On verifying proofs in constant depth, and polynomial identity testing

By

Karteek Sreenivasaiah

MATH10200805003

The Institute of Mathematical Sciences, Chennai

A thesis submitted to the

Board of Studies in Mathematical Sciences

(Theoretical Computer Science)

In partial fulfillment of requirements

For the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

July, 2014

Homi Bhabha National Institute

Recommendations of the Viva Voce Board

As members of the Viva Voce Board, we certify that we have read the dissertation prepared by Karteek Sreenivasaiah entitled "On verifying proofs in constant depth, and polynomial identity testing" and recommend that it maybe accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

_____ Date:

Chair - V. Arvind

_____ Date:

Guide/Convener - Meena Mahajan

_____ Date:

Member 1 - Vikram Sharma

_____ Date:

Member 2 (External Examiner) - Prahladh Harsha

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to HBNI.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it may be accepted as fulfilling the dissertation requirement.

Date:

Place: Guide

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at Homi Bhabha National Institute (HBNI) and is deposited in the Library to be made available to borrowers under rules of the HBNI.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permisiion for extended quotation from or reproduction of this manuscriptin whole or in part may be granted by the Competent Authority of HBNI when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

(Karteek Sreenivasaiah)

DECLARATION

I hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree / diploma at this or any other Institution / University.

(Karteek Sreenivasaiah)

ACKNOWLEDGEMENTS

I am very fortunate for having Meena as my advisor. Apart from her helpfulness and infinite patience, I thank her most for showing me how to think mathematically about problems. This has not only helped me convert high-level ideas into rigorous proofs, but has also allowed me to understand other results at a much deeper level than I did before. I also thank her for putting up with my unusual working hours.

I thank all my co-authors for the wonderful discussions and advice through the last four years. A special thanks to Andreas, Raghu and Nutan for being fantastic to work with, and for making me feel like I was at home whenever I visited them. I also thank Kristoffer, Markus, Olaf and Rahul for being excellent hosts and for giving me an opportunity to work and share ideas with them.

I thank all the computer science faculty in IMSc for their high quality teaching and designing the excellent coursework for my masters degree.

I would like to thank my parents and sister for supporting me throughout my PhD and for actively encouraging me to take up research after my undergraduation.

I thank all my friends for a very memorable time here in IMSc. A special thanks to Maitreyi, Yadu, Rohan, Beli, Madhusree, Sajin and Nitin for all their help and support.

I thank the IMSc adminstration and library staff for providing an excellent research environment. A special thanks to the technical staff in the computers section for sorting out every problem on my machine within an hour of filing the complaint.

Finally, I thank my dog “Shadow” and my cat “Tiger” for their unconditional love and unconditional hate respectively.

Contents

Synopsis

The thesis is divided into two main parts. The first part deals with proof systems computable by Boolean circuit families that characterize the complexity class NC^0 (bounded fanin, constant depth), which is one of the weakest complexity classes. A proof system for a language L is a function $f : \{0,1\}^* \to \{0,1\}^*$ such that Range$(f) = L$. We initiate a study of NC^0 computable proof systems with an overarching goal of showing that such proof systems cannot capture the language TAUT. We begin by studying NC^0 proof systems in the context of regular languages. We give sufficient conditions for a regular language to have a proof system computable in NC^0. On the other hand, we show that an explicit regular language does not have a proof system computable in NC^0. By generalizing techniques used in constructing proof systems for regular languages, we construct NC^0 proof systems for languages complete for various complexity classes ranging from NC^1 to P. It remains open to characterize the regular languages that indeed have proof systems that are computable in NC^0. In the context of TAUT, we study 2TAUT and show a reduction from 2TAUT to the language associated with directed connectivity in terms of proof systems. We show that the set of all undirected graphs that have a path between two fixed vertices s and t has an NC^0 proof system. Our study shows that the question of whether a language can be generated using these restricted proof systems is unrelated to the computational complexity of their associated membership problem.

In the second part of the thesis, we study the problem of testing if a given arithmetic circuit computes the identically zero polynomial (PIT) and give efficient algorithms for certain special cases. We also determine the complexity of other natural problems that arise in the context of arithmetic circuits. We give a multilinearity and identity test for read-thrice formulas. We then give efficient algorithms for PIT on polynomials of the form $f_1 f_2 f_3 \cdots f_m + g_1 g_2 \cdots g_s$ where f_is and g_is are presented as read-once formulas. We show a hardness of representation for the elementary symmetric polynomial against read-once formulas with the added restriction that every leaf is labeled ax where a is a non-zero field element and x is a variable. Finally, we study some natural problems in the context of arithmetic circuits. These include counting the number of monomials, and checking

if a given monomial has non-zero coefficient in the polynomial computed by a given arithmetic circuit.We observe that even for monotone (no negative constants) read-twice formulas, counting the number of monomials is #P-hard. We also show that checking if the coefficient of a monomial is zero in a polynomial computed by a read-once formula is in logspace.

