On Hardness and Lower Bounds in Complexity Theory

Anastasios Viglas

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

January 2002

Abstract

Proving hardness results (lower bounds) is a major problem in Theoretical Computer Science. Many practical problems and applications are based on hardness assumptions (for example, public-key cryptography and security, pseudo-random generators, one-way functions, derandomization). Separating complexity classes is also a central question in Computational Complexity Theory that is closely connected with lower bounds. Although considerable effort and research has been focused in the area of lower bounds, very little progress has been made in attacking important open questions. There are however many interesting results for restricted computation models such as space bounded machines, bounded depth circuits monotone circuits and other models.

In this work we are going to discuss and present lower bounds for specific problems in restricted computation models, and show connections between hardness and several problems in Computational Complexity theory. In particular we are going to present the following four results:

First, we consider the well known problem of satisfiability (SAT). Proving a hardness result for this problem is a famous long standing open question, yet we have seen very little progress towards such a result, even for restricted computation models. We prove [LV99a] a non-linear time lower bound for solving satisfiability when restricted to use only a small amount of space (time-space trade-off for satisfiability). This was the first non-linear time bound for SAT in this model.

We also describe an interesting connection between the hardness of an explicit problem and complexity class separations. The problem we consider is that of deciding whether the intersection of a collection of finite state automata is empty: either this problem requires large circuits, or $\mathcal{NL}$ is not equal to $\mathcal{NP}$. For the uniform case, either this problem does not have fast algorithms or $\mathcal{NL}$ is not equal to $\mathcal{P}$. On the

other hand if this problem is easy, the we can design improved algorithms for subset sum and integer factoring, and we can also prove that non-deterministic time t can be simulated in deterministic time $2^{\epsilon t}$, for any positive ϵ. These results point to a new way of separating fundamental complexity classes ($\mathcal{NL}$ from $\mathcal{NP}$, or $\mathcal{NL}$ from $\mathcal{P}$) and raise many interesting questions.

Considering the connection of the class $\mathcal{P}$ and the non-uniform class of small depth and polynomial size circuits leads to some interesting results [LV01]: If every polynomial time computation can be done by a non-uniform circuit of polynomial size and sub-linear depth (for example if $\mathcal{P} \subseteq \mathcal{NC}/poly$), then $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c})$ for some constant c. This is a connection between the non-uniform depth of $\mathcal{P}$ and improving the well known result of Hopcroft, Paul and Valiant [HPV77] stating that space is more powerful than time. Similar techniques (based on block respecting turing machine computation) also allow us to prove unconditional results for the simulation of polynomial time computation by small depth semi-unbounded type of circuits.

A different approach to separating complexity classes is based on a connection between computational complexity and the length of proofs in propositional logic. We consider the correspondence between proof systems and computation models [LV99b]: starting from a class of automata, we can define a corresponding proof system in a natural way. A new proof system that can be defined through this correspondence is based on the class of push-down automata. This system is strictly more powerful than a certain variant of regular resolution and gives rise to many interesting questions.

Acknowledgments

It has been a great pleasure working these five years in such a great research environment as the one I found at Princeton University. The strong Theory community in the Princeton area helped making my PhD years productive and enjoyable.

My advisor, Richard Lipton, has provided valuable guidance and more help than I could ever hope for. It was a distinct experience working with him and I hope to continue this in the future. I would like to extend my thanks to Sanjeev Arora, Bernard Chazelle and Andy Yao for their help, and all the professors in the Computer Science Department for creating a pleasant working and learning environment. I am also grateful to Melissa Lawson for making almost everything in the Department a lot easier. Working in the Theory group at NEC research was another great experience. It was a pleasure working with Lance Fortnow, Ronnit Rubinfeld and many visitors and graduate students during the summer of 2000. My most sincere thanks also go to many of my good friends (too many to list here) who made my Princeton years enjoyable and gave me so many great memories.

στην οικογενεια μου:

To my parents, Christos and Ourania

and my brother, Kostas.

Contents

List of Figures

Chapter 1

Introduction

1.1 Hardness in Complexity Theory

Hardness is a central concept in Computational Complexity theory. Lower bounds are negative results, showing that a certain problem cannot be solved using less resources (less time or space for example) than a certain bound.

There are however, many positive applications of hardness results. The best example is cryptography and security: in this setting *not* being able to do something efficiently is actually desirable. Showing that a cryptographic system is secure requires a proof of a lower bound, a hardness result. A large part of todays widely used cryptographic schemes are based on assumptions about the computational difficulty of solving certain problems like factoring, discrete logarithms and many others. Another important "positive" aspect of hardness results is derandomization. Explicit hard problems with certain properties can be used to built strong pseudo-random generators, and can be used to derandomize algorithms.

Lower bounds also give very useful insight for designing algorithms, providing better understanding of the structure of a problem, and its behaviour with respect to different parameters, and in different computation models.

Separating complexity classes is also closely connected with hardness: questions about the power of non-determinism, randomness, or separating fundamental classes and computation models can be formulated as lower bound problems in a very natural way.

1.1.1 Lower bounds for different computation models

Although a lot of effort has been given in proving lower bounds, we have seen very little progress for general, unrestricted computation models. For example, the best lower bounds for any explicit problem in $\mathcal{NP}$, for Turing machine computation or boolean circuits is only linear.

For restricted models however we have seen remarkable progress and many interesting, strong lower bound results. Many different restricted systems have been considered: time-space tradeoffs, branching programs, monotone circuits, constant depth circuits, propositional proof systems, and many others.

In most cases, the restricted systems considered are natural and realistic models of computation, and the results in these models give insight, and help develop techniques and ideas to attack more and more general problems. These results also describe how a problem behaves under different restrictions, giving information that helps in understanding its structure and perhaps point to the correct directions for algorithm design.

1.2 Results and outline of this thesis

In this work we are going to discuss and present lower bounds for specific problems, for restricted computation models, and show connections between hardness and several problems in Computational Complexity theory.

In chapter 2 we prove a time-space trade-off lower bound for satisfiability [LV99a]: any Turing machine[1] using at most poly-logarithmic space cannot solve Satisfiability in less than (essentially) $n^{\sqrt{2}}$ time:

$$SAT \notin \mathcal{DTISP}(n^{\sqrt{2}-\epsilon}, poly\log n) \text{ for any } \epsilon > 0 \tag{1.1}$$

For example:

$$SAT \notin \mathcal{DTISP}(n^{1.4}, poly\log n) \tag{1.2}$$

It is easy to see that the best space bound that can be used in our proofs is sublinear, n^s for a small enough $s > 0$. This result improves on previous work by Kannan [Kan84] and Fortnow [For00b]. The time bound exponent has subsequently been improved by Fortnow and van Melkebeek [FvM00] to the golden ratio (1.6).

Also we can use roughly the same techniques to prove a corresponding result for uniform circuits. This conditional result is much weaker, and requires strong uniformity conditions for our circuits.

The main theorem easily extends for the random access machine model. Also, all the proofs work for sub-linear space instead of the poly-logarithmic space bound as mentioned above.

Some interesting extensions to this work have been presented by Tourlakis [Tou00]. The time-space trade-offs for satisfiability are generalized for the case of limited non-uniformity by considering machines that take sublinear advice.

[1]Our proof applies both to multi-tape Turing machines as well as random access machines

Chapter 3 presents connections between the hardness of a problem from classic automata theory and different Computational Complexity questions [KLV00].

We consider (uniform and non-uniform) assumptions for the hardness of deciding whether the intersection of a collection of finite state automata (FSA) is empty.

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 $\mathcal{NP}$). 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.

In more detail, the uniform version of our assumption (if there exists a fast algorithm for the problem considered) will allow us to prove the following results:

1. There is an algorithm solving sub-set sum in $O(n^c \cdot 2^{\epsilon n})$ for all $\epsilon > 0$ and some small constant c
2. Integer factorization can be solved in $O(n^c \cdot 2^{\epsilon n})$ for all $\epsilon > 0$ and c a constant
3. $\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(2^{\epsilon t})$ for all $\epsilon > 0$
4. $\mathcal{NL} \neq \mathcal{P}$

The non-uniform version of the assumption (if there exists a small circuit solving the FSA problem) implies that we can separate $\mathcal{NL}$ from $\mathcal{NP}$ using a different argument.

In chapter 4 we present some new results based on block respecting computation. The well known result of Hopcroft, Paul and Valiant [HPV77] showing that space is more powerful than time, can be improved by making a (strong) assumption about the connection of (linear) deterministic time computations and polynomial size, non-uniform, small (sub-linear) depth circuits: If every polynomial time computation can be done by a non-uniform circuit of polynomial size and sub-linear depth (for example if $\mathcal{P} \subseteq \mathcal{NC}/poly$), then $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c})$ for some constant c.

In chapter 5 we look at the connections between hardness in Complexity theory and propositional proof systems, and present some results on the correspondence between computation models and proof systems.

One way to address the $\mathcal{NP} \overset{?}{=} co\text{-}\mathcal{NP}$ question is to consider the length of proofs of tautologies in various proof systems. A well known theorem [CR79] states that if there exists a *propositional proof system* (with certain properties) in which every tautology has polynomial size proofs, then $\mathcal{NP} = co\text{-}\mathcal{NP}$. A natural question to ask is the following: is there a direct correspondence between a propositional proof system and a computation model ? We know for example, that the proof system of regular resolution corresponds to read-once branching programs in a very natural way; the size of the smallest proof of a tautology in regular resolution is the same as the size of the smallest read-once branching program that solves a search problem associated with the tautology.

In this section we will try to look closer to the correspondence between computation models and propositional proof systems. We consider proof systems defined by appropriate classes of automata [LV99b]: In general, starting from a given class of automata we can define a corresponding proof system in a natural way. An interesting new proof system that we consider is based on the class of push down automata. We present an exponential lower bound for oblivious read-once branching programs which implies that the new proof system based on push down automata is strictly more powerful than oblivious regular resolution. This result gives a characterization of this new system and its power, compared to classic systems from propositional logic.

Chapter 2

Time-Space trade-offs for Satisfiability

Although Satisfiability is considered a hard problem, *proving* any strong hardness result for it seems to be beyond current knowledge. For restricted systems however, we can prove interesting, non-trivial lower bound results. In this section, we prove a time-space trade-off result for Satisfiability: we consider space bounded computation, and show non-linear time lower bounds [LV99a]. This time-space trade-off result is based on simulation and diagonalization techniques.

2.1 Introduction: Trading time for space

In many cases, we can reduce the space required to carry out a specific computation, by allowing the algorithm to use more time. On the other hand, time savings can be achieved if more space is available. This relation can be described by time-space trade off results in various models of computation.

Time space trade-offs are central in Complexity Theory, capturing important aspects of the intrinsic hardness of specific computations, describing the way a problem behaves with respect to time and space.

Time-space trade-offs can also be viewed as lower bound results for restricted computation models. If for example we consider a computation that is restricted to use a certain amount of space, then a trade-off could translate to a lower bound for the time required by this space-bounded computation.

We are interested in proving such time-space lower bound results for the well known problem of Satisfiability. This is the problem of deciding whether a given propositional formula has a satisfying truth assignment. The best lower bounds known for *SAT*, for general computation models are essentially trivial. It is interesting to point out that even though we believe that no polynomial time algorithm could solve this problem, we are unable to rule out the possibility that a *linear* time algorithm exists.

If we consider space bounded computation, then it is possible to prove non-trivial lower bounds for the time required to solve Satisfiability. More specifically, in this section we will prove that any Turing machine[1] using at most poly-logarithmic space cannot solve Satisfiability in less than (essentially) $n^{\sqrt{2}}$ time:

$$SAT \notin \mathcal{DTISP}(n^{\sqrt{2}-\epsilon}, poly\log n) \text{ for any } \epsilon > 0 \tag{2.1}$$

or:

[1]Our proof applies both to multi-tape Turing machines as well as random access machines (the RAM model was discussed in more detail in the work by Fortnow and van Melkebeek [FvM00])

$$SAT \notin \mathcal{DTISP}(n^{1.4}, poly\log n) \tag{2.2}$$

It is easy to see that the best space bound that can be used in our proofs is sublinear, n^s for a small enough $s > 0$. This results improves on previous work by Kannan [Kan84] and Fortnow [For00b]. The exponent has subsequently been improved by Fortnow and van Melkebeek [FvM00] to the golden ratio (1.6).

Also we can use roughly the same techniques to prove a corresponding result for uniform circuits. This result is much weaker, and requires strong uniformity conditions for our circuits. The model we will use is uniform circuits of size t and *width d*. The uniformity condition must be strong enough to provide us with a Turing machine that will produce an appropriate description of the circuit in logarithmic time; the Turing machine will receive as input the index of a gate in the circuit and will output the connections of this gate in the circuit in *log* time[2] (input and output connections). In this model we can prove the following: Assume that the class of non-deterministic circuits of size t is not a subset of the class of circuits of size $t^{1-\epsilon}$ for positive ϵ. Then there are non-deterministic circuits of size n that cannot be simulated by deterministic uniform circuits of size $n^{\sqrt{2}-\epsilon}$ and poly-logarithmic (or sub-linear) width.

The basic structure and the main ideas used in the proof of the uniform circuit result are in general the same as in the Turing machine case (note again that for the uniform circuit case we will only describe a conditional result).

The main theorem easily extends for the random access machine model. Also, all the proofs work for sub-linear space instead of the poly-logarithmic space bound as

[2]In fact *poly*log time or even sublinear time would also work here.

mentioned above.

Some interesting extensions to this work have been presented by Tourlakis [Tou00]. The time-space trade-offs for satisfiability are generalized for the case of limited non-uniformity by considering machines that take sublinear advice.

2.1.1 Related Work

The first time-space lower bounds for Satisfiability were presented by Fortnow [For00b]. The main results for SAT in that work were the following:

$$\overline{SAT} \notin \mathcal{NTISP}(n^{1+o(1)}, n^{1-\epsilon})$$
$$SAT \notin \mathcal{DTISP}(n^{1+o(1)}, n^{1-\epsilon})$$
$$\overline{SAT} \notin \mathcal{NTIME}(n \log^{O(1)} n) \cap \mathcal{NL}$$

In the work described in this section, we use similar ideas and techniques as in [For97, For00b] to prove our time-space lower bound result for non-deterministic polynomial time which will be used to derive the lower bound for SAT:

$$SAT \notin \mathcal{DTISP}(n^{\sqrt{2}-\epsilon}, \textit{poly}\log n) \text{ for any } \epsilon > 0 \tag{2.3}$$

Other related results were also shown in [Kan84]. Kannan proved that there is a constant $k > 0$ (not constructively defined) such that for all polynomial time constructible functions $t(n)$ where $t(n) = \omega(n^k)$, there is a language accepted in non-deterministic time t but not in deterministic time t and (simultaneously) space $o(t^{1/k})$. This also implies that for any positive integer $l \neq k$, non-deterministic time $\mathcal{NTIME}(n^l)$ is not a subset of deterministic time $\mathcal{DTISP}(n^l, \textit{poly}\log n)$. In fact Kannan's techniques and the basic structure of his main proof are very similar to the ones presented in this work.

Gurevich and Shelah [GS90] defined a clique problem solvable by non-deterministic log-space Turing machines in linear time, which cannot be solved by deterministic machine with sequential access to its input tape and work space n^σ in time $n^{1+\tau}$ where $\sigma + \tau < 1/2$.

Other related (earlier) results for uniform models of computation were given by Hopcroft, Paul and Valiant [HPV77] (deterministic time t is strictly contained in deterministic space t for well behaved functions $t = t(n)$), Paul and Reischuk [PR80] (simulation of a deterministic Turing machine by an alternating machine) and Paul, Pippenger, Szemerédi and Trotter [PPST83] (non-deterministic linear time is not a subset of deterministic linear time).

Time-space lower bounds have also been shown for other models such as Jumping Automata for Graphs (JAG) [Edm98, BE98, EP95], branching programs [Kar86, BST98, Bea91] and comparison models [Yao94].

A very successful (non-uniform) model for proving time-space trade-offs is the branching program model [Bor93]. Many techniques have been developed and strong results have been shown for multi-output problems [BC82, Bea91, Abr90], element distinctness [Yao88] and read-k time branching programs [BRS93, BST98]. Strong super-linear time-space trade-offs have been shown using this model for problems in polynomial time. The breakthrough work of Ajtai [Ajt98, Ajt99b, Ajt99a] showed how to prove super-linear trade-offs for element distinctness and for the computation of natural quadratic forms. Based on the techniques developed in that work, Beame, Saks, Sun and Vee [BSSV00] have shown super-linear time-space trade-offs for randomized computation for (decision) problems in $\mathcal{P}$.

2.2 Preliminaries

We will use the multi-tape Turing machine model to describe the proof of the main theorem. It is easy to see that all our arguments carry over to Random Access machines.

Let $\mathcal{NTIME}(t)$ and $\mathcal{DTIME}(t)$ denote the classes of all languages accepted by Turing machines running in nondeterministic time $O(t)$ and deterministic time $O(t)$ respectively. Also, we will use $\mathcal{DTIME}(t,s)$ ($\mathcal{NTIME}(t,s)$) to denote the class of all languages accepted by deterministic (non-deterministic) Turning machines running in time $O(t)$ and using space $O(s)$. For simplicity, we will use $\mathit{poly}\log n$ instead of $\log^{O(1)} n$ to denote poly-logarithmic functions of n. Let M be a Turing machine and C_1, C_2 two configurations of M (the description of a Turing machine configuration contains information about the state, the position of the heads on the tapes and the contents of all tapes). We will write $C_1 \vdash_M^* C_2$ if there is a legal sequence of computation steps of M starting from configuration C_1 and ending with configuration C_2. $C_1 \vdash_M C_2$ means that M starting form C_1 can reach C_2 in one step.

