block of the computation is correct.

If the input x is accepted by the non-deterministic machine M we started from, then there exists a valid accepting computation of M on x. Therefore, for our deterministic simulation described above, the question "does there exist an accepting computation for x" translates to "is there a string accepted by all the automata" in the last step. To answer this question, we will use the fast algorithm for FSA intersection emptiness whose existence is implied by assumption $\mathcal{F}$, to speed up the simulation.

Each automaton will be checking a part of the computation. In order to keep the size of the automata small, we would like to break the computation in such a way so that each automaton will only have to look in specific and as small as possible parts of the computation to verify correctness. This can be achieved if we make the computation "local" in the sense of the "block respecting computation" ideas.

For any Turing machine M running in time t, we can break the computation in t/B segments with B steps per segment, and also each work tape in blocks of the same size, B bits per block. Now the machine M can be easily converted to be block respecting, meaning that during each segment of computation only the contents of at most one block per tape are used/accessed, as mentioned in more detail in the introduction.

Let M be a non-deterministic Turing machine running in time t. On input x, our simulation proceeds as follows: Convert M to a block respecting machine M_B. Consider the trace of the computation of M_B and construct a collection of FSAs that will check if the computation is correct the following way: each FSA will check the correctness for a number of segments of the computation trace. Note that since M_B is block respecting, for each segment of the computation, the automaton needs to check the contents of only one block on each working tape. Now consider the intersection of all the automata and recall that an automaton accepts its input if it corresponds to a valid computation of M_B on x. If the intersection of the automata is non-empty, then there exists a valid accepting computation for M_B and therefore for M.

In order for an FSA to check a computation segment, it needs to know the contents of the corresponding blocks the last time they were accessed in the computation trace. Since the machine M_B is non-deterministic, we need to consider many possibilities for the position of these previous accesses in the computation trace for M_B. These dependencies can be represented by a graph as in [6]. For the deterministic simulation we need to consider all possible graphs (Fig. 2).

Theorem 4.1. *Assumption $\mathcal{F}$ implies $\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(2^{\varepsilon t})$, for any $\varepsilon > 0$.*

Proof. Let M be a non-deterministic machine with l tapes, running in time t. Let M_B be the corresponding block respecting machine, with running time $O(t)$. Consider the computation of M_B on input x. Break that computation in segments of size B each; the number of segments is $O(t/B)$. Consider the directed graph G corresponding to the computation of the block respecting machine as described in [6]: G has one vertex for every time segment (that is t/B vertices) and the edges are defined from the sequence of head positions. Let $v(\Delta)$ denote the vertex corresponding to time segment Δ and Δ_i is the last time segment before Δ during which the ith head was scanning the same block as during segment Δ. Then the edges of G are $v(\Delta - 1) \to v(\Delta)$ and for all $1 \leqslant i \leqslant l$, $v(\Delta_i) \to v(\Delta)$. The number of edges can be at most $O((l+1)t/B)$ and

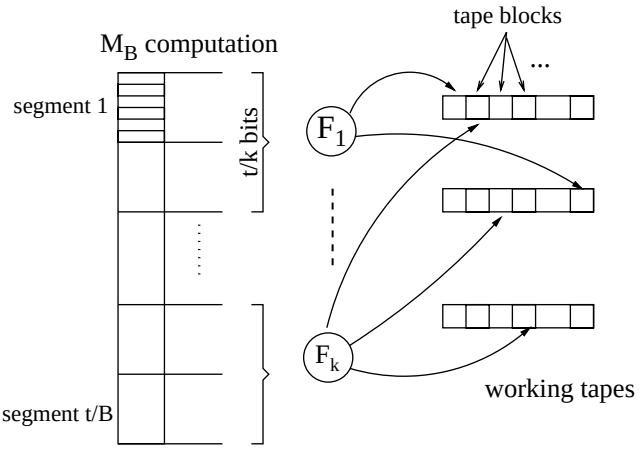

Fig. 2. Checking block respecting computation with finite state automata.

therefore the number of bits required to describe the graph is $O((l+1)t/B \log t/B)$. Since the number of tapes l is a constant, from now on it will be incorporated in the big-O notation.

The general idea for simulating M_B, is to build finite state automata to check the computation that takes place on each vertex of the graph (each vertex corresponds to a segment of the computation). Since the machine M is non-deterministic, we need to consider all $2^{O(t/B \log t/B)}$ possible such graphs.

