On the Power of Automata Based Proof Systems

Richard J. Lipton* Anastasios Viglas†

Computer Science Department
Princeton University

June 7, 1999

Abstract

One way to address the $NP = co - NP$ question is to consider the length of proofs of tautologies in various proof systems. In this work we consider proof systems defined by appropriate classes of automata. 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, in a certain sense, more powerful than oblivious regular resolution.

1 Introduction

One of the famous open questions of complexity theory is: does NP equal $co-NP$? Put another way do tautologies always have "short" proofs? If proof is taken in its most general form, i.e. does some nondeterministic polynomial time Turing Machine correctly accept exactly the class of tautologies, then the question seems to be completely beyond our reach. However, recently there has been considerable progress on attacking a restricted version of this problem. The method is to restrict attention to "reasonable" types of proof systems, and show lower bounds on the size of proofs for tautologies.

One of the tools for proving lower bounds on proof sizes is as follows: Suppose that a tautology $\mathcal{T}$ has a short proof in some system. Then, sometimes one can convert this into a bound on the size of an automaton that accepts a language $\mathcal{L}$ that corresponds to the tautology $\mathcal{T}$. Since lower bounds for certain weak automata are known, this yields a contradiction. For example, the proof system regular resolution corresponds directly to the class of read-once branching programs [Kra95]. Thus, the known technology for proving lower bounds on read-once branching programs ([AM86, SS93]) can be brought to bear on lower bounds for resolution [RWY97].

Our idea is to reverse this correspondence: proof system → automata class. In our case we plan to look at the reverse correspondence: automata class → proof system. Roughly, in a natural way given an class of automata we can associate with it a proof system. The key is that we have to *restrict* the automata class so that the proof system lies in NP.

One of the most interesting new proof systems that we get in this way is based on the class of push down automata (*pda*). Our correspondence shows that there is a proof system in 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 essentially corresponds to finite state automata. However, there are several complications: First, as we stated earlier the language that is used must be a language that encodes a tautology. This requires a somewhat unusual language. Also, We 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, [BC82]) than classic automata.

The definition of the new proof system gives rise to many interesting questions. What is the power of the *pda* based proof system ? Is there some natural generalization of regular resolution that can be seen to be equivalent to the *pda* system ? It is known for example that *BDD*'s cannot compute integer multiplication in polynomial size [Pon95]. Is it possible to solve the same problem in the new *pda* based proof system in polyno-

*Computer Science Department, 35 Olden st, Princeton, NJ 08544, and Bellcore Research (Telcordia). rjl@cs.princeton.edu. Supported in part by NSF CCR-9633103

†Computer Science Dept., 35 Olden st, Princeton, NJ 08544, viglas@cs.princeton.edu

mial size ? The techniques used in the lower bound proof for *BDD*'s in [Pon95] do not seem to generalize to *pda* machines.

It is also interesting to notice that in some sense, the *pda* model moves beyond read-once machines; it is easy to see that a *pda* machine can simulate a *weak read twice* branching program that reads xx^R. A lower bound for the *pda* proof system would be an interesting generalization of existing results.

The exponential separation between the *pda* based proof system and oblivious regular resolution presented in this work, assumes that the clauses we consider are "fsa" type formulas: these are simply formulas that can be computed easily by an *fsa* for any ordering of their variables. The lower bound for oblivious regular resolution is implied by a lower bound for oblivious branching programs, that uses a technique of Alon and Maass [AM86].

2 Preliminaries

Many results have been established recently for restricted computation models and especially for branching programs, but proving lower bounds for general unrestricted models seems to be far beyond current knowledge.