A function $t(n)$ is called polynomial time constructible if there exists a deterministic Turing machine which on any input of length n computes $t(n)$ in polynomial time.

A circuit family $\{C_n\}$ is called log-space (time) uniform if there exists a log-space (time) Turing machine M_C that on input 1^n will output the circuit C_n. For the circuit setting, let $\mathcal{NSIZE}_U(t)$ (and $\mathcal{DSIZE}_U(t)$) denote the class of non-deterministic, *log*-time uniform circuits with t gates (and deterministic size t circuits respectively). $\mathcal{DWIDTH}_U(s,w)$ will denote the class of deterministic log-time uniform circuits with s gates and width w.

2.3 Time-space lower bounds for Satisfiability

In this section we are going to describe some of the methods that have been developed for proving time-space trade-off results for Satisfiabilty (see also a survey by van Melkebeek [vM01]).

2.3.1 Reducing $\mathcal{NTIME}$ to *SAT*

Throughout this chapter we will prove relations between non-deterministic time classes and space bounded deterministic time. Strong enough results of this form (super-linear time and at least poly-logarithmic space) imply lower bounds for Satisfiability using Cook's theorem and reductions of non-deterministic time computations to Satisfiability questions.

SAT is known [Sch78, Coo88] to be complete for quasi-linear nondeterministic time $\mathcal{NTIME}(n \log^{O(1)} n)$, under quasi-linear time reductions [Sch78, Coo88].

Theorem 2.3.1 ([Sch78, Coo88]) SAT *is complete for quasi-linear non-deterministic time under (deterministic) quasi-linear time reductions.*

The reduction of $\mathcal{NTIME}(n\, poly\log n)$ described in the above theorem 2.3.1 and is explicitly given by Cook [Coo88] can also be done "on the fly" using small, poly-logarithmic space. The following theorem is not the strongest form, but good enough for the results that will follow. In particular the theorem can be extended for alternating computation and more tight bounds for the required time and space are known.

Theorem 2.3.2 ([Coo88, Sch78]) *Any non-deterministic computation (of a multi-tape Turing machine) of length $t(n)$ can be reduced to a* SAT *formula of size $t\, poly\log t$. This algorithm will output the i-th bit of the* SAT *formula in poly-logarithmic time and space.*

The results of Robson [Rob91] for reducing RAM computations to SAT formulas can be used to extend theorems for non-deterministic time to equivalent results for Satisfiability.

We can use the above theorems to translate relations between non-deterministic time classes and deterministic time-space, into lower bounds for Satisfiability. If, for example, we can prove that $\mathcal{NTIME}(n) \not\subseteq \mathcal{DTIME}(\tau, s)$ then we can argue that $SAT \notin \mathcal{DTIME}(\tau, s)$ for appropriate values of the parameters τ, s.

Theorem 2.3.3 *For any superlinear time bound $\tau(n) >> O(nlog^{O(1)}n)$, and poly-logarithmic space $s(n) = \log^{O(1)} n$,*

$$\mathcal{NTIME}(n) \not\subseteq \mathcal{DTISP}(\tau, s) \tag{2.4}$$

implies that

$$\text{SAT} \notin \mathcal{DTISP}(\tau, s) \tag{2.5}$$

Proof

The idea is the following: assume that SAT can be solved in $\mathcal{DTISP}(\tau, s)$. Consider any computation in $\mathcal{NTIME}(n)$. Reduce this to a SAT question, and solve it in $\mathcal{DTIME}(\tau, s)$. The reduction uses $n \log^{O(1)} n$ time but the resulting SAT formula has size $n \log^{O(1)} n$ which is bigger than the space available (we only have poly-logarithmic space $s = \log^{O(1)} n$. We do not have to write down the resulting formula however, since by theorem [Coo88] there is an algorithm using only polylogarithmic space that provides random access to the bits of the SAT formula. ▮

2.3.2 Proving uniform time-space lower bounds

The main idea for proving time-space trade-off results on uniform models can be described as follows:

1. Assume that some strong simulation of non-deterministic time is possible (in order to reach a contradiction, and therefore prove that this simulation cannot exist). This assumption will be of the form $\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(t', s)$.

2. Start from some non-deterministic computation, and argue that it can be reduced to a deterministic time-space computation, using the assumption described in the first step.

3. Break the (deterministic) computation of the previous step into blocks and simulate it by an alternation, which will save time based on the power of the alternations (*"trade time for alternations"*).

4. Look at each part of the alternation and use the initial assumption of the first step to reduce the alternations (*"Trade alternations for time"*).

5. The resulting computation (which simulates the original one) is "too fast", violating known time hierarchy theorems (*"diagonalization"*).

The contradiction that we will reach in the final step derives from hierarchy theorems, which are based on direct diagonalization, a well known and old technique that continues to prove its strength with new results (see [For00a] for an interesting account/survey of diagonalization methods in current Complexity Theory research.)

The hierarchy theorem that we will use in the following section to prove the main trade-off result for Satisfiability is the well known non-deterministic time hierarchy theorem [Coo73, SFM73, SFM78].

Theorem 2.3.4 (Non-deterministic Time Hierarchy [SFM78]) *For time constructible functions $t_1(n)$ and $t_2(n)$ such that $t_1(n+1) = o(t_2(n))$:*

$$\mathcal{NTIME}(t_1) \subset \mathcal{NTIME}(t_2) \tag{2.6}$$

The following lemma, is much weaker statement of the non-deterministic time-hierarchy theorem, but will be enough for our purposes:

Lemma 2.3.5 *For any $\epsilon > 0$, there is a language with non-deterministic time complexity $\mathcal{NTIME}(t)$ that is not computable in $\mathcal{NTIME}(t^{1-\epsilon})$.*

$$\mathcal{NTIME}(t) \not\subseteq \mathcal{NTIME}(t^{1-\epsilon}) \tag{2.7}$$

Another, more technical result that is used in the time-space lower bounds, is the following [Kan84]:

Lemma 2.3.6 (Translation Lemma) *If for some $\alpha > 0$, $\mathcal{NTIME}(n) \subseteq \mathcal{DTISP}(n^\alpha, poly\log n)$ then for any $t = n^c$, $c > 1$, $\mathcal{NTIME}(t) \subseteq \mathcal{DTISP}(t^\alpha, poly\log t)$.*

Proof (padding)

Let $\mathcal{NTIME}(n) \subseteq \mathcal{DTISP}(n^\alpha, poly\log n)$. We will show that any language in $\mathcal{NTIME}(t)$ is also in $\mathcal{DTISP}(t^\alpha, poly\log t)$. Consider any language $L \in \mathcal{NTIME}(t)$. Define a new language $L' = \{x\sqcup^{t-|x|}; x \in L\}$; L' consists of all the strings in L padded by enough quasi-blanks to bring the length of the string to t. It is easy to see that $L' \in \mathcal{NTIME}(n)$. Since $\mathcal{NTIME}(n) \subseteq \mathcal{DTISP}(n^\alpha, poly\log n)$ it follows that $L' \in \mathcal{DTISP}(n^\alpha, poly\log n)$. Therefore $L \in \mathcal{DTISP}(t^\alpha, poly\log t)$ (on input x, consider $y = x\sqcup^{t-|x|}$ and solve $y \in L'$ in deterministic time-space $\mathcal{DTISP}(|y|^\alpha, poly\log |y|)$).

Note that we only have poly-logarithmic space available for our computations. For the new language L' mentioned above, we need to pad the input strings by quasi-blanks so that the resulting string has size t which will be bigger that the available space. We do not need to write down this new string though. The padded part of the input consist by all quasi-blanks and therefore all we need to know is which position in the input string we are reading. In other words we only need the original input string and a counter, that will be used when the input pointer goes into the padded part of the string. This counter only needs space $\log t$. ∎

2.3.3 Previous work

The first techniques dealing with the space bounded model that we will consider next, were given by Kannan [Kan84]. His work addressed a different problem: the relation between deterministic and non-deterministic time. His main result was that linear non-deterministic time cannot be simulated by linear deterministic time using sub-linear space.

$$\mathcal{NTIME}(n) \not\subseteq \mathcal{DTISP}(n, o(n)) \tag{2.8}$$

This theorem does not imply a time-space lower bound for Satisfiability, but the techniques used have been proven to be very useful in later results. In particular the results in [LV99a] have the same basic structure with Kannan's.

Theorem 2.3.7 (Kannan [Kan84]) *There is a universal (non-constructible) constant k, such that, for any integer $j \geq k+1$*

$$\mathcal{NTIME}(n^j) \not\subseteq \mathcal{DTISP}(n^j, o(n)) \tag{2.9}$$

The constant k from the above theorem is defined as follows:

- $k = 1$ if $\mathcal{NTIME}(n^j) \not\subseteq \mathcal{DTIME}(n^j)$, for all $j \geq 1$, or
- $k = min\,\{j \geq 1 : \mathcal{NTIME}(n^j) = \mathcal{DTIME}(n^j)\}$ otherwise.

The above theorem addresses the following question: Is $\mathcal{NTIME}(t)$ equal to $\mathcal{DTIME}(t)$ for polynomial t ? Kannan [Kan84] proves a partial result towards that goal, for restricted space (the strongest version of the theorem uses the constant k).

Using a somewhat different approach, Fortnow [For97, For00b] was able to prove stronger results, and the first (slightly) super-linear time-space trade-off for Satisfiability.

Theorem 2.3.8 (Fortnow) *For any unbounded function $r(n)$ such that $r(n) = \frac{\log n}{\log\log n}$ and for any $\epsilon > 0$*

$$\overline{\text{SAT}} \notin \mathcal{NTIME}(n^{1+\frac{1}{r}}) \cap NTISP(n^{o(r)}, n^{1-\epsilon}) \tag{2.10}$$

A number of interesting results follow from this theorem:

$$\overline{SAT} \notin \mathcal{NTISP}(n^{1+o(1)}, n^{1-\epsilon})$$
$$SAT \notin \mathcal{DTISP}(n^{1+o(1)}, n^{1-\epsilon})$$
$$\overline{SAT} \notin \mathcal{NTIME}(n^{1+o(1)}) \cap \mathcal{NL}$$
$$\overline{SAT} \notin \mathcal{NTIME}(n \log^{O(1)} n) \cap \mathcal{NL}$$

The proofs of these theorems follow the same spirit as mentioned above but different methods and ideas are use in some of the steps. This results to strong results for $\overline{SAT}$ as well as some interesting uniform circuit results. The main theorem 2.3.8 is stated for $\overline{SAT}$. Starting from the assumption that $\overline{SAT}$ can be solved in non-deterministic time-space of the form $\mathcal{NTISP}(n^{1+o(1)}, n^{1-\epsilon})$, it is argued that an alternating computation can be simulated "too fast", in a new alternating class, with fewer alternations using less time. This simulations is achieved by reducing parts of the computation to $\overline{SAT}$ questions and then using the initial assumption to reduce the time and space, and argue that the resulting computation is in a new alternating class. Then, a direct diagonalization between the two alternating classes completes the proof.

Based on these ideas, the improved time-space lower bounds for Satisfiablity described in the next sections can be derived [LV99a]. Compared to Fortnow's [For00b] theorems, these results show a stronger time-bound (about $n^{1.4}$) but assume more restricted space bounds (sublinear space). These results where improved even further by Fortnow and van Melkebeek [FvM00]. Other extensions were considered by Tourlakis [Tou00]. These improvements are also discussed in section 2.6.

2.4 A time-space Lower bound for $\mathcal{NTIME}$ and SAT

Using many of the ideas mentioned in the previous section, we can prove the first essentially non-linear time lower bound for space bounded Turing Machine computation. The main theorem proved in this section, is the following.

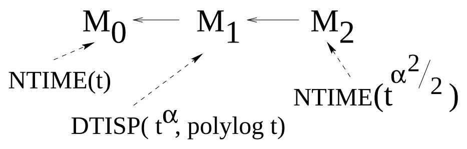
Theorem 2.4.1 *If $\mathcal{NTIME}(n) \subseteq \mathcal{DTISP}(n^\alpha, poly\log n)$ then $\alpha \geq \sqrt{2} - \epsilon$ for any $\epsilon > 0$.*

The proof will proceed as follows:

- Assume that the theorem is not true (and therefore $\mathcal{NTIME}(n) \subseteq \mathcal{DTISP}(n^\alpha, poly\log n)$ for some $\alpha < \sqrt{2}$ call this the "initial assumption").
- Start with any non-deterministic computation running in time t and show how to simulate it in slightly less non-deterministic time (which is a contradiction with the known hierarchy theorems).

To describe this *"too-efficient"* simulation of the second step, we will proceed as follows:

- Denote by M_1 the nondeterministic computation (Turing machine) we are starting from. This is a $\mathcal{NTIME}(t)$ computation.
- M_1 can be simulated by a computation M_2 in $\mathcal{DTISP}(t^\alpha, poly\log n)$ (by our initial assumption)
- M_2 can be simulated by an alternating computation M_3 using less time.
- Using again the initial assumption, we can reduce the alternations and end up with a non-deterministic computation in $\mathcal{NTIME}(t^{\alpha^2/2})$ roughly.
- Argue that if α is smaller than $\sqrt{2}$, then we get a contradiction with the hierarchy results.



Proof (of theorem 2.4.1) Assume the following:

$$\mathcal{NTIME}(n) \subseteq \mathcal{DTISP}(n^{\alpha}, \mathit{poly}\log n) \tag{2.11}$$

For any $t = n^c$, $c > 1$, lemma 2.3.6 implies:

$$\mathcal{NTIME}(t) \subseteq \mathcal{DTISP}(t^{\alpha}, \mathit{poly}\log t) \tag{2.12}$$

We will show that if $\alpha < \sqrt{2}$ then $\mathcal{NTIME}(t)$ is contained in $\mathcal{NTIME}(t^{1-\beta})$ for some $\beta > 0$; this contradicts lemma 2.3.5 (non-deterministic time hierarchy theorem).

Consider any language $L \in \mathcal{NTIME}(t)$ and let M_1 denote the non-deterministic Turing machine deciding L running in time t. Since $\mathcal{NTIME}(t) \subseteq \mathcal{DTISP}(t^{\alpha}, \mathit{poly}\log t)$ it follows that there is a deterministic machine M_2 running in time-space $\mathcal{DTISP}(t^{\alpha}, \mathit{poly}\log n)$ deciding L.

Consider the computation of M_2 (t^{α} steps). Each configuration of M_2 has size $\mathit{poly}\log n$ since M_2 runs in space $\mathit{poly}\log n$. Break up the computation in $t^{\alpha/2}$ blocks, of size $t^{\alpha/2}$ and let $A_1, \ldots, A_{t^{\alpha/2}}$ denote the configurations of M_2 every $t^{\alpha/2}$ steps; that is A_1 is the initial starting configuration, A_2 is the configuration after $t^{\alpha/2}$ steps of computation, and so on; $A_{t^{\alpha/2}}$ is the final configuration.

We will construct a non-deterministic machine M_3 that simulates M_2 efficiently; on input x ($|x| = n$), M_3 will proceed as follows[3]:

[3]The technique described here is a variant of a standard method that was also used in [For97, Kan84].

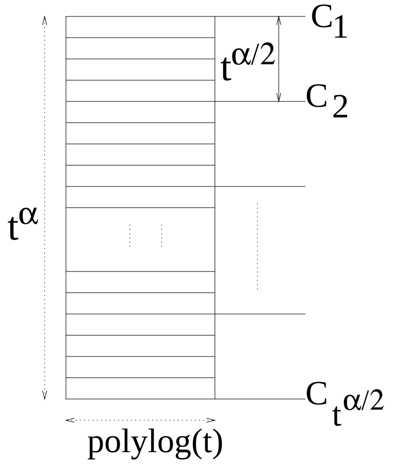


Figure 2.1: Break computation into blocks

$$\exists A_1, \ldots, A_{t^{\alpha/2}} \forall i < t^{\alpha/2} (A_i \vdash_M^* A_{i+1}) \tag{2.13}$$

1. guess the configurations $A_1, \ldots, A_{t^{\alpha/2}}$ of M_2 on input x, every $t^{\alpha/2}$ steps of computation.

2. check that A_1 is a legal starting state and $A_{t^{\alpha/2}}$ is a legal final state and that for every $0 < i < t^{\alpha/2}$, $(A_i \vdash_M^* A_{i+1})$. To find out if there is a configuration A_i such that M_2 will not reach A_{i+1} after $t^{\alpha/2}$, universally choose (guess) the index i and check if $(A_i \vdash_M^* A_{i+1})$.

In very simple terms the above alternation states the following (see figure 2.2): To express the fact that "there is a way to get from the point A to the point B" on a line, we can also say, "there exist intermediate *check-points* $x_1, \ldots, x_n$, such that for each intermediate point there is a way to get to the next (for all i we can get from x_i to x_{i+1})." This alternate expression is the alternation described for this proof.