Now consider the computation of the block respecting machine during time segment Δ: this time segment contains B computation steps. In each step, the machine reads and writes the bits on the head positions in the blocks corresponding to Δ depending on the non-deterministic choice at that step (Fig. 3).

In order to check if the computation is correct during one step, we could use an FSA of constant size which actually depends only on the number of tapes of the machine. This check can be done by a decision tree of size $2^{O(B)}$.

Let k be the number of FSAs. Then for any $\varepsilon > 0$ we can pick $k = 1/\varepsilon$ such that each automaton checks $(1/k)t/B = \varepsilon t/B$ segments or $(\varepsilon t/B)B = \varepsilon t$ steps. The size of each FSA is therefore $2^{\varepsilon t}$ (decision tree size).

Our deterministic algorithm that will simulate M must construct the transition diagrams for these k FSAs. For each transition (arc in the decision tree), we need to simulate M_B for at most 2^B steps. Since $2^{\varepsilon t}$ is the total number of transitions, the total time required is $2^{\varepsilon t} \cdot 2^B$. The running time for the construction of all the FSAs for all possible graphs is therefore:

$$2^{\varepsilon t} 2^B 2^{O((t/B)\log(t/B))} = 2^{O(\varepsilon t)}. \tag{9}$$

In order to check if there exists an accepting computation for M on input x it suffices to check if the intersection of all FSAs $\bigcap_{i=1}^{k} F_i$ is non-empty. Under our assumption

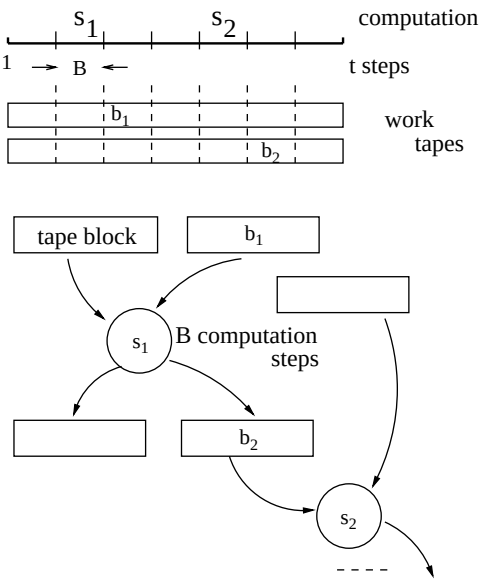

Fig. 3. Graph description of a block-respecting computation.

$\mathcal{F}$, the time needed to intersect k FSAs of size $\sigma = 2^{\varepsilon t} = 2^{t/k}$ is

$$\begin{aligned}\sigma^{(k/f(k))+d} &= (2^{t/k})^{(k/f(k))+d} \\ &= 2^{(1/f(k))t+(d/k)t}. \end{aligned} \tag{10}$$

Since $f(k) = o(k)$, the time needed for testing the intersection for emptiness is $2^{O((1/f(k))t)}$. Therefore the total time for our simulation is the time to construct the FSAs plus the time to check the intersection: $2^{O(\varepsilon t)} + 2^{O((1/f(k))t)}$. Since $f(k)$ is unbounded we can always pick k appropriately so that $f(k) < 1/\varepsilon$ and the total time is $2^{O(\varepsilon t)}$. □

5. Separating complexity classes

Consider the problem of separating two complexity classes, for example, $\mathcal{P}$ from $\mathcal{NP}$. One way to approach this problem is to show that $\mathcal{NP}$ is "too hard", meaning that there exists a problem in $\mathcal{NP}$ that cannot be solved in deterministic polynomial time. A different way to view this separation problem, is to prove that $\mathcal{P}$ is actually "too easy", in the sense that everything in $\mathcal{P}$ can be solved in small non-deterministic time. For example, if we can prove that every problem in $\mathcal{P}$ has fixed polynomial size non-uniform circuits (for example size n^5) then $\mathcal{P} \neq \mathcal{NP}$. This approach tries to prove separation results using upper bounds and algorithmic techniques rather than lower bounds. In order to complete the separation we still need to apply a known diagonalization, hierarchy theorem or counting result. In the following section we will

see an example of this method: If there exists a fast enough algorithm solving the FSA intersection emptiness problem, then $\mathcal{NL}$ is "too easy" for polynomial time $\mathcal{P}$, meaning that everything in $\mathcal{NL}$ can be done in fixed polynomial (less than n^2) time. The separation follows immediately from the well known time hierarchy results. This means that if one would like to separate $\mathcal{NL}$ from $\mathcal{P}$ it would suffice to improve the algorithm for the FSA intersection emptiness problem. On the other hand, this could be considered as a hardness result for the FSA problem. Since separating these fundamental complexity classes is considered quite hard, this theorem could be an indication of the hardness of the FSA problem.