Branching programs are graph representations of discrete functions. A branching program B for a function $f : \Sigma_1^n \to \Sigma_2$, is a directed acyclic graph, with one node of in-degree 0 (source node) and (up to) $|\Sigma_2|$ sink nodes. All non-sink nodes are labeled with a name of an input variable of the function $f = f(x_1, \ldots, x_n)$, and the number of outgoing edges is $|\Sigma_1|$. 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 x the output of the function $f(x)$ is given by the label of the sink node we will reach, after following the computation path designated by the values of the input variables: start at the source, and on each node of the branching program labeled by x_i follow the edge labeled by the value of the i-th bit of x. The previous definition describes the *R-way branching programs* proposed by Borodin, Cook [BC82], where $R = |\Sigma_1|$.

For our purposes we can assume that the vertices of the branching program are arranged in levels, such that edges connect nodes in consecutive levels only. 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 each 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 *oblivious* if the variables are tested in the same order along each path. Note that it is an open question whether branching programs are more powerful than the oblivious version.

Some known results about branching programs are listed next: Neciporuk proved a strong lower bound for the size of an unrestricted branching program computing a function that lies in NP in [Nec66]. Also Babai et al. in [BPRS90] proved a super-linear lower bound for the majority function (Pudlak also proves a non-trivial bound for majority function in [Pud84]). Haken in [Hak85] has shown that the pigeonhole principle requires exponential size resolution proofs (see also [BT88, BP96]).

Many lower bounds and general techniques are known for restricted forms of branching programs and mostly for oblivious and read-once branching programs: exponential lower bounds are known for computing the parity of the number of triangles in a graph [ABH$^+$86, BHST87, SS93], size of regular resolution proofs for the pigeonhole principle [Hak85, RWY97], integer multiplication [Pon95], for the clique-only function [BRS93, Weg87], the k-regularity problem for a graph [BHST87, SS93] and many other explicit functions [Gál97, ABH$^+$86, BHST87, Dun85]. Recent work of Thathachar [Tha98] proves an exponential separation between consecutive levels of the hierarchy of read-k-times branching programs. For a survey of lower bound results for branching programs see [Raz91].

3 Definitions and Lemmata

As we mentioned in the introduction, in order to prove a 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. One way to formalize this idea, is to assume that the input clauses are "fsa" formulas. We will define *fsa* formulas to be formulas whose truth value can be computed by a polynomial size *fsa*, regardless of the ordering in which the input variables are read. In other words, an *fsa* formula 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 π of the variables x. We define the *fsa* size of a formula to be the following measure: $||\phi||_{fsa} = \max_\pi\{$ smallest *fsa* accepting $x_{\pi_1}, \ldots, x_{\pi_n}$ for which $\phi(x_1, \ldots, x_n) = 1\}$ where $x_{\pi_1}, \ldots, x_{\pi_n}$ is a permutation π of the variables.

Definition 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.

For this model, we can show that a proof system based on oblivious read-once branching program machines (and therefore oblivious regular resolution) is strictly less powerful than the proof system of push down *BDD*'s. We will prove this separation using the meander lemma from [AM86]. If we allow the input to be ordinary clauses, then it is open if the push down model has greater power.

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 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 $\mathcal{C}$. Note that for our purposes we will assume 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.

Definition 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* $\mathcal{C} = \bigwedge_{i=1}^{k} C_i$*, if there exists a machine* $M \in \mathcal{M}$ *such that, for a given input* $\bar{x}$*,* M *can find and output a clause from* $\mathcal{C}$ *that is false: (1)* $M(\bar{x}) = i \Longrightarrow C_i(\bar{x}) = false$ *and (2)* $M(\bar{x}) = i \Longrightarrow i \in \{1, 2, \ldots, k\}$*.*

In order to ensure that our proof systems lie in NP we must consider classes of finite machines for which the correctness of the computation can be checked in polynomial time. Checking the correctness of the computation of a machine M from a class of automata $\mathcal{M}$, involves verifying the two properties of definition 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 $\bar{C}_i$ nonempty ? 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 3.3 *If* $\mathcal{M}$ *is any class of machines and* $M \in \mathcal{M}$*, the correctness of the computation of* M *can be checked in time polynomial in the size of* M*, if the following hold: (1) The class* $\mathcal{M}$ *is closed under Cartesian product with polynomial size finite state automata: that is* $(M \times F) \in \mathcal{M}$ *for any* $M \in \mathcal{M}$ *and polynomial size* fsa F*, and also we can compute the machine* $M \times F$ *in polynomial time. (2) The emptiness problem for any* $M \in \mathcal{M}$*, can be solved in time polynomial in the size of* M*.*

Proof 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 $C_i(\bar{x}) = false$.