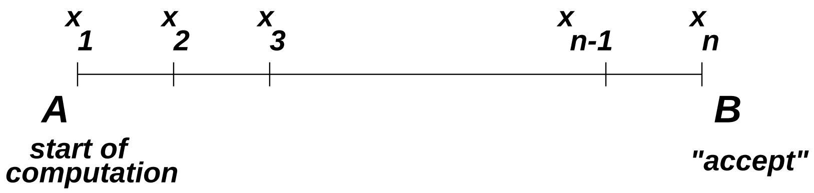


Figure 2.2: Getting from A to B

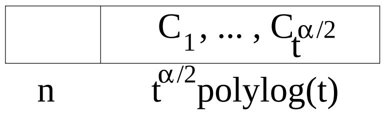


Figure 2.3: The input of the non-deterministic computation (second step).

The first step requires non-deterministic time $t^{\alpha/2}\,poly\log t$. The second step is a non-deterministic computation on input of size $n + t^{\alpha/2}\,poly\log t$ (n is the size of the initial input x, $t^{\alpha/2}$ is the number of steps from A_i to A_{i+1} and $poly\log t$ is the size of each configuration) and requires time $n + t^{\alpha/2}\,poly\log t$: in order to check if $A_i \vdash_M^* A_{i+1}$ we need to move the input head to the correct position in the input x and simulate M_2 for $t^{\alpha/2}$ steps.

Assumption 2.12 implies[4] that this task can be done in deterministic time-space $\mathcal{DTISP}((n + t^{\alpha/2}\,poly\log t)^{\alpha},\, poly\log t)$. For large enough $t \gg n$, $(n + t^{\alpha/2}\,poly\log t)^{\alpha} \simeq t^{\alpha^2/2}\,poly\log t$. M_3 runs in non-deterministic time

$$t^{\alpha/2}\,poly\log t + t^{\alpha^2/2}\,poly\log t \tag{2.14}$$

[4]Note that it suffices to use assumption 2.11 for this case.

Let $\alpha_\epsilon = \alpha + \epsilon$; then for any $\epsilon > 0$, $t^{\alpha/2} poly\log n \ll t^{\alpha_\epsilon/2}$ for large enough t. This implies that any non-deterministic time t machine can be simulated in non-deterministic time $t^{\alpha_\epsilon^2/2}$:

$$\mathcal{NTIME}(t) \subseteq \mathcal{NTIME}(t^{\alpha_\epsilon^2/2}) \tag{2.15}$$

If $\alpha_\epsilon^2/2 < 1$ then we get a contradiction with lemma 2.3.5. ▮

2.4.1 Improving the time-space lower bound

In order to improve theorem 2.4.1, we could try to use one additional idea when simulating the $\mathcal{DTISP}(t^\alpha, poly\log n)$ machine M_2. The idea is to break the computation recursively into smaller and smaller blocks. The technique that we will describe below will *not* enable us to improve our result in a straighforward way. It is possible however to improve the time bound by combining additional ideas. Fortnow and van Melkebeek were able to show that this technique can be used to improve the bound to the golden ratio by reversing the order of the quantifiers (see [FvM00] for the improved bound).

We will describe this method briefly. Let $\mathcal{NTIME}(n) \subseteq \mathcal{DTISP}(n^\alpha, poly\log n)$ for some $\alpha > 0$. By lemma 2.3.6 we have the following (for $t = n^c$, $c > 1$):

$$\mathcal{NTIME}(t) \subseteq \mathcal{DTISP}(t^\alpha, poly\log t) \tag{2.16}$$

for some $\epsilon > 0$. Proceed in the same way as in theorem 2.4.1. The main difference is in equation 2.13. Recall that the computation of any machine M_2 from

$\mathcal{DTISP}(t^\alpha, \mathit{poly}\log t)$ can be simulated as follows:

$$\exists A_1, \ldots, A_{t^{\alpha/2}} \forall i < t^{\alpha/2} (A_i \vdash_M^* A_{i+1}) \tag{2.17}$$

There are $t^{\alpha/2}$ steps between A_i and A_{i+1}. In other words we have broken down the computation of M_2 into blocks of size $t^{\alpha/2}$ and A_i is the first step of the i-th block. For simplicity we will refer to the i-th block as "block A_i".

Consider breaking the computation of M_2 in blocks A_i of size t^a for some $a < \alpha$ that we will choose appropriately. Then, in order to check the transition in a block $A_i \vdash_M^* A_{i+1}$, recurse once more and break the computation in block A_i in sub-blocks B_j of size t^b each.

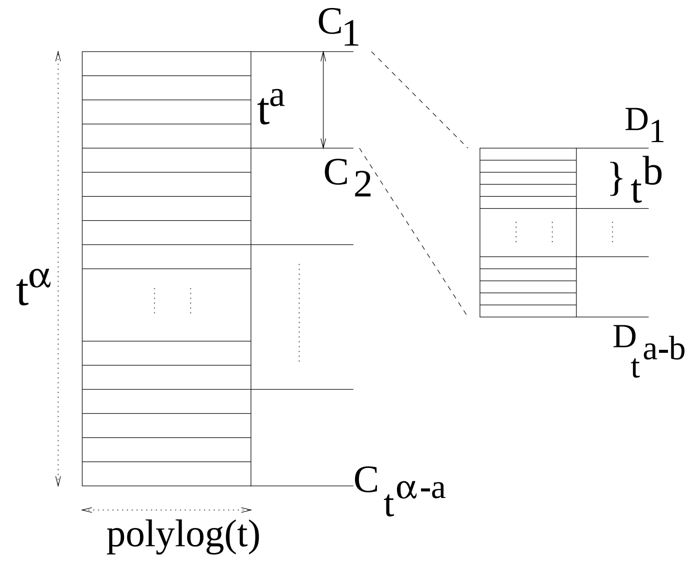


Figure 2.4: Break the computation into sub-blocks recursively

Note that we can simulate the computation of M_2 by a non-deterministic machine M_3 in the following way:

$$\begin{aligned} &\exists A_1, \ldots, A_{t^{\alpha-a}} \forall i < t^{\alpha-a} \\ &\exists B_1, B_2, \ldots B_{t^{a-b}} \forall j < t^{a-b} (B_j \vdash^* B_{j+1}) \end{aligned} \tag{2.18}$$

1. Guess the computation of M_2, every t^a steps: $A_1, \ldots, A_{t^{\alpha-a}}$
2. Check that the computation steps are correct: guess an index i and check if the transition $A_i \vdash_M^* A_{i+1}$ is correct;
3. In order to check whether $A_i \vdash_M^* A_{i+1}$, break the computation from A_i to A_{i+1} in sub-blocks; of size t^b (the number of sub-blocks is t^{a-b}), and guess the configurations $B_1, \ldots, B_{t^{a-b}}$ of M_2 every t^b steps starting from A_i and ending with A_{i+1}.
4. To verify that for all $0 < j < t^{a-b}$, $B_j \vdash_{M_2}^* B_{j+1}$, check if there is an incorrect computation step B_j; universally choose the index j of the incorrect step and check if $B_j \vdash_M^* B_{j+1}$.

The last step is a non-deterministic computation on input of size $n + t^{a-b}\, poly\log t$ (the original input x of M_2 and t^{a-b} blocks (B_j) of size $poly\log t$ each) and requires time $n + t^b\, poly\log t$. By assumption 2.16 this is in deterministic time-space $\mathcal{DTISP}((n + t^b\, poly\log t)^\alpha, poly\log t)$ or $\mathcal{DTISP}(t^{b\alpha}\, poly\log t, poly\log t)$ for large enough t. Steps 3 and 4 are a non-deterministic computation on input of size $n + t^{\alpha-a}\, poly\log t$ and require time $n + t^{a-b}\, poly\log t + t^{b\alpha}\, poly\log t$. By assumption 2.16 this can be done deterministically in time $t^{\alpha(a-b)}\, poly\log t + t^{\alpha^2 b}\, poly\log t$. Steps

2,3,4 form a non-deterministic computation and require time

$$\begin{aligned}&\mathcal{NTIME}(t^{\alpha(a-b)}\, poly\log t + t^{\alpha^2 b}\, poly\log t) \subseteq \\ &\qquad \mathcal{DTISP}(t^{\alpha^2(a-b)}\, poly\log t + t^{\alpha^3 b}\, poly\log t,\, poly\log t)\end{aligned} \tag{2.19}$$

Finally the entire computation of M_3 is in non-deterministic time

$$\begin{aligned}&\mathcal{NTIME}(t^{\alpha-a}\, poly\log t + t^{\alpha^2(a-b)}\, poly\log t + \\ &\qquad t^{\alpha^3 b}\, poly\log t)\end{aligned} \tag{2.20}$$

(for large enough $t \gg n$).

As in the proof of theorem 2.4.1, we can drop the poly-log factors, by adding any small constant $\delta > 0$: $\alpha_\delta = \alpha + \delta$. This will give us the following time bound:

$$t^{\alpha_\delta - a} + t^{(a-b)\alpha_\delta^2} + t^{b\alpha_\delta^4} \tag{2.21}$$

If we continue for w alternations (for any fixed w) and each time we break the computation in blocks of size t^{a_i}, $0 < i \leq w$, then we get a time bound of the form:

$$\begin{aligned}&t^{(\alpha-a_1)} + t^{(a_1-a_2)\alpha_\delta^2} + \cdots \\ &+ t^{(a_{w-1}-a_w)\alpha_\delta^{2w-2}} + t^{a_w \alpha_\delta^{2w-1}}\end{aligned} \tag{2.22}$$

One would hope that by picking the parameters a_i appropriately the lower bound of theorem 2.4.1. More careful analysis of this method [FvM00] shows that this technique will not improve the exponent. However as we mentioned before by reversing the order of quantification it has been shown that the bound can be improved up to the golden ratio [FvM00].

Corollary 2.4.2 *Any deterministic Turing machine solving Satisfiability that uses poly-logarithmic space, requires infinitely often almost $n^{\sqrt{2}}$ time:*

$$\mathrm{SAT} \notin \mathcal{DTISP}(n^{\sqrt{2}-\epsilon}, \mathit{poly}\log n) \tag{2.23}$$

for any $\epsilon > 0$.

This corollary follows from the discussion in section 2.3.1 and theorems 2.3.2, 2.3.3.

2.5 Lower Bounds for Uniform Circuits

For the circuit setting we will consider the class $\mathcal{NSIZE}_U(n)$ of uniform[5] non-deterministic circuits of size $s(n)$, where n is the size of the input. The analog of $\mathcal{DTISP}(n^\alpha, \mathit{poly}\log n)$ in this setting will be the class $\mathcal{SIZEWD}_U(n^\alpha, \mathit{poly}\log n)$ of (deterministic) uniform circuits of size $O(n^\alpha)$ and width $\mathit{poly}\log n$.

Using the same methods as in theorem 2.4.1 we can prove that if every circuit in $\mathcal{NSIZE}_U(n)$ can be simulated by a deterministic log-time uniform circuit of width $\mathit{poly}\log n$ and length n^α then $\alpha \geq \sqrt{2} - \epsilon$ for any $\epsilon > 0$:

$$\mathcal{NSIZE}_U(n) \nsubseteq \mathcal{SIZEWD}_U(n^{\sqrt{2}-\epsilon}, \mathit{poly}\log n) \tag{2.24}$$

To prove an equivalent result as in theorem 2.4.1 we would need the following lemma (the equivalent of lemma 2.3.5):

[5]log-time uniformity is assumed as discussed in the previous sections

For any $\epsilon > 0$ and $t = n^c$, $c > 1$ (where n is the size of the input) there is a language that can be computed by non-deterministic circuits of size t but is not computable by any non-deterministic circuit of size $t^{1-\epsilon}$, for any $\epsilon > 0$.

$$\mathcal{NSIZE}_U(t) \not\subseteq \mathcal{NSIZE}_U(t^{1-\epsilon}) \tag{2.25}$$

Theorem 2.5.1 *Assuming that* (2.25) *is true, the class of log-time uniform non-deterministic circuits of size n is not a subset of the class of* log*-time uniform deterministic circuits of width* $poly\log n$ *and length* $n^{\sqrt{2}-\epsilon}$*, for any* $\epsilon > 0$.

Proof We will show how the ideas of the proof of theorem 2.4.1 can be used in the uniform circuit setting to prove the weaker result:

$$\begin{aligned} \mathcal{NSIZE}_U(n) \subseteq \mathcal{SIZEWD}_U(n^\alpha, poly\log n) \implies \\ \alpha \geq \sqrt{2} - \epsilon \end{aligned} \tag{2.26}$$

for any $\epsilon > 0$.

Let $\mathcal{NSIZE}_U(n) \subseteq \mathcal{SIZEWD}(n^\alpha, poly\log n)$. For any non-deterministic circuit C_1 of size t, there exists a circuit C_2 in $\mathcal{SIZEWD}_U(t^\alpha, poly\log t)$ that simulates C_1 (accepts the same language as C_1). C_2 has width $poly\log t$ and length t^α. This corresponds to $poly\log t$ space bounded computation of length t^α described in theorem 2.4.1. Assume that the gates of C_2 are arranged in levels (there at most t^α such levels, and each level consists of at most $poly\log t$ nodes). We will define a non-deterministic circuit C_3 that simulates C_2. C_3 proceeds as follows:

1. Partition C_2 into blocks, such that there are $t^{\alpha/2}$ levels in each block. Guess the states $A_1, \ldots, A_{t^{\alpha/2}}$ of C_2 after each block of computation (between A_i and A_{i+1} there are $t^{\alpha/2}$ levels of C_2).

2. Check that the sequence $A_1, \ldots, A_{t^{\alpha/2}}$ is correct: for all i: $A_i \vdash^*_{C_2} A_{i+1}$. This task can be done by choosing universally an index i and checking whether $A_i \vdash^*_{C_2} A_{i+1}$ by simulating C_2 for $t^{\alpha/2}$ steps (levels).

Note that in the second step, we need a non-deterministic sub-circuit of size $t^{\alpha/2} poly\log t$ whose input has size $t^{\alpha/2} poly\log t + n$. The simulation is made possible by our strong uniformity assumptions. Since $\mathcal{NSIZE}_U(t^\alpha) \subseteq \mathcal{SIZEWD}_U(t^{\alpha^2/2}, poly\log t)$, it follows that C_3 is a non-deterministic circuit of size $t^{(\alpha+\epsilon)^2/2}$ for any $\epsilon > 0$, and therefore:

$$\mathcal{NSIZE}_U(t) \subseteq \mathcal{NSIZE}_U(t^{(\alpha+\epsilon)^2/2}) \tag{2.27}$$

Assumption (2.25) implies that $\alpha > \sqrt{2} - \epsilon$. ∎

2.6 Discussion

In this work we have proven a separation between non-deterministic time n and deterministic time $n^{\sqrt{2}-\epsilon}$ in poly-logarithmic space. The proof uses mostly known and in some sense elementary techniques. For example, Kannan in [Kan84] proceeds almost the same way to prove his result; Fortnow also uses about the same techniques for simulating non-deterministic machines by alternations, and for simulating alternating Turing machines by non-deterministic machines. We also presented a similar (conditional) result for log-space uniform circuits, using the same techniques and ideas.

It would be interesting to see if the time bound can be improved using the same techniques; Fortnow and van Melkebeek have improved the bound to the golden ratio. The quadratic bound still seems beyond our reach. Proving a non-uniform result is probably the most challenging task. It seems that the techniques used in this paper do not generalize in the non-uniform setting. The uniform result was shown for poly-logarithmic width circuits. A more natural and interesting result would be to prove a lower bound for the size of poly-logarithmic depth circuits.

Chapter 3

Finite State Automata and Complexity Theory

We consider [KLV00] (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 $\mathcal{NP}$). 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.

3.1 Introduction

Separating complexity classes is a major problem in complexity theory. There are only a few unconditional results and many open questions. Another major open problem is to give an explicit hard function for the circuit and Turing machine models. In this work, 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 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 $\mathcal{NL} \neq \mathcal{P}$. On the other hand, we will also prove that if the finite state automata intersection emptiness problem has indeed fast algorithms (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:

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

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: $L(F) \neq \varnothing$. The size of F is $O(\sigma^k)$.

Let $\mathcal{F}$ denote the assumption that there is a better algorithm for checking the

[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.

intersection emptiness problem for a collection of a fixed number k of automata, namely:

Assumption $\mathcal{F}$: *Let $F_1, F_2, \ldots, F_k$ be k FSA's of size σ. There is a deterministic algorithm that can decide whether*

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

in time $\sigma^{\frac{k}{f(k)}+d}$, 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 $O(n^{O(1)} \cdot 2^{\epsilon n})$ for all $\epsilon > 0$
2. Integer factorization can also be solved in $O(n^{O(1)} \cdot 2^{\epsilon n})$ for all $\epsilon > 0$
3. $\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(2^{\epsilon t})$ for all $\epsilon > 0$

A slight modification of assumption $\mathcal{F}$ also allows us to separate $\mathcal{NL}$ from $\mathcal{P}$.

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 FSA's 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. 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 (the size of the automata) is much harder, known to be $\mathcal{PSPACE}$-complete [Koz77]. If k is a constant then the problem has a polynomial time algorithm as described above.

A similar result was given by Feige and Kilian [FK97]. 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 (small, $\log n$ size cliques) remains an open problem. Feige and Kilian [FK97] prove that if this problem is solvable in polynomial time then there is a subexponential simulation of non-deterministic computations:

$$\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(\tau^{\sqrt{\tau}}) \tag{3.1}$$

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* [DF92, DF99]. 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, Pippenger, Szemeredi and Trotter [PPST83]. 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 section.)