A similar, more general result can be shown for a non-uniform variant of our assumption. If we assume that there is a small enough non-uniform circuit solving the FSA intersection emptiness problem, then we can separate $\mathcal{NL}$ from $\mathcal{NP}$. This is based on a result of Kannan [8].

In the previous sections, the assumption $\mathcal{F}$ that was used, was that given k FSAs $F_1, F_2, \ldots, F_k$ of the same size σ, there is an algorithm that can check whether their intersection is empty in time $\sigma^{(k/f(k))+d}$. We modify slightly this assumption to the following:

Assumption 2 ($\mathcal{F}'$). Let $F_1, F_2, \ldots, F_k$ be k FSAs of size σ and G a FSA of size σ'. Then there is a deterministic algorithm that can decide whether

$$\bigcap_{i=1}^{k} F_i \cap G \overset{?}{\neq} \emptyset$$

in time $\sigma^{(k/f(k))+d}\sigma'$, where $f(\cdot)$ is an unbounded function and $d>0$ is a constant.

Notice that the new assumption $\mathcal{F}'$ differs from $\mathcal{F}$ only in the introduction of an extra FSA G which may not have the same size as the rest of the FSAs. The problem can still be solved by the standard method of taking the Cartesian product of the $k+1$ automata and deciding whether its language is empty in time $O(\sigma^k\sigma')$. This is a natural generalization, stating basically the same fact as the original assumption $\mathcal{F}$ ("is there a faster algorithm for FSA intersection emptiness?") and is used to overcome a technical point in our proof.

We also prove a similar separation result for the non-uniform setting. For this case, we will state a more general assumption, which is just the non-uniform version of $\mathcal{F}'$: instead of assuming that there exists a fast enough algorithm solving the finite state automata intersection emptiness problem, assume that there is a *non-uniform* circuit that will solve the same problem in small size. Call this assumption $\mathcal{F}_C$.

Assumption 3 ($\mathcal{F}_C$). Let $F_1, F_2, \ldots, F_k$ be k FSAs of size σ and G a FSA of size σ'. Then there is a circuit (family of non-uniform circuits) that can decide whether

$$\bigcap_{i=1}^{k} F_i \cap G \overset{?}{\neq} \emptyset$$

with size $\sigma^{(k/f(k))+d}\sigma'$, where $f(\cdot)$ is an unbounded function, and $d>0$ is a constant.

In both the uniform and the non-uniform cases, the proofs of the separation theorems will proceed as follows. Think of the separation of $\mathcal{NL}$ from $\mathcal{P}$:

(1) Start from any $\mathcal{NL}$ Turing machine M_L and consider the computation on some input x. We will check if there exists any accepting computation on input x.
(2) "Break" the computation of the machine into blocks. In fact we will break the work tape into blocks, and each such block will correspond to a part of the computation.
(3) Build one finite state automaton for each block to check if a given computation string is correct, in all parts that correspond to that block, ignoring the rest of the computation string.
(4) The question "does there exist an accepting computation of M_L on x" translates to "does there exist a string accepted by all the automata"
(5) Argue that the entire simulation can be done in some fixed polynomial time or circuit size.

In order to speed up the simulation described above, we will use the fast algorithm or small circuit for the intersection emptiness problem for the last two steps.

For the uniform setting, this simulation will give an almost linear time simulation of $\mathcal{NL}$. Then the deterministic time-hierarchy results will complete the separation proof of $\mathcal{NL}$ and $\mathcal{P}$.

Theorem 5.1 (Deterministic time hierarchy). *For any* $k>0$, $\mathcal{DTIME}(n^k) \subset \mathcal{DTIME}(n^{k+1})$.

For the non-uniform case, the simulation will give non-uniform circuits of size less than n^2. To separate $\mathcal{NL}$ from $\mathcal{NP}$, the following result by Kannan [8] will be used:

Theorem 5.2 (Kannan [8]). *For any* $k>0$, *there is a language in* $\Sigma_2^p \cap \Pi_2^p$ *that does not have circuits of size* n^k.