The language $L(M)$ 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 $M^i = (M \times \bar{C}_i \times F_n)$, where F_n denotes an *fsa* that accepts all strings of length n. If the class $\mathcal{M}$ is closed under Cartesian product with a polynomial size *fsa*, then $M \in \mathcal{M} \Rightarrow M^i \in \mathcal{M}$. Observe that if $L(M^i)$ is non-empty and $\bar{x} \in L(M^i)$ then: (a) $|\bar{x}| = n$, (b) $\bar{x}$ is accepted by $\bar{C}_i$ and therefore $C_i(\bar{x})$ is false and (c) if $M(\bar{x}) = i$ then indeed $C_i(\bar{x}) = false$. 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 NP. ∎

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 poly-size *fsa*'s is also a poly-size *fsa*. In the same way, branching programs, binary decision diagrams and so on are also classes that imply proof systems that lie in NP.

We can prove that for push down automata, correctness is also checkable in time polynomial in the size of the machine: the class of (polynomial size) *pda*'s is closed under Cartesian product with a poly-size *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.

Theorem 3.4 *The emptiness problem for a push down automaton with* n *states can be decided in time polynomial in* n*.*

Proof (Outline) The proof 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 check 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 for context free grammars (see [HU79]). It is easy to see that the conversion to a context free grammar is polynomial in n and also, the emptiness algorithm for context free grammars runs in time polynomial in n. ∎

We will consider a class of machines based on *BDD*'s that read their input once, but have access to a push-down stack:

Definition 3.5 *Let $X = (x_1, \ldots, x_n)$ denote the set of input variables, over the alphabet Σ. A push down* BDD *is defined by a directed acyclic graph $G = (V, E)$ combined with a push-down stack; $v_0 \in V$ is the source node (of in-degree 0), $F \subseteq V$ is the set of sink nodes. Every node is labeled by an input variable and every edge is labeled according to the transition function $\delta : V \times \Sigma_\epsilon \times \Gamma_\epsilon \to V \times \Gamma_\epsilon$ where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$, $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$, and Γ is the stack alphabet.*

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 on the top of the stack (the out degree of all non-sink nodes in the graph G can be $|\Sigma| \cdot |\Gamma|$). Also, the push-down *BDD*'s have the same structure as the usual push-down automata, but they can "choose" any (fixed) order 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 3.4 and lemma 3.3 we conclude the following:

Corollary 3.6 *The push-down* BDD *proof system lies in NP.*

We also mention, very briefly, the main lemma of Alon and Maass from [AM86]. Let $X = x_1 x_2 \ldots x_m$, $x_i \in \{1, 2, \ldots, n\}$, and $S, T \subseteq \{1, 2, \ldots, n\}$. An interval $x_i x_{i+1} \ldots x_{i+j}$ is called 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 $x_{i+j} \in S$.

Lemma 3.7 (Meander Lemma) *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.*

4 The Lower Bound for the branching program model

Proving lower bounds for the length of proofs in some proof system, can be achieved by proving bounds for the size of an automaton that corresponds to the considered proof system. We will prove an exponential lower bound for oblivious branching programs deciding a language that requires only polynomial size push-down *BDD*'s.

Define the language $PH\mathcal{L}_n$ (Pigeonhole language) as follows: Consider strings over the alphabet $\{0, 1, \circ\}$, where $\circ$ is a symbol that can be ignored by our machines (we will call it "skip" symbol). 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 $PH\mathcal{L}_n$ 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 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 $PH\mathcal{L}_n$: 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 $\neg PH\mathcal{L}_n$ 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 $\{x_i | i < r\}$ and $\{y_i | i < s\}$ is the same.