3.2 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.

3.2.1 Subset sum

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

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

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 $2^{n/3} n^{O(1)}$. By choosing a collection of k automata, the resulting algorithm runs in $2^{\epsilon n} n^{O(1)}$ for any $\epsilon > 0$.

Theorem 3.2.1 *Assumption $\mathcal{F}$ implies that there is an algorithm solving subset sum in $O(2^{\epsilon \cdot n} \cdot n^{O(1)})$ for all $\epsilon > 0$*

Proof Pick two primes[2] p, q of size $n/3$ and build two machines M_p and M_q testing $\sum_{i=1}^{n} a_i x_i \equiv b(mod\ p)$ and $\sum_{i=1}^{n} a_i x_i \equiv b(mod\ q)$. The size (number of states) of these two machines is $|M_p| = O(p \cdot n^{O(1)})$ (same for M_q), where $p < 2^{n/3}$. Consider the intersection $L(M_p) \cap L(M_q)$. If there is a solution to the given knapsack problem then this intersection is nonempty, since $\sum_{i=1}^{n} a_i x_i \equiv b(mod\ p)$ modulo any number p. If, on the other hand, $\sum_{i=1}^{n} a_i x_i \equiv b$ modulo both primes, then $\sum_{i=1}^{n} a_i x_i \equiv b$ or

[2] p and q only need to be relatively prime

$b - \sum_{i=1}^{n} a_i x_i$ is a multiple of p and q. Define the set S to be the set of exceptions: all those numbers of the form $b - \sum_{i=1}^{n} a_i x_i$ (that correspond to different values of the input, binary vector x, and for which $b - \sum_{i=1}^{n} a_i x_i \neq 0$ but the sum is zero modulo both primes. The set S is explicit (multiples of p, q) and we can construct a tree-like acceptor. The size of S is less than $2^{n/3}$ since the number of vectors x which make the quantity $b - \sum_{i=1}^{n} a_i x_i$ equal to zero modulo both p and q is:

$$2^n \cdot \frac{1}{2^{n/3}} \cdot \frac{1}{2^{n/3}} = 2^{n/3}$$

Therefore the size of the automaton for S is at most $O(2^{n/3})$. Now Consider the following intersection problem:

$$L(M_p) \cap L(M_q) \cap \bar{S} \overset{?}{\neq} \varnothing \tag{3.2}$$

If this intersection is non-empty, then there exists a solution to the given knapsack problem. The FSA for S is acyclic (tree-like). We can compute the intersection of such an acceptor with the automaton M_q without computing the cartesian product of the two machines. We can combine S and any FSA M into a new machine with $|S| + |M|$ states that accepts $L(\bar{S}) \cap L(M)$.

In order to get the desired bound ($2^{\epsilon n}$ for any positive ϵ), use a similar construction for k automata: pick k primes $p_1, \ldots, p_k$ of size $\frac{n}{k+1}$ and follow the same ideas described above to construct k automata M_{p_k} that check if $b - \Sigma_{i=1}^{n} \equiv 0(mod\ p_k)$. Construct the tree-like acceptor S as before, and consider the emptiness problem for the intersection

$$\bigcap_{i=1}^{k} M_{p_i} \cap \bar{S} \tag{3.3}$$

The size of the automata M_{p_k} is at most $\sigma = O(2^{\frac{n}{k+1}})$. S has also size σ, and will be combined in an automaton for $M_{p_k} \cap S$ of size $O(\sigma)$. 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 $O(\sigma^{\frac{k}{f(k)}+d})$. This will give us the following upper bound:

$$\exp\left(\frac{k}{k+1}\frac{1}{f(k)}n + \frac{d}{k+1}n\right) \tag{3.4}$$

For large enough k the above expression becomes

$$2^{\frac{1}{f(k)}n} \tag{3.5}$$

(assuming that $f(k)$ grows slower than $O(k)$.) Since $f(k)$ is unbounded, $\frac{1}{f(k)}$ can become less than any constant $\epsilon > 0$ by choosing k appropriately. ▮

3.2.2 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 $O(2^{\epsilon n})$, for any $\epsilon > 0$, 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 $2^{\frac{1}{4}n}$. With the Extended Riemann

Hypothesis this bound only improves to $2^{\frac{1}{5}n}$ (See [Bac90]).

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

Theorem 3.2.2 *The assumption $\mathcal{F}$ implies that factoring can be solved in time $O(n^{O(1)}2^{\epsilon n})$ for any $\epsilon > 0$.*

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 $n/3$ and consider the corresponding FSA's M_p, M_q, checking whether $x \cdot y \equiv z (mod\ p)$ and $x \cdot y \equiv z (mod\ q)$ respectively. Again we build the set S of those strings x, y for which $x \cdot y \equiv z$ modulo both primes p and q but $x \cdot y \neq z$. The size of S is $|S| \leq 2^{n/3}$, and since S is explicit we can construct a tree-like FSA acceptor. The following emptiness problem solves the factoring problem:

$$L(M_p) \cap L(M_q) \cap \bar{S} \overset{?}{\neq} \varnothing \tag{3.6}$$

The input of the finite state machines is the string $x\#y$. 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. Since the length of the 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 $M_q = q \cdot n^{O(1)}$). Based on the assumption $\mathcal{F}$, we can check the intersection (3.6) for emptiness in $2^{n/3} \cdot n^{O(1)}$.

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

$$\bigcap_{i=1}^{k} M_{p_i} \cap \bar{S} \tag{3.7}$$

where S is the set of all numbers x, y such that $x \cdot y \equiv z(mod\ p_i)$ for $1 \leq i \leq k$, but $x \cdot y \neq z$. S is a tree acceptor FSA, and can be combined with M_{p_k} into one FSA of size $2^{\frac{n}{k+1}}$ as discussed above.

By our assumption $\mathcal{F}$ the intersection from equation (3.7) can be solved in time $\sigma^{\frac{k}{f(k)}+d}$. This yields the following upper bound:

$$n^{O(1)} \cdot \exp\left(\frac{k}{k+1}\frac{1}{f(k)}n + \frac{d}{k+1}n\right) \tag{3.8}$$

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 $O(n/2)$ times, we can find all the bits of the number x. ∎

3.3 Deterministic simulation of non-deterministic computation

In this section we show how to build a collection of automata to check 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 be “subexponential”.

The main theorem is proved for multi-tape Turing machines. 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 nondeterministic choice of M at this step. This description of the computation of a Turing machine M on a certain input is also refered 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 computation (Turing machine) 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 means 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 certain (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.

The notion of *block respecting* computation was introduced by Hopcroft Paul and Valiant in [HPV77] to prove that deterministic space is strictly more powerful than deterministic time ($\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t/\log t)$). Block respecting Turing machines are also used in [PPST83] to prove that non-deterministic linear time is more powerful than deterministic linear time (see also [PR81] for a generalization of the results from [HPV77] for RAMs and other machine models).

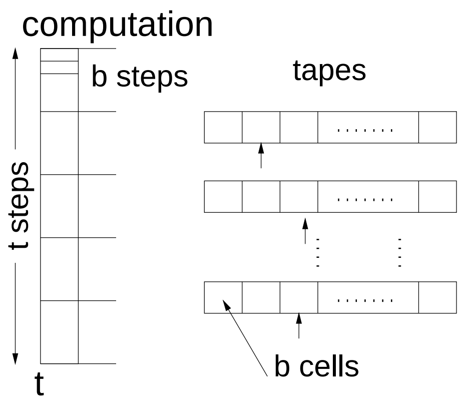


Figure 3.1: Block respecting computation

Definition 3.3.1 *Let M be a machine running in time $f(n)$, where n is the length of its input x. Let the computation of M be partitioned in $a(n)$* segments*, where each segment consists of $b(n)$ consecutive steps ($a(n) \cdot b(n) = f(n)$). 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.*

Here are some more details for the ideas behind the proof. Let M be a nondeterministic Turing machine running in time t. On input x, convert M to a block respecting machine M_b. Consider the trace of the computation of M_b and construct a collection of FSA's 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 (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.

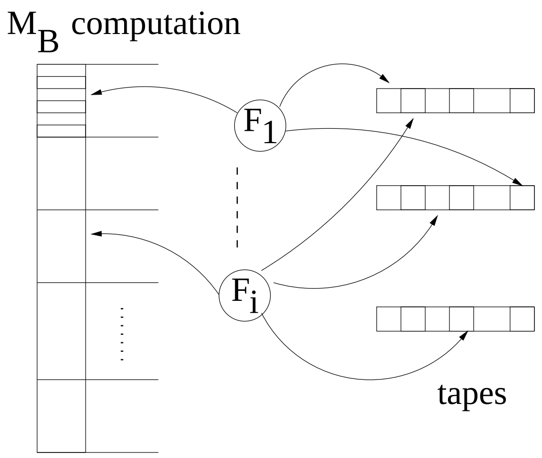


Figure 3.2: Checking block respecting computation with finite state automata

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 nondeterministic, 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 [HPV77]. For the deterministic simulation we need to consider all possible graphs.

Theorem 3.3.2 *Assumption $\mathcal{F}$ implies $\mathcal{NTIME}(t) \subseteq \mathcal{DTIME}(2^{\epsilon t})$, for any $\epsilon > 0$.*

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 $O(t)$. Break the computation of M_B (on input x) 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 [HPV77]: 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 i-th 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 $1 \leq i \leq l$, $v(\Delta_i) \to v(\Delta)$. The number of edges can be at most $O((l+1)\frac{t}{B})$ and therefore the number of bits required to describe the graph is[3] $O\left((l+1)\frac{t}{B} \log \frac{t}{B}\right)$.

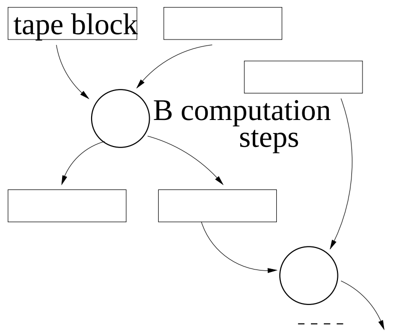


Figure 3.3: Graph description of a block-respecting computation

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

[3]Since l is a constant, from now on it will be incorporated in the big-O notation.

a segment of the computation). Since the machine M is non-deterministic, we need to consider all $2^{O\left(\frac{t}{B}\log\frac{t}{B}\right)}$ 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.

In order to check if the computation is correct during one step, we could use an FSA of constant size (the size 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 FSA's. Then for any $\epsilon > 0$ we can pick $k = 1/\epsilon$ such that each automaton checks $\frac{1}{k}\frac{t}{B}) = \epsilon t/B$ segments or $\frac{\epsilon t}{B}B = \epsilon t$ steps. The size of each FSA is therefore $2^{\epsilon t}$ (decision tree).

Our deterministic algorithm that will simulate M must construct the transition diagrams for these k FSA's. For each transition (arc in the decision tree) we need to simulate M_b for at most 2^B steps. Since $2^{\epsilon t}$ is the total number of transitions, the total time required is $2^{\epsilon t} \cdot 2^B$. The running time for the construction of all the FSA's for all possible graphs is therefore:

$$2^{\epsilon t}2^B 2^{O(\frac{t}{B}\log\frac{t}{B})} = 2^{O(\epsilon t)} \tag{3.9}$$

In order to check if there exists an accepting computation for M on input x it suffices to check if the intersection of all FSA's $\cap_{i=1}^{k} F_i$ is non-empty. Under our assumption $\mathcal{F}$, the time needed to intersect k FSA's of size $\sigma = 2^{\epsilon t} = 2^{t/k}$ is:

$$\begin{aligned}\sigma^{\frac{k}{f(k)}+d} &= \left(2^{t/k}\right)^{\frac{k}{f(k)}+d} \\ &= 2^{\frac{1}{f(k)}t+\frac{d}{k}t}\end{aligned} \tag{3.10}$$

Since $f(k) = o(k)$, the time needed for testing the intersection for emptiness is $2^{O(\frac{1}{f(k)}t)}$. Therefore the total time for our simulation is the time to construct the FSA's plus the time to check the intersection: $2^{O(\epsilon t)} + 2^{O(\frac{1}{f(k)}t)}$. Since $f(k)$ is unbounded we can always pick k appropriately so that the total time is $2^{O(\epsilon t)}$. ▮

3.4 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 actually 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 "algorithms" rather than lower bounds. 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 actually "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 [Kan81].

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

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

$$\bigcap_{i=1}^{k} L(F_i) \cap G \overset{?}{\neq} \varnothing$$

in time $\sigma^{\frac{k}{f(k)}+d}\sigma'$, 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 FSA's. 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 will 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 $\mathcal{F}_\mathcal{C}$.

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

$$\bigcap_{i=1}^{k} L(F_i) \cap G \overset{?}{\neq} \varnothing$$

with size $\sigma^{\frac{k}{f(k)}+d}\sigma'$, 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.

2. Given the $\mathcal{NL}$ machine, and the input x, we will check if there exist any accepting computations on input x.

3. "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)

4. 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 (ignore the rest of the computation string)

5. 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"

6. 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 (last two steps).

For the uniform setting, this simulation will give a (better than) n^2 time simulation of $\mathcal{NL}$. Then the deterministic time-hierarchy results will complete the separation proof of $\mathcal{NL}$ and $\mathcal{P}$.

Theorem 3.4.1 (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 n^2. To separate $\mathcal{NL}$ from $\mathcal{NP}$, the following result by Kannan [Kan81] will be used:

Theorem 3.4.2 (Kannan [Kan81]) *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 restated as a general lemma (see section 3.6) that can be used to separate complexity classes by designing fast algorithms and/or simulations rather than proving lower bounds.

3.4.1 Uniform assumption: $\mathcal{NL}$ vs $\mathcal{P}$

In the uniform setting, we will 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 3.4.3 *Assumption $\mathcal{F}'$ implies $\mathcal{NL} \subseteq \mathcal{DTIME}(n^{1+\epsilon})$, for any $\epsilon > 0$.*

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

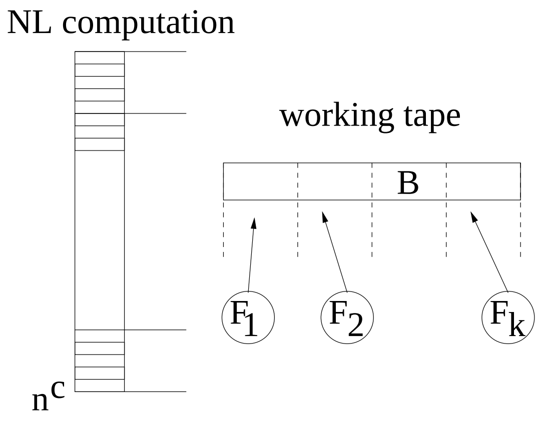


Figure 3.4: Checking non-deterministic Logspace computation

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 FSA's (i.e. 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 FSA's cannot check whether the input (on the read-only input tape) appears correctly throughout the computation string (it 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 nondeterministic choice of $M_L(x)$ at this step and the symbol read/written on the working tape.

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 $k = \frac{c \log n}{B}$. 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 FSA's 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 transitions of F_i (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 while the number of transitions is at most $O(\text{number of states of } F_i) = 2^{O(B)}$. Hence the time needed to construct the FSA's 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 $\cap_{i=1}^{k} L(F_i) \cap L(G)$ 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

$$\begin{aligned} |F_i|^{\frac{k}{f(k)}+d}|G| &= d_1 2^{d_2(\frac{kB}{f(k)}+B)} n \\ &= d_1 2^{\frac{d_2 c \log n}{f(k)}+\frac{d_2 c \log n}{k}} n \end{aligned} \tag{3.11}$$

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

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

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

From the well known time hierarchy theorem 3.4.1 we get the following:

Corollary 3.4.4 *Assumption $\mathcal{F}'$ implies $\mathcal{NL} \neq \mathcal{P}$*

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

The non-uniform version of our assumption $\mathcal{F}_\mathcal{C}$ implies that $\mathcal{NL} \neq \mathcal{NP}$. We will start by showing that $\mathcal{NL}$ has small (fixed polynomial size) circuits ($\mathcal{NL}$ 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 good. This will be obvious in the proof, where Kannan's [Kan81] result is used.

Theorem 3.5.1 *Assumption $\mathcal{F}_\mathcal{C}$ implies that $\mathcal{NL}$ can be simulated by fixed polynomial size (size n^2) (non-uniform) circuits.*

Proof The proof is essentially the same as for theorem 3.4.3. Each of the automata F_i, $i = 1 \dots k$, 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 $\mathcal{F}_\mathcal{C}$, there is a size $|F_i|^{\frac{k}{f(k)}+d}|G| = n^{O(\frac{c}{f(k)}+\frac{c}{k})}$ circuit that given the description of automata F_i, $i = 1 \dots k$, and G from theorem 3.4.3 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 3.5.2 *Assumption $\mathcal{F}_\mathcal{C}$ 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 3.4.2 implies that for any constant β there is a language in Σ_2^p and therefore $\mathcal{NL}$ that is not computable by circuits of size n^β.
3. By theorem 3.5.1 $\mathcal{NL}$ has fixed polynomial (n^2) size circuits. Contradiction

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

Note that the proof of corollary 3.5.2 can be considered as an application of lemma 3.5.3 (presented in the next section).

3.5.1 A general Lemma

The following lemma is a general way to state Kannan's [Kan81] result, 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 3.5.3 *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 $>> n^k$.