In simple terms, for any k, the polynomial hierarchy contains hard problems, that require circuits bigger that n^k. This theorem can be applied to prove a general lemma (see Section 7) that can be used to separate complexity classes by designing fast algorithms and/or simulations rather than proving lower bounds.

5.1. Uniform assumption: $\mathcal{NL}$ vs. $\mathcal{P}$

In the uniform setting, we separate $\mathcal{NL}$ from $\mathcal{P}$ based on assumption $\mathcal{F}'$, by showing that $\mathcal{NL}$ is very easy given the power of polynomial time: every non-deterministic log-space computation can be done in less than n^2 time.

Theorem 5.3. *Assumption* $\mathcal{F}'$ *implies* $\mathcal{NL} \subseteq \mathcal{DTIME}(n^{1+\varepsilon})$, *for any* $\varepsilon>0$.

Proof. Without loss of generality, we can assume that an $\mathcal{NL}$ machine has only one working tape.

The main idea is the following: break the working tape of the machine into blocks. This corresponds into breaking the computation of the machine into segments. We will use one FSA for each tape block that will accept only strings representing 'correct' computations for this particular block. This is done by having the automaton going through its input (claimed to be a valid computation) until the head of the working tape enters the tape block assigned to this automaton. Then the FSA starts simulating the computation steps of the machine in this block, and checks whether the input represents a valid computation. The FSA continues to check all computation steps in the input until the work tape head leaves its pre-assigned block. Then the automaton goes through the rest of the computation, ignoring everything, until the head enters that block again or the computation ends. Note that the automaton 'remembers' the contents in its tape block in its own state, in order to do the simulation the next time it encounters its block in its input.

If there is a string that belongs in the languages accepted by all the FSAs, that is, if the intersection of their languages is non-empty, then this string corresponds to a computation that is correct for each block. There is a technical problem however: the FSAs cannot check whether the input on the read-only input tape, appears correctly throughout the computation string would require bigger finite state automata. To overcome this difficulty we will use another FSA that will only check the input of the computation on the computation string. This requires the modification of the assumption $\mathcal{F}$ as discussed above.

More specifically, let $L \in \mathcal{NL}$ and M_L be the corresponding block-respecting Turing machine, using at most $c \log n$ working space and therefore time n^c, for some constant $c > 0$. The computation on input x of this machine can be described by a string of computation steps: each step contains information about the position of the head of the working tape, the state of $M_L(x)$, the input symbol read, the non-deterministic choice of $M_L(x)$ at this step and the symbol read/written on the working tape (Fig. 4).

We break the working tape of $M_L(x)$ into k blocks of size B each (k is a parameter to be determined later). Then $k = c \log n/B$. For each block B_i, $i = 1 \ldots k$ we construct a FSA F_i that does the following:

(1) F_i reads its input until the working tape head enters B_i.
(2) Simulate the computation in B_i until head moves to B_{i-1} or B_{i+1}.
(3) Go through the rest of the computation string. If the working tape head enters B_i again, repeat the previous step.
(4) When the end of the computation is reached and the computation string read was correct, then accept/reject according to what $M_L(x)$ does.
(5) If any errors in were discovered in the computation string, reject.

In order to perform the second step, F_i has to keep in its state the contents of B_i and the current position of the working head, therefore it needs to remember $O(c2^B \log \log n)$ bits, and since we are going to pick B large enough, the size of F_i is $|F_i| = 2^{O(B)}$. The FSAs are constructed in a straightforward way, as decision trees, branching on every input bit from the pre-assigned positions on the tape blocks. In order to compute the

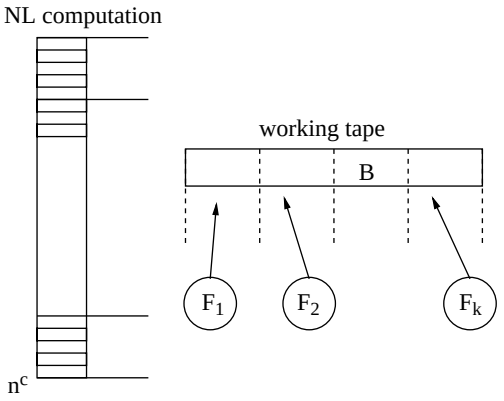

Fig. 4. Checking non-deterministic Logspace computation.

transitions of F_i and label the transitions in the automaton for a single computation step, we need to run $M_L(x)$ starting from all possible configurations of B_i. The number of transitions is at most O(number of states of $F_i) = 2^{O(B)}$. Hence the time needed to construct the FSAs is at most $2^{O(B)}$.