Theorem 4.1 *A push-down* BDD *can recognize the language $PH\mathcal{L}_n$ in time polynomial in n (and the size of the push-down* BDD *is also polynomial in n).*

Proof To check whether $(x, y) \in PH\mathcal{L}_n$: Read x from right to left. For each symbol $x_r \neq \circ$, push x_r on the stack. Then 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 $(x, y) \in PH\mathcal{L}_n$. ∎

In the branching program model, it is not possible to solve the search problem associated with $PH\mathcal{L}_n$ in polynomial size. Note that if we do not use the symbols

'$\circ$' in our definition of $PH\mathcal{L}_n$, 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 4.2 *Any (3−way) oblivious (read−once) branching program of width* $2^{n/2^{h(n)}}$ *solving the search problem associated with the language* $PH\mathcal{L}_n$*, 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: $\alpha_i = j$ if the i-th level vertices of the branching program are labeled by x_j or $\alpha_i = n + j$ if they are labeled by y_j. 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 3.7 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 matching positions. If we consider an instance $I_{A'B'} =< z'_1, \ldots, z'_{2n} >$, such that $A \neq A'$ then 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 $I_{A'B}$ (for any A'). 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 the *BDD* is $2n$ and therefore the exponential lower bound for $PH\mathcal{L}_n$ follows.

5 Discussion and Open Problems

Proving the separation for ordinary clauses seems to be more difficult. The clauses for $PH\mathcal{L}_n$ must express that $(x, y) \in PH\mathcal{L}_n$ iff $x_i = y_j = 1$ and the number of non-skip symbols 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). A different definition of the language $PH\mathcal{L}_n$ is needed in order to prove the more general result.

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.

An important problem for the push-down *BDD* model is to prove an exponential lower bound for the size of the proof of some special kind of tautology (prove that the push down *BDD* proof system is not polynomial). 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 5.1 *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* $S^{O(1)}$

It is also interesting to see if well known techniques for proving lower bounds for branching programs or resolution systems, generalize for the push-down proof system. For example, counting the number of subfunctions is a combinatorial method (see [SS93, Gál97]) which does not seem to generalize to prove lower bounds for the push down system[1].

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 resolution or to other well known proof systems.

References

[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 *18th ACM Symposium on Theory of Computing*, pages 30–38, 1986.

[1] If we consider the language $PH\mathcal{L}_n$ we can see that for any subset of the input variables of size k the number of subfunctions is 2^k but $PH\mathcal{L}_n$ has polynomial size proofs in the push down system.

[AM86] Noga Alon and Wolfgang Maass. Meanders, Ramsey theory and lower bounds for branching programs. In *27th Symposium on Foundations of Computer Science*, pages 410–417. IEEE, 1986.

[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.

[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, 1987.

[BP96] Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *37th Annual Symposium on Foundations of Computer Science*, pages 274–282, Burlington, Vermont, 14–16 October 1996. IEEE.

[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, 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.

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

[Dun85] Dunne. Lower bounds on the complexity of 1-time only branching programs. *Fundamentals of Computation Theory*, 5, 1985.

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

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

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

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

[Nec66] Neciporuk. A boolean function. *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 *27th ACM Symposium on Theory of Computing*, pages 130–139, 1995.

[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.

[RWY97] Alexander Razborov, Avi Wigderson, and Andrew Yao. Read-Once Branching Programs, Rectangular Proofs of the Pigeonhole Principle and the Traversal Calculus. *29th ACM Symposium on Theory of Computing, STOC'97*, pages 739–748, 1997.

[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 S. Thathachar. On separating the read-k-times branching program hierarchy. In *30th ACM Symposium on Theory of Computing (STOC-98)*, pages 653–662, 1998.

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