3. for some fixed k, $\mathcal{C}_1$ has circuits of size $\leq n^k$ 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, and therefore all $\mathcal{C}_2$ has fixed polynomial size circuits. But this contradicts 2. ▮

3.6 Remarks

The results mentioned in this section can be viewed as a method of separating $\mathcal{NL}$ from $\mathcal{P}$ or $\mathcal{NP}$ (See Fortnow [For00a] for 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 section. It would also be interesting to see if there are other connections between similar problems and Complexity theory questions.

Chapter 4

Non-Uniform depth of Polynomial Time

We discuss some connections between polynomial time and non-uniform, small depth circuits. A connection is shown with simulating deterministic time in small space. The well known result of Hopcroft, Paul and Valiant [HPV77] showing that space is more powerful than time can be improved, by making a (strong) assumption about the connection of (linear) deterministic time computations and polynomial size, non-uniform, small (sub-linear) depth circuits. To be more precise, in this section we will prove the following: If every polynomial time computation can be done by a non-uniform circuit of polynomial size and sub-linear depth (for example if $\mathcal{P} \subseteq \mathcal{NC}/poly$), then $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c})$ for some constant c.

4.1 Introduction

Hopcroft Paul and Valiant [HPV77] proved in 1977 that space is more powerful than time: $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t/\log t)$. The proof of this trade-off result is based on pebbling techniques and the notion of *block respecting* computation. Improving the space simulation of deterministic time has been a long standing open problem. Paul Tarjan and Celoni [PTC77] proved an $n \log n$ lower bound for pebbling a certain family of graphs. This lower bound implies that the trade-off result $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t/\log t)$ of [HPV77] cannot be improved using pebbling arguments.

In this work we will present a connection between space simulations of deterministic time and the depth of non-uniform circuits simulating polynomial time computations. If every problem in $\mathcal{P}$ can be solved by a polynomial size non-uniform circuit of small (sub-linear) depth then every deterministic computation of time t can be simulated in space t^{1-c} for some constant c (that depends only on our assumption about the non-uniform depth of $\mathcal{P}$):

$$\begin{aligned} \mathcal{DTIME}(n) \subseteq & \mathcal{SIZE}\text{–}\mathcal{DEPTH}(poly(n), n^{1-\epsilon}) \\ & \Longrightarrow \mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c}) \end{aligned} \tag{4.1}$$

A similar result was shown by Sipser [Sip86, Sip88] from the point of view of reducing randomness required for randomized algorithms. His result considers the problem of constructing expanders with certain properties. Assuming that those expanders can be constructed efficiently, the main theorem proved is that if $\mathcal{P}$ is equal to $\mathcal{R}$ then the space simulation of Hopcroft, Paul and Valiant [HPV77] can be improved for a more restricted case:

Theorem 4.1.1 (Sipser [Sip86]) *Under the assumption that certain expanders have explicit constructions, there exists an $\epsilon > 0$ such that*

$$P = R \implies (\mathcal{DTIME}(t) \cap 1^*) \subseteq \mathcal{SPACE}(t^{1-\epsilon}) \tag{4.2}$$

The above results shows how to simulate unary languages in $\mathcal{DTIME}(t)$.

4.2 Notation - Definitions

We will use the standard notation for time and space complexity classes $\mathcal{DTIME}(t)$ and $\mathcal{SPACE}(t)$. $\mathcal{SIZE}$–$\mathcal{DEPTH}(s,d)$ will denote the class of non-uniform circuits withs size (number of gates) $O(s)$ and depth $O(d)$. At some points in this section, we will also avoid writing poly-logarithmic factors in detail and use the notation $\tilde{O}(n)$ to denote $O(n \log^k n)$ for constant k. In this chapter we will consider time complexity functions that are time constructible: A function $t(n)$ is called fully time constructible if there exists a deterministic Turing Machine that on input of length n halts after exactly $t(n)$ steps. In general a function $f(n)$ is t-time constructible, if there is a deterministic Turing Machine that on input x outputs $1^{f(|x|)}$ and runs in time $O(t)$. t, s time-space constructible functions are defined similarly.

For the proof of the main result we will use the notion of block respecting Turing machines introduced by roft Paul and Valiant in [HPV77]. Recall that block respectin Turing machine computation is defined as follows (definition 3.3.1):

Let M be a machine running in time $f(n)$, where n is the length of its input x. Let the computation of M be partitioned in $a(n)$ *segments*, where each segment consists of $b(n)$ consecutive steps ($a(n) \cdot b(n) = f(n)$). 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.

Recall also that block respecting Turing machines are also used in [PPST83] to prove that non-deterministic linear time is more powerful than deterministic linear time (see also [PR81] for a generalization of the results from [HPV77] for RAMs and other machine models).

4.3 Main Results

We show that if polynomial time has small non-uniform circuit depth (for polynomial size circuits) then $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c})$ for some $c > 0$.

The strongest form of the main result is the following: if (deterministic) linear time has polynomial size, non-uniform circuits of sublinear depth, then $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c})$ (for some small $\epsilon > 0$):

$$\mathcal{DTIME}(n) \subseteq \mathcal{SIZE\text{–}DEPTH}(poly, n^{\epsilon}) \implies \mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c}) \quad (4.3)$$

It is easy to see though, that by padding arguments, the assumption mentioned above, implies that $\mathcal{P} \subseteq \mathcal{SIZE\text{–}DEPTH}(poly, n^{o(1)})$

The main idea is the following: Start with a deterministic Turing machine M running in time t and convert it in a block respecting machine M_B with block size B. In each segment of the computation, M_B reads and/or writes in exactly one block on each tape. We will argue that we can check the computation in each such segment with the *same* sub-circuit and using an assumption of the form $\mathcal{P} \subseteq \mathcal{NC}/poly$ we can actually construct this sub-circuit in polynomial size and small (poly-logarithmic) depth. Combining all these sub-circuits together we can build a larger circuit that will check the entire computation of M_B in small depth. The final step is a technical lemma that shows how to evaluate this circuit in small space (equal to its depth).

Theorem 4.3.1 *Let t be a reasonable time complexity function. If $\mathcal{P} \subseteq \mathcal{NC}/poly$ then $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^{1-c})$ for some constant c.*

Proof

Consider any Turing Machine M running in deterministic time t. We will show how to simulate M in small space using the assumption that polynomial time has shallow (poly-logarithmic depth) polynomial size circuits. Here is a brief outline:

1. convert given TM in a block respecting machine with block size B.

2. construct the graph that describes the computation. Each vertex corresponds to a computation segment of B steps.

3. The computation on each vertex can be checked by the *same* TM U that runs in polynomial time (linear time)

4. Since $\mathcal{P} \subseteq \mathcal{NC}/poly$, there is a circuit U_C that can replace U. U_C has polynomial size and polylogarithmic depth.

5. Construct U_C by trying all possible circuits.

6. Plug in the sub-circuit U_C to the entire graph. This graph is the description of a circuit of small depth, that corresponds to the computation of the given TM. Evaluate the circuit (in small space)

In more detail: Convert M to a block respecting machine M_B. Break the computation of M_B (on input x) in segments of size B each; the number of segments is t/B. Consider the directed graph G corresponding to the computation of the block respecting machine as described in [HPV77]: 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 $v(\Delta)$ denotes the vertex corresponding to time segment Δ and Δ_i the last time segment before Δ during which the i-th 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 $1 \leq i \leq l$,

$v(\Delta_i) \to v(\Delta)$. The number of edges can be at most $O(\frac{t}{B})$ and therefore the number of bits required to describe the graph is $O\left(\frac{t}{B}\log\frac{t}{B}\right)$.

Each vertex of this graph corresponds to B computation steps of M_B. During this computation, M_B reads and writes only in one block from each tape. In order to check the computation that corresponds to a vertex of this graph, we would need to simulate M_B for B steps and check $O(B)$ bits from M_B's tapes. For each vertex we need to check/simulate a different segment of M_B's computation: this can be done by a Turing machine that will check the corresponding computation of M_B. It is easy to see that the Turing machine that checks the computation on each vertex of the graph is the same; the machine needs to simulate M_B for B steps and check the content of specific tape blocks. The only thing that is different for each vertex is which blocks on the working tapes we need to check and which part of the computation we need to simulate (starting state of M_B. Therefore we can assume that the exact same machine is used on all vertices to check the computation of M_B, with an extra input: this extra input is an index that shows which segment of the computation of M_B must be simulated (index of the vertex for example). Therefore we have the same ("universal") machine U on all vertices of the graph which runs in deterministic polynomial time.

If $\mathcal{P} \subseteq \mathcal{SIZE}\text{–}\mathcal{DEPTH}(n^k, \log^l n)$ then U can be simulated by a circuit U_C of size $O(B^k)$ and small depth $O(\log^l B)$, for some k, l. The same circuit is used on all vertices of the graph. In order to construct this circuit, we can try all possible circuits and simulate them on all possible inputs. This requires exponential time, but only small amount of space: the size of the circuit is B^k and its depth polylogarithmic in B. We need $\tilde{O}(B^k)$ bits to write down the circuit and only polylog space to evaluate it (using lemma 4.3.2).

Once we have constructed U_C, we can build the entire circuit that will simulate

M_B. This circuit derives directly from the (block-respecting) computation graph where each vertex is an instance of the sub-circuit U_C. The size of the entire circuit is too big to write down. We have up to t/B sub-circuits (U_C) that would require a size of $\tilde{O}(\frac{t}{B}B^k)$ for some constant k. But since it is the same sub-circuit U_C that appears throughout the graph, we can implicitly describe the entire circuit in much less space. For the evaluation of the circuit, we only need to be able to describe the exact position of a vertex in the graph, and determine the immediate neighbors of a given vertex (previous and next vertices). This can easily be done in space $\tilde{O}(t/B + B^k)$.

In order to complete the simulation we need to show how to evaluate a small-depth circuit in small space (see Borodin [Bor77]).

Lemma 4.3.2 *Consider a directed acyclic graph G with one source (root). Assume that the leaves are labeled from $\{0,1\}$, its inner nodes are either AND or OR nodes and the depth is at most d. Then we can evaluate the graph in space at most $O(d)$.*

Proof (of lemma. See [Bor77] for more details).

Convert the graph to a tree (by making copies of the nodes). The tree will have much bigger size but the depth will remain the same. We can prove (by induction) that the value of the tree is the same as the value of the graph from which we started. Evaluating the tree corresponds to computing the value of its root. In order to find the value of any node v in the tree, proceed as follows: Let $u_1, \ldots, u_k$ denote the child-nodes of v.

If v is an AND node, then compute (recursively) the value of its first child u_1. If $value(u_1) = 0$ then the value of v is also 0. Otherwise continue with the next child. If the last child has value 1 then the value of v is 1. Note that we do not need to remember the value of the child-nodes that we have evaluated. If v is an OR node, the same idea can be applied. We can use a stack for the evaluation of the tree. It is easy to see that the size of the stack will be at most $O(d)$, that is as big as the depth of the tree. ∎

$$B^k + \frac{t}{B}\log^l B \tag{4.4}$$

To get the desired result, we need to choose the size B of the blocks appropriately to balance the two terms in (4.4). B will be $t^{1/c}$ for some constant c that is larger than k.

∎

As mentioned above, the exact same proof would work even if we allow almost linear depth for the non-uniform circuits for $\mathcal{P}$. The stronger theorem is the following:

Theorem 4.3.3 *If* $\mathcal{DTIME}(n) \subseteq \mathcal{SIZE}\text{–}\mathcal{DEPTH}(n^k, o(n))$ *for some* $k > 0$ *then* $\mathcal{DTIME}(t) \subseteq \mathcal{SPACE}(t^c)$ *for some constant* $c < 1$.

These proof ideas seem to fail if we try to simulate non-deterministic time in small space. In that case, evaluating the circuit would be more complicated: we would need to use more space in order to make sure that the non-deterministic guesses are consistent throughout the evaluation of the circuit.

4.4 Semi-unbounded circuits

These simulation ideas using block respecting computation can also be used to prove an *unconditional* result relating uniform polynomial time and non-uniform circuits. The simulation of the previous section implies unconditionally a (weak) trade-off type of result for the size and depth of a non-uniform circuit that simulates a uniform computation. The next theorem proves that any deterministic time t computation can be simulated by a non-uniform circuit of size $\sqrt{t} \cdot 2^{\sqrt{t}}$ and depth $\sqrt{t}$, which has "semi-unbounded" fan-in. It follows that deterministic time t can be simulated by non-uniform circuits that can be described in size $2^{\sqrt{t}}$ and have depth $\sqrt{t}$. This description is possible since the same subcircuit is used many times in the construction described in the following theorem:

Theorem 4.4.1 *Let t be a reasonable time complexity function. Then $\mathcal{DTIME}(t) \subseteq \mathcal{SIZE}\text{–}\mathcal{DEPTH}(\sqrt{t} \cdot 2^{\sqrt{t}}, \sqrt{t})$, and the simulating circuits requires exponential fan-in for AND gates and polynomial for OR gates (or vice-versa)*

Proof

Given a Turing machine running in $\mathcal{DTIME}(t)$, construct the block respecting version, and repeat the exact same construction as the one presented in the proof of theorem 4.3.1: Construct the graph describing the block respecting computation, which has t/B nodes, and every node corresponds to a segment of B (we will chose the size B later in the proof) computation steps. Use this graph to construct the non-uniform circuit: For every node, build a circuit, say in DNF, that corresponds to the computation that takes place on that node. This circuit has size exponential in B in the worst case, $2^{O(B)}$, and depth 2. The entire graph describes a circuit of size $\frac{t}{B} 2^{O(B)}$ and depth $O(B)$. Also, note that for every sub-circuit that corresponds to each node, the input gates (AND gates as described in the proof) have a fan-in of at most $O(B)$, while the second level might need exponential fan-in. This construction yields a circuit of "semi-unbounded" fan-in type. ▮

Chapter 5

Tautologies and Propositional Proof Systems

One way to address the $\mathcal{NP} \overset{?}{=} co\text{-}\mathcal{NP}$ question is to consider the length of proofs of tautologies in various proof systems. A well known theorem [CR79] states that if there exists a *propositional proof system* (with certain properties) in which every tautology has polynomial size proofs, then $\mathcal{NP} = co\text{-}\mathcal{NP}$. This is a very interesting connection between Propositional logic and fundamental questions in Complexity theory. Many different proof systems have been considered and strong lower bounds have been shown for different tautologies. A natural question to ask is the following: is there a direct correspondence between a propositional proof system and a computation model? We know for example, that the proof system of regular resolution corresponds to read-once branching programs in a very natural way; the size of the smallest proof of a tautology in regular resolution is the same as the size of the smallest read-once branching program that solves a search problem associated with the tautology.

In this section we will try to look closer to the correspondence between computation models and propositional proof systems. We consider proof systems defined by appropriate classes of automata [LV99b]: In general, starting from a given class of automata we can define a corresponding proof system in a natural way. An interesting new proof system that we consider is based on the class of push down automata. We present an exponential lower bound for oblivious read-once branching programs which implies that the new proof system based on push down automata is strictly more powerful than oblivious regular resolution. This result gives a characterization of this new system and its power, compared to classic systems from propositional logic.

5.1 Introduction

One of the famous open questions of complexity theory is whether $\mathcal{NP}$ is equal to *co-*$\mathcal{NP}$. Cook and Reckhow [CR79] showed that this question translates in a very basic and natural in terms of propositional logic: $\mathcal{P}$ equals *co-*$\mathcal{NP}$ if and only if there exists a propositional proof system in which every tautology has short (polynomial size) proofs. This question, in its general form seems to be completely beyond our reach. However, recently there has been considerable progress on attacking a restricted version of this problem.

In our case we plan to look more closely at this correspondence between computation models (automata classes) and propositional proof systems. It is known, for example, that the proof system of regular resolution corresponds directly to the class of read-once branching programs. A lower bound on the size of regular resolution proofs for a language $\mathcal{L}$ that is defined by a tautology, translates directly to a lower bound for the size of read-once branching programs solving a search problem associated with $\mathcal{L}$. Thus, the known technology for proving lower bounds on read-once branching programs can be used to prove lower bounds for resolution.

We will mostly try to look at the "reverse" correspondence: computation model $\rightarrow$ proof system. Starting from a class of automata (with certain properties) we will show how to associate with it a proof system, in a "natural" way. One important point is that we have to *restrict* the automata class so that the proof system lies in $\mathcal{NP}$. This correspondence will allow us to compare the power of automata classes (computation models) with classic propositional proof systems. It also provides a way to try to design proof systems with the power to prove certain tautologies, and translate ideas from computation models into the context of propositional logic.

An interesting new proof systems that we get in this way is based on the class of push down automata (PDA's). Our correspondence shows that there is a proof system that is in $\mathcal{NP}$ that corresponds to PDA. We can show that, in a certain sense, it is more powerful than oblivious regular resolution. This seems to be expected since regular resolution corresponds to essentially finite state automata.

As we stated earlier the language that is used must be a language that encodes a tautology. In our case this will require a somewhat unusual language, which will be based on the pigeonhole principle. In order to prove our results, we will allow the automata to read their inputs in any order. Thus, the famous example ww^R that is not accepted by a finite automata is accepted if the finite automata can read the inputs in one right order. The latter point is essentially that our automata classes are more like binary decision diagrams (BDD's) than classic automata.