We still need to check whether the input (of M_L) is read correctly, if the input bits appear correctly in the computation string. We cannot assign this task to the F_i's since this requires too many bits to keep track of. Thus we construct one more FSA G with O(n) states that goes through the computation string and just checks the positions where the input bits are read by the $\mathcal{NL}$ machine M_L.

If the intersection $\bigcap_{i=1}^{k} L(F_i) \cap G$ is non-empty, then there is a computation string that represents a correct accepting computation of $M_L(x)$ (as checked by the F_i's), in which the input tape bits appear correctly (as checked by G). Using our assumption $\mathcal{F}'$, the emptiness of this intersection can be decided in deterministic time

$$\begin{aligned} |F_i|^{(k/f(k))+d}|G| &= d_1 2^{d_2((kB/f(k))+B)} n \\ &= d_1 2^{\frac{d_2 c \log n}{f(k)} + \frac{d_2 c \log n}{k}} n \end{aligned} \tag{11}$$

for some constants $d_1, d_2 > 0$. Considering the time needed to construct the FSAs, and for $f(k) = o(k)$, the total time needed for the deterministic simulation is at most

$$n^{(d_3 c/f(k))+1}$$

for some constant $d_3 > 0$, and thus we can always pick a big enough k so that $d_3 c/f(k) < \varepsilon$, for any $\varepsilon > 0$, since $f(\cdot)$ is unbounded. □

From the well known Time Hierarchy Theorem 5.1 we get the following:

Corollary 5.4. *Assumption $\mathcal{F}'$ implies $\mathcal{NL} \neq \mathcal{P}$.*

6. Non-uniform assumption: $\mathcal{NL}$ vs. $\mathcal{NP}$

The non-uniform version of our assumption $\mathcal{F}_C$ implies that $\mathcal{NL} \neq \mathcal{NP}$. We start by showing that $\mathcal{NL}$ has small, fixed polynomial size circuits, or in other words that $\mathcal{NL}$ is "too easy".

The following theorem proves that $\mathcal{NL}$ has size n^2 circuits, but note that any fixed polynomial size circuit simulation would work just as well. This will be obvious in the proof, where Kannan's [8] result is used.

Theorem 6.1. *Assumption $\mathcal{F}_C$ implies that $\mathcal{NL}$ can be simulated by fixed polynomial size circuits.*

Proof. The proof is essentially the same as for Theorem 5.3. Each of the automata F_i, $i = 1 \ldots k$, is of size $2^{O(B)}$ and thus can be described by a circuit of size $2^{O(B)}$. G can be described by a circuit of size $O(n^2)$. Assuming $\mathcal{F}_C$, there is a size $|F_i|^{(k/f(k))+d}|G| = n^{O(c/f(k)+c/k)}$ circuit that given the description of automata F_i, $i = 1 \ldots k$, and G from Theorem 5.3 decides the emptiness of their intersection. By picking k large enough this size can be made less than n^2 (any constant in the exponent would be sufficient here, as long as it is independent of c). Hence every language in $\mathcal{NL}$ has a circuit of size n^2. □

Corollary 6.2. *Assumption $\mathcal{F}_C$ implies $\mathcal{NL} \neq \mathcal{NP}$.*

Proof. (1) $\mathcal{NL}$ has fixed polynomial size circuits.

(2) If $\mathcal{NL} = \mathcal{NP}$ then the polynomial time hierarchy collapses to $\mathcal{NL}$, and Kannan's Theorem 5.2 implies that for any constant β there is a language in Σ_2^p and therefore in $\mathcal{NL}$ that is not computable by circuits of size n^β.

(3) By Theorem 6.1 $\mathcal{NL}$ has fixed polynomial (n^2) size circuits. This is a contradiction.

Therefore $\mathcal{NL} \neq \mathcal{NP}$. □

Note that the proof of Corollary 6.2 can be considered as an application of Lemma 6.3, presented in the next section.

6.1. A general lemma

The following lemma is a general way to state Kannan's result [8], as a tool for separating Complexity classes. As mentioned earlier, this technique provides a somewhat different approach since it gives a method to separate complexity classes by proving upper bounds, designing algorithms and efficient reductions.