Many questions arise from the definitions and results presented in this section: what does this PDA based proof system mean? Is there some natural generalization of regular resolution that can be seen to be equivalent to our PDA system? A related interesting question is the class of BDD's based on PDA. What is their power? For example, if these PDA systems could compute $x * y = z$ in polynomial bounded size, then we would have interesting consequences for factoring (see section 5.6). It is, of course, known that this is impossible for normal BDD's (Ponzio, [Pon95]). However, the lower bound proof does not seem to generalize to PDA based BDD's: the technique used in [Pon95] is based on a lemma about counting subfunctions for fixed size subsets of input variables which seems to fail for the PDA machines (see section 5.6).

The new PDA based proof system allows us to move beyond read-once machines. For example, simply consider a branching program with a push-down stack, that first reads its input x, pushes all symbols on the stack, and then pops the stack and reads the input again in reverse, x^R. This is equivalent to a *weak read-twice* branching program, that reads xx^R. The PDA in general seems to be more powerful than ordinary branching programs, since it is allowed to read the input in different ways. For example, the PDA can scan the input $(x_1, \ldots, x_n)$ and also compute some "useful" value $\sum_i f(x_i)$ and then use this value to scan $(x_n, \ldots, x_1)$ more wisely. A lower bound for the PDA proof system would be an interesting generalization of existing results.

The definitions for the correspondence between automata models and proof systems as well as the exponential separation between the PDA based proof system and oblivious regular resolution presented in this section, assume that the clauses we consider are "fsa" type formulas: these are simply formulas that can be computed easily by an finite state automaton for any ordering of their variables. The lower bound for oblivious regular resolution is implied by an exponential lower bound for oblivious branching programs, which is proved by applying a technique of Alon and Maass [AM88].

5.2 Preliminaries

The connections between complexity theory and propositional proof systems were first proposed and formalized by Cook and Reckhow [CR73]. A propositional proof system S can be defined[1] as a polynomial time computable predicate S such that, a formula ϕ is a tautology if and only if there exists a "proof" p, for which $S(\phi, p)$ is true. A propositional proof system is usually defined in terms of a set of axioms and a set of inference rules. For a general survey of propositional proof complexity see [BP98, Urq95].

The proof system of resolution is defined ([Bla37, DP60, Rob65]) as a very simple propositional proof system, with no axioms and only one inference rule, the *resolution rule*. In this proof system (see [Kra95]) we consider *clauses* which are defined as sets of literals (disjunctions of literals). A literal is a propositional variable or its negation. If C_1, C_2 are two clauses, and x a variable, then the resolution rule is the following: If x and its negation appear in two clauses, then we can replace those two clauses by their disjunction, after we have removed the occurrences of x and $\neg x$.

$$\frac{C_1 \vee x, C_2 \vee \neg x}{C_1 \vee C_2}$$

In order to prove a tautology, we start from its negation (a set of clauses $C = \{C_1, \ldots, C_k\}$), and apply the resolution rule until we reach the empty clause (contradiction). This will give us a resolution refutation, which is a sequence of clauses, that are produced by the resolution rule, and ending with the empty clause. The refutation can be represented as a (directed acyclic) graph, in which every node

[1] See Beame and Pitassi [BP98], Krajicek [Kra95] and Cook and Reckhow [CR73] for more details on the definition.

corresponds to a clause, and the edges show the applications of the resolution rule.

Regular Resolution is a restricted version of the resolution proof system in which every literal appears at most once in every path in the resolution refutation (in the graph of the refutation).

As mentioned in the introduction regular resolution is known to be equivalent to the computation model of branching programs.

Branching programs are graph representations of discrete functions. Many restricted types of branching programs have been studied, as this type of representation of boolean functions allows us to prove many non trivial lower bounds for various problems.

A branching program B for a function $f : \Sigma_1^n \to \Sigma_2$, where $|\Sigma_1| = \sigma_1$, and $|\Sigma_2| = \sigma_2$ is a directed acyclic graph, with one node of in-degree 0 (source node) and (up to) σ_2 sink nodes. All non-sink nodes are labeled with some variable name from the set of input variables $X = \{x_1, \ldots, x_n\}$ of the function $f = f(x_1, \ldots, x_n)$, and the number of outgoing edges is σ_1. Each edge is labeled by a symbol from the input alphabet Σ_1. The sink nodes are labeled by elements of the output alphabet Σ_2. For each input $c = (c_1, \ldots, c_n) \in \Sigma_1^n$, the output of the function $f(c)$ is given by the label of the sink node we will reach, after following the computation path designated by the input variables: start at the source, and on each node of the branching program labeled by x_i follow the edge labeled c_i. The previous definition describes the *R-way branching programs* proposed by Borodin, Cook [BC82], where $R = \sigma_1$ is the size of the input alphabet.

As a simple example, figure 5.1 shows the branching program representation of the boolean function f, defined as follows: $f(x, y, z) = 1$ iff at least 2 of its inputs $\{x, y, z\}$ are equal to 1.

For our purposes we can assume that the vertices of the branching program are

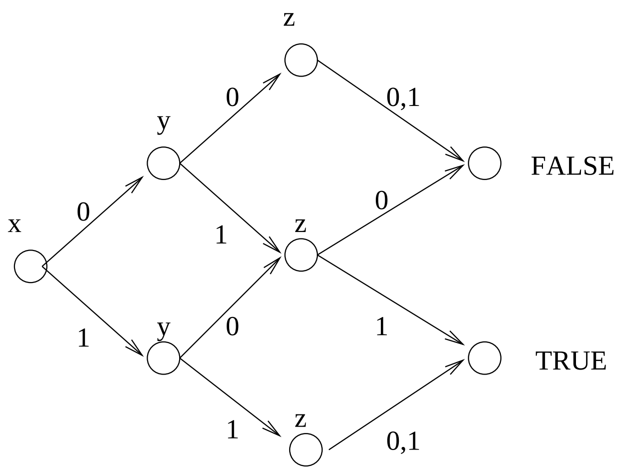


Figure 5.1: Example of a branching program

arranged in levels, such that edges connect nodes in consecutive levels only[2]

The width of the branching program is the maximum number of nodes in one level, and the length is the number of levels (maximum length of a path from the source node to a sink). A branching program is called *read-once* if along every path p every variable is tested at most once. A read-once branching program is also called Binary Decision Diagram (BDD). A read-once branching program is called *uniform* if for any path p starting from the root, the set of variables tested along p depends only on the terminal node u of the path p, and every path from the root to a sink contains all variables. A uniform read-once branching program is called *oblivious* if the variables are tested in the same order along each path. It has been shown that any read-once branching program can be polynomially simulated by an equivalent uniform branching program ([RWY97]). It is an open question whether oblivious read-once branching programs are less powerful than "non-oblivious" ones.

To see the correspondence between branching programs and the resolution proof

[2]We can convert an arbitrary branching program into a leveled branching program such that both length and width will not exceed the size of the original device, up to a constant factor.

system, consider the following: Define the *search problem* associated with a set of unsatisfiable clauses C (clauses that correspond to the negation of a tautology) is this: given a truth assignment α find a clause in C that is false under α. This problem can be solved by a branching program whose sink nodes are labeled by the clauses in C.

Theorem 5.2.1 *The smallest*[3] *regular resolution refutation of a set of clauses C is equal to the size (number of nodes) of the smallest read-once branching program solving the search problem associated with C.*

For the proof of this theorem one simply needs to construct a read-once branching program given a regular resolution refutation and vice versa (see [Kra95]).

5.2.1 Previous Results

Haken in [Hak85] has shown that the pigeonhole principle PHP_n^{n+1} requires exponential size resolution proofs, while further improvements were given by Buss and Turan [BT88]. Recent results by Ran Raz [Raz98] have shown that the weak pigeonhole principle requires exponential size resolution proofs, solving a long standing open question. PHP_n^m has polynomial size proofs in extended resolution systems, Frege proof systems and in the cutting planes proof system [BT88, CCT87, Bus87].

Lower bounds for a function in $\mathcal{NP}$ computed by unrestricted branching programs, were given by Neciporuk [Nec66]. Also Babai et al. [BPRS90] prove a super-linear lower bound for the majority function (Pudlak also proves a non-trivial bound for majority function in [Pud84]).

Many lower bounds and general techniques have been given for restricted forms of branching programs and mostly for oblivious and read-once branching programs:

[3] *The size of a resolution refutation is the number of clauses that appear in the refutation*

exponential lower bounds are known for computing the parity of the number of triangles in a graph [ABH$^+$86, BHST87, SS93], size of regular resolution proofs for the pigeonhole principle [Hak85, RWY97] as well as the modular counting principle (see also [Kra95]), integer multiplication [Pon95] (also [Bry91]), for the clique-only function [BRS93, Weg87], the k-regularity problem for a graph [BHST87, SS93]. Recent work of Thathachar [Tha98] proves an exponential separation between consecutive levels of the hierarchy of read-k-times branching programs (result also applies to non-deterministic branching programs).

Except for these more natural problems, many exponential lower bounds have been proven for explicit functions (for some less "natural" problems than the ones mentioned in the previous paragraph) [Gál97, Pon95, ABH$^+$86, BHST87, Dun85]. For a more complete survey of lower bound results for branching programs see [Raz91].

The branching program model is also very useful for proving time-space tradeoff results (as mentioned in section 2.1). Recent work of Beame, Saks, Sun and Vee [BSSV00], based on previous results of [Ajt98, Ajt99b, Ajt99a] show very strong super-linear time-space trade-offs for randomized computation for (decision) problems in $\mathcal{P}$.

5.3 Definitions and Lemmata

As we mentioned in the introduction, in order to prove the separation result between the new push-down proof system and oblivious regular resolution, we need to allow the machines to read the input variables in any (fixed) order. A simple way to describe this fact, is to assume that the formulas we will be dealing with, are "easy" to compute by finite state automata, for any ordering of the variables, in which the automata will read the input. Another way to think about this restriction, is that we will require the given formulas to have the following property: for any ordering of the formula variables, there is a polynomial size finite state automaton description of the formula (i.e. a FSA that accepts all inputs (truth assignments) that make the formula true).

We will call boolean formulas with the above property, "fsa" formulas. For a given ordering of the (input) variables, FSA formulas can be described by a finite state automaton, of size polynomial in the number of variables: Let $\phi(x)$ be a formula, depending on the variables $x = (x_1, \ldots, x_n)$. We say that ϕ is an FSA formula, if there exists a finite state automaton of size polynomial in n, that computes the formula ϕ, for any permutation of the variables x.

More formally: consider a formula $\phi(x_1, \ldots, x_n)$, and let $x_{\pi_1}, \ldots, x_{\pi_n}$ be a permutation π of the variables. We will denote by $||\phi||_{fsa}$ the maximum size of the smallest FSA accepting all strings $x_{\pi_1}, \ldots, x_{\pi_n}$ for a permutation π with $\phi(x_1, \ldots, x_n) = 1$ (where the maximum is taken over all permutations π of the variables). Lets call this measure *the FSA-size of a formula*:

$$||\phi||_{fsa} = \max_{\pi} \left\{ \begin{array}{c} \text{smallest FSA accepting } x_{\pi_1}, \ldots, x_{\pi_n} \\ \text{for which } \phi(x_1, \ldots, x_n) = 1 \end{array} \right\}$$

Definition 5.3.1 *An FSA formula, is any formula $\phi(x_1, \ldots, x_n)$ whose FSA-size is polynomial in the number of variables n.*

For example, $||x_1 \vee \cdots \vee x_n||_{fsa} = O(1)$ and $||x_1 \oplus \cdots \oplus x_n||_{fsa} = O(1)$, but for any threshold function $||T_k^n||_{fsa} = O(n)$ since computing threshold functions involves counting. On the other hand consider the problem of checking whether the input is a palindrome, ww^R, where w^R is the string w reversed. It is easy to see that there is an ordering of the variables that will force an exponential FSA size. On the other hand there is also an ordering that makes the FSA size constant. In other words, the hardness of this problem depends on the ordering of the input variables. With our definition of FSA formulas, we can argue that the complexity of computing the formula is independent of the ordering of the variables.

For this model, we can show that a proof system based on oblivious read-once branching program machines is strictly less powerful than the proof system of push down BDD's. We can prove this separation using the meander lemma of Alon and Maass [AM88]. If, on the other hand, we allow the input to be ordinary clauses, then it is open if the push down model has greater power. Since it is known that read-once branching programs correspond to regular resolution, the fact that the push down model can solve problems that require exponential size oblivious read-once branching programs, implies that the push-down BDD model is a strictly more powerful proof system than oblivious[4] regular resolution.

Let $\mathcal{C}$ be a set of clauses, $\mathcal{C} = \{C_1, \ldots, C_k\}$ where the clauses C_i depend on variables $x = (x_1, \ldots x_n)$. A truth assignment α satisfies $\mathcal{C}$ if it satisfies all clauses in $\mathcal{C}$ (we can consider $\mathcal{C}$ as a conjunction $\mathcal{C} = \bigwedge_{i=1}^{k} C_i$). The set $\mathcal{C}$ is unsatisfiable, if there is no truth assignment which satisfies all clauses in $\mathcal{C}$. For our purposes we need to assume that C_i are FSA clauses.

Definition 5.3.2 *Define an automata-based proof system to be a class of machines $\mathcal{M}$. We say that $\mathcal{M}$ can give a refutation of a set of clauses $\mathcal{C}$, if there exists a machine $M \in \mathcal{M}$ such that, for a given truth assignment α which assigns the values $\bar{x}$ to the variables x, M can find and output a clause from $\mathcal{C}$ that is false:*

1. $M(\bar{x}) = i \Longrightarrow C_i(\bar{x}) = false$

2. $M(\bar{x}) = i \Longrightarrow i \in \{1, 2, \ldots, k\}$

[4]We need to consider oblivious BDD's in order to apply the techniques of Alon and Maas [AM88]

Note that the clauses C_i are described by finite state automata (FSA formulas). Since we allow our machines to read the variables in any (fixed) order, the complexity of computing the clauses C_i should be independent of the ordering of the variables.

As mentioned above, we will consider proof systems that lie in $\mathcal{NP}$. This implies that we must restrict our finite machines to classes for which the correctness of the computation can be checked in polynomial time.

The following lemma (5.3.3) gives a characterization of some automata based proof systems that lie in $\mathcal{NP}$. Checking the correctness of the computation of a machine M from a class of automata $\mathcal{M}$, involves verifying the two properties of definition 5.3.2; the second property ($i \in \{1, 2, \ldots, k\}$) is easy to check. For the first property (if M outputs i, then the i-th clause is false) we need to check if there exists an input $\bar{x}$ such that M will output i and C_i is false under $\bar{x}$ (and $\bar{x}$ has length n). This can be formulated as a standard emptiness problem for an intersection of automata: is there a string $\bar{x}$ (of length n) for which M outputs i *and* C_i is false under $\bar{x}$? In other words, is the intersection of M and C_i non-empty[5] ? The standard algorithm for testing whether the intersection of finite automata is empty, is to construct their "cartesian product" and solve the emptiness problem for that machine. Using these ideas we can prove the following lemma:

Lemma 5.3.3 *If $\mathcal{M}$ is any class of machines and $M \in \mathcal{M}$, it is possible to check the correctness of the computation of M in time polynomial in the size of M, if the following hold:*

- *The class $\mathcal{M}$ is closed under Cartesian product with finite state automata: that is $(M \times F) \in \mathcal{M}$ for any $M \in \mathcal{M}$ and FSA F, and also we can compute the machine $M \times F$ in polynomial time.*

[5]We also need to check that $\bar{x}$ has the right length, so we will use one more automaton to check that, and solve the emptiness problem for the intersection of the three automata in lemma 5.3.3

- *The emptiness problem for any $M \in \mathcal{M}$, can be solved in time polynomial in the size of M.*

Proof Consider a class $\mathcal{M}$, and lets assume that $\mathcal{M}$ solves the unsatisfiability problem for a set $\mathcal{C} = \{C_1, \ldots, C_n\}$. To verify the correctness of the computation (verify the proof of unsatisfiability of $\mathcal{C}$) of a machine $M \in \mathcal{M}$, we must check that the output i of the machine is in $\{1, \ldots, k\}$ and also, if $M(\bar{x}) = i$ then $C_i(\bar{x}) = false$.

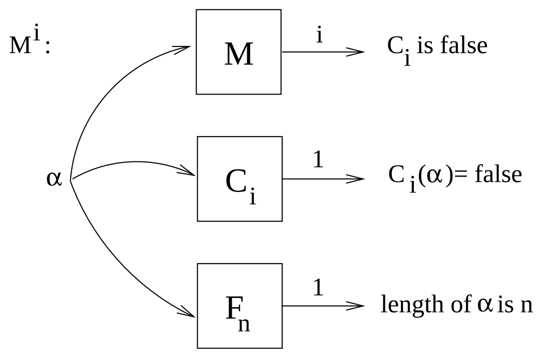


Figure 5.2: Checking correctness of a proof system

The language[6] $L(M)$ contains all strings $\bar{x}$ that do not satisfy some clause C_i. The clause C_i is described by an FSA (denoted by C_i) and $\bar{C}_i$ is the FSA that accepts all inputs that make the clause false. Consider the machine $M^i = (M \times \bar{C}_i)$. If the class $\mathcal{M}$ is closed under Cartesian product with an FSA, then $M \in \mathcal{M} \implies M^i \in \mathcal{M}$. We now want to solve the emptiness problem for M^i: does $L(M^i)$ contain a string of length n? In order to formalize this as a standard emptiness problem, denote by F_n an FSA which accepts only strings of length n and define $M^i = (M \times \bar{C}_i \times F_n)$. Observe that if $L(M^i)$ is non-empty and $\bar{x} \in L(M^i)$ then:

- $|\bar{x}| = n$

[6]The machine M accepts a string $\bar{x}$ if there is a clause C_i such that $C_i(\bar{x}) = false$ and outputs the index i.

- $\bar{x}$ is accepted by $\bar{C}_i$ and therefore $C_i(\bar{x})$ is false.

- if $M(\bar{x}) = i$ then indeed $C_i(\bar{x}) = false$

So, since $M^i \in \mathcal{M}$, if the emptiness problem for machines from $\mathcal{M}$ is decidable in polynomial time, then the correctness of a system based on $\mathcal{M}$ can be checked in polynomial time, and therefore the proof system is in $\mathcal{NP}$. ▮

The previous lemma, gives a method for proving that a class $\mathcal{M}$ defines a proof system that lies in $\mathcal{NP}$. For example, consider the class of finite state automata. The emptiness problem is easy: it can be reduced to a reachability problem in the graph describing the FSA (find a path from the start state to any final state). It is also easy to see that the Cartesian product of two FSA's is also an FSA. In the same way, branching programs, binary decision diagrams and so on are also classes that imply proof systems that lie in $\mathcal{NP}$.

We can prove that for push down automata, correctness is also checkable in time polynomial in the size of the machine.

Recall that the definition of a PDA is the following [HU79]: A push-down automaton (PDA) is the system $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where Q is a finite set of states, Σ and Γ are the input and stack alphabets respectively, $q_0 \in Q$ is the initial state, $Z_0 \in \Gamma$ is the initial stack symbol, $F \subseteq Q$ is the set of final states and δ is the transition function: $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \implies \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$, where $\mathcal{P}(X)$ denotes the power set of X and $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$.

The class of (polynomial size) PDA's is closed under Cartesian product with an FSA, and also we can see that the emptiness problem for a PDA can be solved in polynomial time. The following theorem combines classic results for push-down automata and context free languages. But note that those results, do not usually state the properties that we need for our purposes, namely that the well-known algorithms for push-down automata need time polynomial in the *size of the machine*, too.

Theorem 5.3.4 *The emptiness problem for a push down automaton with n states can be decided in time polynomial in n.*

Proof The proof is straightforward, and combines the well known technique for converting a PDA to a context free grammar and the standard algorithms for the emptiness problem for context free languages.

In order to see if the language of a PDA is empty, first we convert it to a context free grammar and then we use the standard algorithm for the emptiness problem in context free grammars (see [HU79])

- Convert the PDA to an equivalent context-free grammar: Define the grammar $G = (V, \Sigma, P, S)$, such that: the set of non-terminal symbols is $V = Q \times \Gamma \times Q = \{[q, A, p] : p, q \in Q, A \in \Gamma\}$. The size of V is $|V| = O(n^2 \cdot |\Gamma|)$ where the size of the alphabet is fixed, so $|V| = O(n^2)$. The set of Production rules P consists of productions of the form: $[q, A, q_{m+1}] \implies a[q_1, B_1, q_2] \ldots [q_m, B_m, q_{m+1}]$ where $q_i \in Q$, $a \in \Sigma$, $A, B_i \in \Gamma$, and $(q_1, B_1 \cdots B_m) \in \delta(q, a, A)$, and $S \implies [q_0, Z_0, q]$ for all $q \in Q$. The size of P is also polynomial in n, since the size of the transition function δ of the PDA is $O(n^2 \cdot |\Sigma| \cdot |\Gamma|^2)$, and P will have the same size as δ.

- The standard algorithm for checking emptiness of a context-free grammar runs in time linear in the number of non-terminal symbols $|V| = O(n^2 \cdot |\Gamma|)$. The following algorithm checks whether the symbol S is useless or not. The set N contains all the non-terminal symbols that can lead to a string of Σ^* consisting of terminal symbols only.

  1. $N_0 = \emptyset$, $i = 1$.

2. $N_i = \{A | A \implies \alpha$ in P and $\alpha \in (N_{i-1} \cup \Sigma)^*\} \cup N_{i-1}$

3. if $N_i = N_{i-1}$ then $i = i + 1$ and goto (2) else if $S \in N_i$ then the language in non-empty else the language is empty.

Step 2 also requires polynomial time, since $|V| = O(n^2 \cdot |\Gamma|)$.

▮

We will consider a class of machines based on BDD's that read their input once, but have access to a push-down stack: a machine M from this class is defined as follows:

Definition 5.3.5 *Let $X = (x_1, \ldots, x_n)$ denote the set of input variables, over the alphabet Σ. A push down BDD is the system $(G, \Sigma, \Gamma, \alpha, \delta, v_0, Z_0, F)$: $G = (V, E)$ is a directed acyclic graph, Σ and Γ are the input and stack alphabets respectively, $v_0 \in V$ is the source node (of in-degree 0) in the graph G, Z_0 is the initial stack symbol, and $F \subseteq V$ is the set of sink nodes, and $|F| = n$. Every node is labeled by an input variable given by the labeling function $\alpha : V \to X$ and every edge is labeled according to the transition function $\delta : V \times \Sigma_\epsilon \times \Gamma_\epsilon \to V \times \Gamma_\epsilon$*

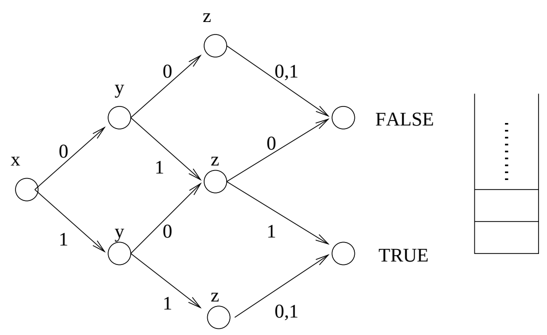


Figure 5.3: Push-down branching programs

The push-down BDD machines, have the same basic structure as BDD's, but in addition, they have access to a push down stack, and the transition from each node, depends on both the value of the input variable tested on that node and the value of the symbol in the top of the stack (the out degree of all non-sink nodes in the graph G is $|\Sigma| \cdot |\Gamma|$). On the other hand, the push-down BDD's have the same structure as the usual push-down automata, but they can "choose" any (fixed) way to read their input. Since our push-down BDD proof system will handle FSA formulas, we know that any ordering of the variables is easy. Once we fix the ordering of the input variables given the fact that we deal with FSA formulas, we know that our push-down system corresponds to a push down automaton of polynomial size reading the variables in this ordering. So, using theorem 5.3.4 and lemma 5.3.3 we conclude the following (Recall that the push-down BDD proof system is defined as described in definition 5.3.2.):

Corollary 5.3.6 *The proof system based on the class of push-down BDD's lies in $\mathcal{NP}$.*

5.4 Links and Meanders

Alon and Maass [AM88] describe a general technique for proving lower bounds for branching programs.

Let $X = x_1 x_2 \ldots x_m$ be a sequence of numbers $x_i \in \{1, 2, \ldots, n\}$, and consider two disjoint subsets of $\{1, 2, \ldots, n\}$, $S, T \subseteq \{1, 2, \ldots, n\}$. We say that an interval $x_i x_{i+1} \ldots x_{i+j}$ is a *link between S and T* if $x_{i+1} \ldots x_{i+j-1} \notin S \cup T$ and $x_i \in S$, $x_{i+j} \in T$, or $x_i \in T$ and $x_{i+j} \in S$.

We say that a sequence of X over n numbers is a *meander* if for any two disjoint sets $S, T \subseteq \{1, 2, \ldots, n\}$ with $|S| = |T|$, there are in X at least $|S|$ links between S and T. For any function $g : N \implies R^+$, X is a $g(x)$-meander if for any disjoint sets $S, T \subseteq \{1, 2, \ldots, n\}$ with $|S| = |T|$, there are at least $g(|S|)$ links between S and T.

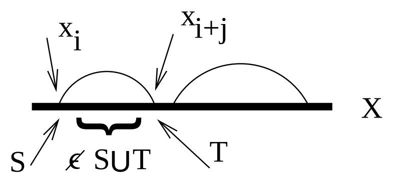


Figure 5.4: Links and meanders

Lemma 5.4.1 *Assume $s(n)$ is some arbitrary function and X is a sequence over $\{1, 2, \ldots, n\}$. In order to prove that $|X| = \Omega(n \cdot s(n))$ it is sufficient to show for some $k \leq n/2^{s(n)}$ that for any two sets $S \subseteq \{1, 2, \ldots, n/2\}$ and $T \subseteq \{n/2 + 1, \ldots, n\}$ of size k there are in X at least $s(n)$ links between S and T.*

In [AM88] the above result is used to prove two lower bounds for oblivious read-once branching programs:for the *Set Equality* problem $S(n, m)$ and the *sequence equality* function $Q(n)$.

5.5 The Lower Bound for the branching program model

In order to show that the proof system based on push-down BDD automata described above is more powerful than the BDD model, we prove an exponential lower bound for oblivious read-once branching programs, for deciding an appropriate language. We define the language $\mathcal{L}_n$ (based on the Pigeonhole principle) as follows: Consider strings over the alphabet $\{0, 1, \circ\}$, where $\circ$ is a symbol that can be ignored by our machines. Let $x, y \in \{0, 1, \circ\}^n$ where $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$. Also denote by $\hat{x}$ and $\hat{y}$ the strings resulting from x and y if we omit all occurrences of the symbol $\circ$. Define the language $\mathcal{L}_n$ to be a set of pairs of strings x, y, having the following properties:

- x, y have the same length, $|x| = |y| = n$
- $\hat{x}, \hat{y}$ have the same length, $|\hat{x}| = |\hat{y}| = m$, and there are $\frac{m}{2} + 1$ ones and $\frac{m}{2} - 1$ zeroes in both x and y. m depends only on n. Assume that $m = n/2$.

By the above definition, it follows (by the pigeonhole principle) that there exists an index i such that $\hat{x}_i = \hat{y}_i = 1$. This is the search problem associated with $\mathcal{L}_n$: given two strings x and y, find a position i such that $\hat{x}_i = \hat{y}_i$.

The properties given above, can be described by a set of clauses that encode the negation $\neg\mathcal{L}_n$ of our pigeonhole language, as follows: either there are less than $m/2+1$ ones in x or y, *or* the two strings, x and y, "agree" in some position. The latter can be expressed by a set of clauses of the form $\{\mathcal{C}_{r,s}(x, y)\}$ for all $r, s < n$, where each clause $\mathcal{C}_{r,s}(x, y)$ describes that, $x_r = y_s$ and the number of skip "$\circ$" symbols in the prefixes $\{x_i | i < r\}$ and $\{y_i | i < s\}$ is the same.

Theorem 5.5.1 *A push-down BDD can recognize the language $\mathcal{L}_n$ in time polynomial in n (and the size of the push-down BDD is also polynomial in n).*

Proof

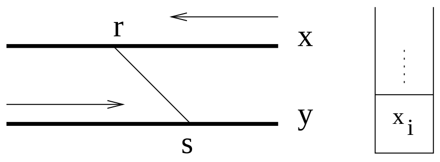


Figure 5.5: Push-down BDD accepts $\mathcal{L}_n$ in polynomial time.

The PDA can check whether (x, y) is in the language $\mathcal{L}_n$ the following way: Read x from right to left. For each symbol $x_r \neq \circ$, push x_r on the stack. When we finish reading x, we start reading y from left to right: for each symbol $y_s \neq \circ$ of y pop from the stack the top symbol x_r and check if $x_r = y_s = 1$. This way we can verify if a pair $(x, y) \in \mathcal{L}_n$. ▮

In the branching program model, it is not possible to solve the search problem associated with $\mathcal{L}_n$, unless we allow the branching programs to have super-polynomial number of nodes. Note that if in our definition of $\mathcal{L}_n$ we did not use the symbols '$\circ$', then BDD's would also be able to solve the search problem described above, since BDD's can choose the ordering in which they will read their input.

Theorem 5.5.2 *Any (3−way) oblivious (read−once) branching program of width $2^{n/2^{h(n)}}$ solving the search problem associated with the language $\mathcal{L}_n$, has length $\Omega(n \cdot h(n))$.*

Proof

Consider an oblivious read-once branching program with length $m = 2n$ and width $w \leq 2^{n/2^h}$. The input of the branching program, will be the strings $x, y \in \{0, 1, \circ\}^n$ which will be read in some arbitrary (fixed) order. The size of the input is $2n$. Let X be a sequence of length $m = 2n$, $X =< \alpha_1, \ldots \alpha_{2n} >$ over $\{1, 2, \ldots, 2n\}$ such that:

$$\alpha_i = \begin{cases} j & \text{if the } i\text{-th level vertices are labeled by } x_j \\ n+j & \text{if they are labeled by } y_j \end{cases}$$

Set $s \equiv h/2$ and suppose $S \subseteq \{1, \ldots, n\}$ and $T \subseteq \{n+1, \ldots, 2n\}$ with $|S| = |T| = 2n/2^s$. By lemma 5.4.1 it is sufficient to show that there are in X at least s links between S and T.

Consider inputs of the form $I_{AB} =< z_1, \ldots, z_{2n} >$, where $z_i = \circ$ for all $i \notin S \cup T$, and $A =< z_i >_{i \in S}$, $B =< z_j >_{j \in T}$ such that A and B have a "1" at the same position. We can now use a "crossing sequence" argument to find the number of links between S and T: If we consider an instance $I_{A'B'} =< z'_1, \ldots, z'_{2n} >$, such that $A \neq A'$ then

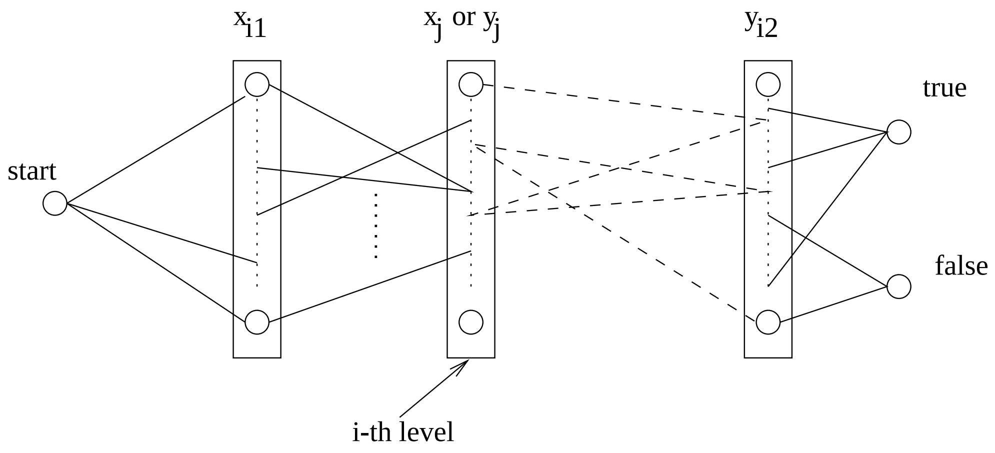


Figure 5.6: Oblivious read-once branching program for the $\mathcal{L}_n$

there is a link l such that the computation path in the branching program for I_{AB} differs from that of $I_{A'B'}$ on that level of the branching program that corresponds to the last element of the link l. Otherwise, the branching program for I_{AB} would also accept the input $I_{A'B}$. The number of different choices for A is $2^{|S|}$ (if A and A' differ even in only one position they should lead to different states in the branching program). So, if we denote the set of links between S and T by L:

$$w^{|L|} \geq 2^{|S|} = 2^{2n/2^s} \Rightarrow |L| \geq 2^s \geq s$$

∎

The length of our BDD's is $2n$, since they are read-once, and therefore the previous theorem shows that the size of such machines solving the search problem corresponding to $\mathcal{L}_n$ is exponential.

5.6 Discussion and Open Problems

Proving the separation for ordinary clauses seems to be more difficult. For the language $\mathcal{L}_n$ our clauses must express that $x, y \in \mathcal{L}_n$ iff $x_i = y_j = 1$ and the number of symbols equal to one is the same in both $\{x_1, \ldots, x_i\}$ and $\{y_1, \ldots, y_j\}$. The second condition involves counting, and would require super-polynomial size formulas (but only polynomial size FSA formulas). One way to try to overcome this problem is to change the definition of the language, and allow the use of auxiliary bits in the input that count the number of ones in the string: then the clauses describing the new language are polynomial in size, (ordinary clauses). The new language, $\mathcal{L}'_n$, is the following: A string $x_1, a_1, \ldots, x_n, a_n, y_1, b_1, \ldots, y_n, b_n \in \mathcal{L}'_n$ iff $x_i = x_j = 1$ and $a_i = b_j$, where $a_i = |\{x_k | k \leq i, \text{ and } x_k = 1\}|$ (same for b_i's). Note that $x_i, y_i \in \{0, 1, \circ\}$ and both a_i and b_i have length $O(\log n)$. The length of the input is $2n + 2n \log n$. The correctness of the numbers a_i, b_i can be verified by both push down and FSA machines by simply checking that $a_{i+1} = a_i + x_i$. Since $|a_i| = O(\log n)$, we can write the previous equation as a polynomial size formula. The lower bound proof however does not go through for this case. Some different approach seems to be needed.