Lemma 6.3. *Let $\mathcal{C}_1, \mathcal{C}_2$ be two complexity classes such that*:
(1) $\mathcal{C}_1 \subseteq \mathcal{P}/poly$

(2) *if* $\mathcal{C}_1 = \mathcal{C}_2$ *then for any* k, *there is a language in* $\mathcal{C}_2$ *that requires circuits of size* $\gg n^k$.
(3) *for some fixed* k, $\mathcal{C}_1$ *has circuits of size* $\leqslant n^k$ *with access to an oracle from* $\mathcal{C}_2$.
Then $\mathcal{C}_1 \neq \mathcal{C}_2$.

Proof. Let $\mathcal{C}_1 = \mathcal{C}_2$. Consider the fixed polynomial size circuit implied by (3). Since $\mathcal{C}_1 = \mathcal{C}_2$ the $\mathcal{C}_2$ oracle has also fixed polynomial size implied, by (1), and therefore all $\mathcal{C}_2$ has fixed polynomial size circuits. But this contradicts (2). □

7. Remarks

The results mentioned in this work can be viewed as a method of separating $\mathcal{NL}$ from $\mathcal{P}$ or $\mathcal{NP}$ (see Fortnow [5] for a related survey). Improving the algorithm for the finite state automata intersection emptiness problem would provide very interesting separation results as well as fast algorithms and simulations discussed in this work.

A uniform algorithm for the FSA intersection problem would separate $\mathcal{NL}$ from $\mathcal{P}$, but it would also provide a sub-exponential simulation of non-deterministic time (Theorem 4.1). Such a result might be considered unlikely, in which case our result should be considered as a strong indication that the FSA intersection problem is an explicit hard function, a good candidate for proving a non-linear lower bound.

On the other hand, separating $\mathcal{NL}$ from $\mathcal{NP}$ requires a non-uniform construction for the FSA problem. Since $\mathcal{NL}$ is believed to be different from $\mathcal{NP}$, it would be very interesting to see if the power of non-uniformity can be used to provide such a construction. It would also be interesting to see if there are other connections between similar problems and Complexity theory questions. Some of the ideas of this work could be applied for $\mathcal{NL}$-complete or other problems that capture $\mathcal{NL}$ computations leading to similar results and separations.

Acknowledgements

We would like to thank the anonymous referees for their comments.

References

[1] E. Bach, Number-theoretic algorithms, in: Annual Review of Computer Science, Vol. 4, Annual Reviews, Inc., 1990, pp. 119–172.
[2] R.G. Downey, M.R. Fellows, Fixed-parameter intractability (extended abstract), in: Proc. 7th Annual Structure in Complexity Theory Conf., Boston, MA, 22–25 June 1992, IEEE Computer Society Press, Silverspring, MD, pp. 36–49.
[3] R.G. Downey, M.R. Fellows, Parameterized Complexity, Springer, Berlin, 1999.
[4] U. Feige, J. Kilian, On limited versus polynomial nondeterminism, Chicago J. Theoret. Comput. Sci., March 1997.
[5] L. Fortnow, Diagonalization, Bull. Eur. Assoc. Theoret. Comput. Sci. 71 (2000) 102–112 (Columns: Computational Complexity).

[6] J. Hopcroft, W. Paul, L. Valiant, On time versus space, J. ACM 24 (2) (1977) 332–337.
[7] N.D. Jones, Space-bounded reducibility among combinatorial problems, J. Comput. System Sci. 11 (1975) 68–85.
[8] R. Kannan, A circuit-size lower bound, in: 22nd Ann. Symp. on Foundations of Computer Science, Los Alamitos, CA, USA, IEEE Computer Society Press, Silverspring, MD, October 1981, pp. 304–309.
[9] D. Kozen, Lower bounds for natural proof systems, in: 18th Ann. Symp. on Foundations of Computer Science, IEEE, London, October 1977, pp. 254–266.
[10] W.J. Paul, Nicholas Pippenger, Endre Szemerédi, William T. Trotter, On determinism versus non-determinism and related problems (preliminary version), in: 24th Ann. Symp. on Foundations of Computer Science, Tucson, Arizona, 7–9 November 1983, IEEE, London, pp. 429–438.
[11] W. Paul, R. Reischuk, On time versus space II, J. Comput. System Sci. 22 (3) (1981) 312–327.