Let $\mathcal{M}$ be a class of read-once machines, and denote by $\mathcal{M}_O$ the class of machines from $\mathcal{M}$ which are oblivious: For any computation path of the machine, the input variables are tested in the same order. For example, if $\mathcal{M}$ = class of FSA's, then $\mathcal{M}_O$ is the class of oblivious BDD's. We can prove the following:

Proposition 5.6.1 *The emptiness problem is polynomial for $\mathcal{M}$ if and only if it is polynomial for the corresponding oblivious class $\mathcal{M}_O$.*

Note that $\mathcal{M}_O \subseteq \mathcal{M}$ and therefore the "only if" part is trivial. For the "if" part, notice that any machine $M \in \mathcal{M}$ will read a specific input in some order. We can construct an oblivious machine in $\mathcal{M}_O$, which will read the input in the order defined by M. And since the emptiness problem is polynomial for $\mathcal{M}_O$, it is also polynomial for M.

The BDD model is as powerful as oblivious regular resolution [Kra95]. Theorem 5.5.2 shows that the push down BDD proof system is strictly more powerful than oblivious regular resolution, when our formulas are general FSA machines. If we consider ordinary formulas and not generalized ("fsa") formulas, then it is open whether the push down model of the BDD logic is more powerful than the BDD model.

As mentioned in the introduction, the push-down based proof system essentially contains a form of read-twice branching programs: If x is the input, using the push-down stack we can easily simulate a read-twice branching program reading xx^R. It is not clear however what is the exact relation between the two proof systems. This weak read-twice version of the PDA system seems to have enough power to make a lower bound for pigeonhole principle harder to prove. For example consider the following problem [RWY97]: given an input $A \in \{1, 2, \ldots, n\}^m$ find some $i_1, i_2 \in \{1, 2, \ldots, m\}$ where $i_1 \neq i_2$ such that $A_{i_1} = A_{i_2} = j$. This is called the row model for the pigeonhole principle. A PDA BDD can make a single pass on the input A, and compute some useful function of the input variables, and use this information to re-read the input. So if the PDA machine has access to some function that is easy to compute for any order of the input variables it clearly has an advantage, which seems to pose new problems in proving a lower bound.

A natural question for the push-down BDD model is the following: is there an exponential lower bound for the size of the proof of some special kind of tautology? The integer multiplication problem seems to be a candidate for an exponential lower bound, since a polynomial size proof for $x * y = z$ would give improved results for factoring. In particular we can prove the following:

Proposition 5.6.2 *If there exists a push-down BDD M_{mult} accepting the language*

$x \cdot y = z$ with size S, then factoring can be solved in time $S^{O(1)}$

Proof Indeed, given the push-down machine M_{mult} we can factor a number $z = x \cdot y$ in the following manner: starting from M_{mult} construct the push down machine M^1_{mult} by restricting the first bit of y to be equal to 1. If the language accepted by M^1_{mult} is non-empty, then there is a factor y of z whose first bit is 1. Otherwise the first bit has to be 0. Continue in the same way by fixing the i-th, $i = 1, 2, ..$ bit of y and solving the emptiness problem for the corresponding M^i_{mult}. This will give us the factor y of z. The size of the machines M^i_{mult} have the same size as M_{mult} and the emptiness problem is solvable in time polynomial in the size of M_{mult} (by theorem 5.3.4). Therefore the factoring algorithm described above, is polynomial in the size of the machine M_{mult}. ▮

Another interesting problem, is to generalize the results of this section, to probabilistic machines. This will enable us to consider a notion of probabilistic automata based proof systems. One difficulty for this generalization would be to show that the emptiness problem for probabilistic machines is easy.

There is a number of general methods for proving lower bounds in restricted proof systems. Counting the number of subfunctions for example is a combinatorial method used in many proofs of lower bounds for read-once branching programs ([SS93, Gál97]). The technique is the following: Let X be a set of variables and consider a partition of X into $Y \subseteq X$, and $B = X \setminus Y$. Every truth assignment α for the variables in Y, defines a *subfunction* $f_\alpha : B \implies \{0, 1\}$. Denote by $\{f, B\}$ the set of such subfunctions and by $N(f, Y)$ the number of subfunctions (cardinality of $N(f, Y)$). The following lemma gives a general technique for proving lower bounds for the size of read-once branching programs:

Lemma 5.6.3 ([Gál97, SS93]) *Let f be a boolean function on the variables X,*

$|X| = n$. If m is an integer, $1 \leq m \leq n$, such that for any subset $Y \subseteq X$, of size $|Y| = m$, $N(f, Y) = 2^m$ then the size of any read-once branching program computing f is at least $2^m - 1$.

The technique described in the previous lemma does not seem to generalize to directly apply in the case of the push down proof system.

In conclusion, here is list of some open problems for the new push-down BDD proof system:

1. Prove a separation between push-down BDD proof system and branching programs for ordinary (not FSA) clauses.

2. Prove that the push down BDD proof system is exponential.

3. Find the complexity of the Pigeonhole principle and integer multiplication in the push-down BDD model.

4. Find the relation of the new proof system to read-twice branching programs or to other well known proof systems like resolution.

Bibliography

[ABH$^+$86] Miklós Ajtai, László Babai, Péter Hajnal, János Komlós, Pavel Pudlák, Vojtěch Rödl, Endre Szemerédi, and György Turán. Two lower bounds for branching programs. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 30–38, Berkeley, California, 28–30 May 1986.

[Abr90] K. Abrahamson. A time-space tradeoff for Boolean matrix multiplication. In IEEE, editor, *Proceedings: 31st Annual Symposium on Foundations of Computer Science: October 22–24, 1990, St. Louis, Missouri*, volume 1, pages 412–419, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990. IEEE Computer Society Press.

[Ajt98] Ajtai. Determinism versus non-determinism for linear time RAMs with memory restrictions. In *ECCC: Electronic Colloquium on Computational Complexity, technical reports*, 1998.

[Ajt99a] M. Ajtai. A non-linear time lower bound for Boolean branching programs. In IEEE, editor, *40th Annual Symposium on Foundations of Computer Science: October 17–19, 1999, New York City, New York,*, pages 60–70, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. IEEE Computer Society Press.

[Ajt99b] Miklos Ajtai. Determinism versus non-determinism for linear Time RAMs with memory. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 632–641, New York, May 1999. Association for Computing Machinery.

[AM88] Noga Alon and Wolfgang Maass. Meanders and their applications in lower bound arguments. *Journal of Computer and System Sciences*, 37(2):118–129, October 1988.

[Bac90] Eric Bach. Number-theoretic algorithms. In *Annual Review of Computer Science*, volume 4, pages 119–172. Annual Reviews, Inc., 1990.

[BC82] A. Borodin and S. Cook. A Time space trade off for sorting on a general sequential model of Computation. *SIAM Journal on Computing*, 11:287–297, 1982.

[BE98] Greg Barnes and Jeff A. Edmonds. Time-space lower bounds for directed *st*-connectivity on graph automata models. *SIAM Journal on Computing*, 27(4):1190–1202, August 1998.

[Bea91] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM Journal on Computing*, 20(2):270–277, April 1991.

[BHST87] László Babai, Péter Hajnal, Endre Szemerédi, and György Turán. A lower bound for read-once-only branching programs. *Journal of Computer and System Sciences*, 35(2):153–162, October 1987.

[Bla37] A. Blake. *Canonical expressions in boolean algebra.* PhD thesis, University of Chicago, 1937.

[Bor77] A. Borodin. On relating time and space to size and depth. *SIAM Journal of Computing*, 6(4):733–744, December 1977.

[Bor93] Borodin. Time-space tradeoffs. In *ISAAC: 4th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE))*, 1993.

[BP98] Paul Beame and Toniann Pitassi. Propositional proof complexity: Past, present, future. *Bulletin of the EATCS*, 65:66–89, June 1998.

[BPRS90] L. Babai, P. Pudlák, V. Rödl, and E. Szemeredi. Lower bounds to the complexity of symmetric Boolean functions. *Theoretical Computer Science*, 74(3):313–323, 28 August 1990.

[BRS93] Allan Borodin, Alexander A. Razborov, and Roman Smolensky. On lower bounds for read-k-times branching programs. *Computational Complexity*, 3(1):1–18, 1993.

[Bry91] R. Bryant. On the Complexity of VLSI implementations and graph representations of boolean functions with applications to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[BSSV00] P. Beame, M. Saks, Xiaodong Sun, and E. Vee. Super-linear time-space tradeoff lower bounds for randomized computation. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 169–179, 1109

Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000. IEEE Computer Society Press.

[BST98] Paul Beame, Michael Saks, and Jayram S. Thathachar. Time-space tradeoffs for branching programs. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 254–263, 1998.

[BT88] S. Buss and G. Turan. Resolution proofs of generalized pigeonhole principle. *Theoretical Computer Science*, 62:311–317, 1988.

[Bus87] Samuel R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52(4):916–927, December 1987.

[CCT87] W. Cook, C. Coullard, and G. Turan. On the Complexity of Cutting Plane Proofs. *Discrete Applied Mathematics*, 18:25–38, 1987.

[Coo73] Stephen A. Cook. A hierarchy for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7(4):343–353, August 1973.

[Coo88] Stephen A. Cook. Short propositional formulas represent nondeterministic computations. *Information Processing Letters*, 26(5):269–270, January 1988.

[CR73] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, August 1973.

[CR79] Stephen Cook and Robert Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44:36ff, 1979.

[DF92] Rodney G. Downey and Michael R. Fellows. Fixed-parameter intractability (extended abstract). In *Proceedings of the Seventh Annual Structure* *in Complexity Theory Conference*, pages 36–49, Boston, Massachusetts, 22–25 June 1992. IEEE Computer Society Press,.

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer, 1999.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[Dun85] P. E. Dunne. Lower bounds on the complexity of 1-time only branching programs. *FCT: Fundamentals (or Foundations) of Computation Theory*, 5, 1985.

[Edm98] Jeff A. Edmonds. Time-space tradeoffs for undirected st-connectivity on a graph automata. *SIAM Journal on Computing*, 27(5):1492–1513, October 1998.

[EP95] Jeff Edmonds and Chung Keung Poon. A nearly optimal time-space lower bound for directed st-connectivity on the NNJAG model. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 147–156, Las Vegas, Nevada, 29 May–1 June 1995.

[FK97] Uriel Feige and Joe Kilian. On limited versus polynomial nondeterminism. *Chicago Journal of Theoretical Computer Science*, March 1997.

[For97] L. Fortnow. Nondeterministic polynomial time versus nondeterministic logarithmic space: Time-space tradeoffs for satisfiability. In *Proceedings of the 12th Annual IEEE Conference on Computational Complexity (CCC-97)*, pages 52–60, Los Alamitos, June24–27 1997. IEEE Computer Society.

[For00a] L. Fortnow. Diagonalization. *Bulletin of the European Association for Theoretical Computer Science*, 71:102–112, June 2000. Columns: Computational Complexity.

[For00b] Lance Fortnow. Time-space tradeoffs for satisfiability. *JCSS: Journal of Computer and System Sciences*, 60:337–353, 2000.

[FvM00] Lance Fortnow and Dieter van Melkebeek. Time-space tradeoffs for nondeterministic computation. In *Computational Complexity Conference*, pages 2–13, Florence, Italy, 4–7 July 2000.

[Gál97] Anna Gál. A simple function that requires exponential size read-once branching programs. *Information Processing Letters*, 62(1):13–16, 14 April 1997.

[GS90] Y. Gurevich and S. Shelah. Nondeterministic linear-time tasks may require substantially nonlinear deterministic time in the case of sublinear work space. *Journal of the ACM, JACM*, 37(3):674–687, July 1990.

[Hak85] A. Haken. The Intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[HPV77] J. Hopcroft, W. Paul, and L. Valiant. On time versus space. *Journal of the ACM.*, 24(2):332–337, April 1977.

[HU79] John Hopcroft and Jefrey Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Kan81] R. Kannan. A circuit-size lower bound. In *22th Annual Symposium on Foundations of Computer Science*, pages 304–309, Los Alamitos, Ca., USA, October 1981. IEEE Computer Society Press.

[Kan84] Ravindran Kannan. Towards separating nondeterminism from determinism. *Mathematical Systems Theory*, 17(1):29–45, April 1984.

[Kar86] M. Karchmer. Two time-space tradeoffs for element distinctness. *Theoretical Computer Science*, 47(3):237–246, 1986.

[KLV00] George Karakostas, Richard J Lipton, and Anastasios Viglas. On the Complexity of intersecting finite state automata. In *Proceedings of the 15th Annual IEEE Conference on Computational Complexity*, pages 229–234, Florence, Italy, July7–9 2000.

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science*, pages 254–266. IEEE, October 1977.

[Kra95] Jan Krajicek. *Bounded Arithmetic, Propositional Logic and Complexity Theory*. Cambridge University Press, 1995.

[LV99a] Richard J. Lipton and Anastasios Viglas. On the complexity of SAT. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 459–464, New York City, New York, 17–19October 1999.

[LV99b] Richard J Lipton and Anastasios Viglas. On the Power of Automata based Proof Systems. In *Proceedings of the 2nd Panhellenic Logic Symposium*, Delphi, Greece, July13–17 1999.

[LV01] Richard J Lipton and Anastasios Viglas. New results using Block Respecting Turing Machine computation, 2001.

[Nec66] Neciporuk. A boolean function. *DOKLADY: Russian Academy of Sciences Doklady. Mathematics (formerly Soviet Mathematics–Doklady)*, 7, 1966.

[Pon95] Stephen Ponzio. A lower bound for integer multiplication with read-once branching programs. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, pages 130–139, Las Vegas, Nevada, 29 May–1 June 1995.

[PPST83] Wolfgang J. Paul, Nicholas Pippenger, Endre Szemerédi, and William T. Trotter. On determinism versus non-determinism and related problems (preliminary version). In *24th Annual Symposium on Foundations of Computer Science*, pages 429–438, Tucson, Arizona, 7–9 November 1983. IEEE.

[PR80] Wolfgang J. Paul and Rüdiger Reischuk. On alternation II. A graph theoretic approach to determinism versus nondeterminism. *Acta Informatica*, 14:391–403, 1980.

[PR81] W. Paul and R. Reischuk. On time versus space II. *Journal of Computer and System Sciences*, 22(3):312–327, June 1981.

[PTC77] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. *Mathematical Systems Theory*, 10:239–251, 1977.

[Pud84] P. Pudlak. A Lower bound on the Complexity of Branching Programs. *Conference on the Mathematical Foundations of Computer Science*, 176:480–489, 1984.

[Raz91] A. A. Razborov. Lower bounds for deterministic and nondeterministic branching programs. *Lecture Notes in Computer Science*, 529:47–61, 1991.

[Raz98] Ran Raz. Resolution Lower Bounds for the Weak Pigeonhole Principle. Technical Report TR01-021, Electronic Colloquium on Computational Complexity ECCC, 1998.

[Rob65] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rob91] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, May 1991.

[RWY97] Alexander Razborov, Avi Wigderson, and Andrew Yao. Read-Once Branching Programs, Rectangular Proofs of the Pigeonhole Principle and the Traversal Calculus. *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, STOC'97*, pages 739–748, 1997.

[Sch78] C. P. Schnorr. Satisfiability is quasilinear complete in NQL. *Journal of the ACM*, 25(1):136–145, January 1978.

[SFM73] J. I. Seiferas, M. J. Fischer, and A. R. Meyer. Refinements of the nondeterministic time and space hierarchies. In Ronald V. Book, editor, *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 130–137, University of Iowa, October 1973. IEEE Computer Society Press.

[SFM78] Joel I. Seiferas, Michael J. Fischer, and Albert R. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25(1):146–167, January 1978.

[Sip86] M. Sipser. Ezpanders, randomness, or time versus space. In Alan L. Selman, editor, *Proceedings of the Conference on Structure in Complexity Theory*, volume 223 of *LNCS*, pages 325–329, Berkeley, CA, June 1986. Springer.

[Sip88] M. Sipser. Expanders, randomness, or time versus space. *Journal of Computer and System Sciences*, 36, 1988. Contains a discussion on efficiently reducing the probability of error in randomized algorithms. It also describes a relationship between pseudorandomness, time and space used by certain algorithms if certain types of expander graphs can be explicitly constructed.

[SS93] Janos Simon and Mario Szegedy. A new Lower Bound Theorem for Read-Only-Once Branching Programs and its Applications. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 13, 1993.

[Tha98] Jayram Thathachar. On Separating the Read-k-Times Branching Program Hierarchy. Technical Report 2, Electronic Colloquium on Computational Complexity ECCC, 1998.

[Tou00] Iannis Tourlakis. Time-space lower bounds for SAT on uniform and non-uniform machines. In *Computational Complexity Conference*, pages 22–33, Florence, Italy, 4–7 July 2000.

[Urq95] Alasdair Urquhart. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 1(4):425–467, December 1995.

[vM01] Dieter van Melkebeek. Time-space lower bounds for satisfiability. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 73, 2001.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions.* John Wiley & Sons, 1987.

[Yao88] A. C.-C. Yao. Near-optimal time-space tradeoff for element distinctness. In IEEE, editor, *29th annual Symposium on Foundations of Computer Science, October 24–26, 1988, White Plains, New York*, pages 91–97, 1109

Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1988. IEEE Computer Society Press.

[Yao94] Andrew Chi-Chih Yao. Near-optimal time-space tradeoff for element distinctness. *SIAM Journal on Computing*, 23(5):966–975, October 1994.