List of Figures

Chapter 1

Introduction

The thesis consists of two parts. In the first part, we study proof systems that can be computed by very little resources. In the second part, we look at the problem of testing if the polynomial computed by an input arithmetic circuit is identically zero. The background and motivation for each of the parts together with a summary of our results is described below.

1.1 Proof systems computable in NC^0

1.1.1 Background and Motivation

The complexity class NP is the set of all languages that have short proofs that can be efficiently verified. More precisely, $L \in \mathsf{NP}$ if there exists an efficient algorithm $\mathcal{A}$ such that for every string x, there exists a string y, $|y| = O(|x|^c)$ such that $x \in L \iff \mathcal{A}(x, y) = 1$. For example: SAT - The language of all satisfiable propositional formulas. A proof that a propositional formula F is indeed satisfiable can be an assignment $\vec{a}$ to the variables in F that satisfies it. The verifier algorithm just needs to check if indeed $\vec{a}$ is a satisfying assignment and this can be done in polynomial time. An interesting fundamental question is: For some formula F that is not satisfiable, is there a short proof of $F \notin$ SAT that can be verified efficiently? Equivalently: Are there efficient procedures and short certificates to prove that a propositional formula is a tautology? This forms the basis of the question: Is NP = coNP? It is in this context that the notion of proof systems was first introduced by Cook and Reckhow in [CR79].

A computable function $f : \{0,1\}^* \to \{0,1\}^*$ is a proof system for L if $\text{Range}(f) = L$.

From the fact that TAUT is coNP complete, it follows that NP = coNP if and only if there exists a polynomial time computable proof system for TAUT with short proofs (See [Coo71]). Hence, one way of showing NP ≠ coNP is to show that for every polynomial time computable proof system f for TAUT, there exists a family of tautologies such that f needs inputs of super polynomial length to produce members of this family. As of now, we do not know how to show even a super linear lower bound on proof length.

Since we do not yet know whether or not NP equals co-NP, a reasonable question to ask is how much more computational power and/or proof length is needed before we can show that TAUT has a proof system. For instance, allowing the verifier the power of randomized polynomial-time computation on polynomial-sized proofs characterizes the class MA; allowing quantum power characterizes the class QCMA; one could also allow the verifier access to some advice, yielding non-uniform classes; see for instance [Hir10, HI10, CK07, Pud09].

An even more interesting, and equally reasonable, approach is to ask: how much do we need to reduce the computational power of the verifier before we can formally establish that TAUT does not have a proof system within those bounds? This approach has seen a rich body of results, starting from the path-breaking work of Cook and Reckhow [CR79]. Haken showed in [Hak85] that any proof for the pigeon hole principle (PHP) in the Resolution proof system requires exponential length. This was followed by a result by Ajtai in [Ajt94] showing that PHP needs super polynomial length proof for AC^0–Frege proof systems. Krajicek et al. in [KPW95] improved Ajtai's result to an exponential lower bound for length of proofs in AC^0–Frege systems for PHP. The common theme in these results is limiting the verifier's power by restricting the nature of proof verification, equivalently, the syntax of the proof. See [BP01, Seg07] for excellent surveys on the topic.

Instead of restricting the proof syntax, if we only restrict the computational power of the verifier, it is not immediately obvious that we get anywhere. This is because it is already known that NP is characterized by short proofs with verifiers computable by uniform AC^0 circuit families. AC^0 is the family of circuits over $\{\wedge, \vee, \neg\}$ that have constant depth and unbounded fanin. With regard to computing Boolean functions, AC^0 is a very weak class of circuits. For instance, in AC^0 we cannot even check if a binary string has an odd number of 1s [Ajt83, FSS84, Hås86]. This handicap of AC^0 does not seem to matter when being used to compute proof systems. The fact that every language in NP has an AC^0 proof system means that NP = coNP if and only if TAUT has a proof system computable in uniform AC^0. Hence, it is natural to ask what happens when we restrict verifiers to be computed by a class even weaker than AC^0. Hence, we go even below AC^0 and ask: Is there a proof system for TAUT that can be computed in uniform NC^0? The

class NC^0 consists of circuits over $\{\wedge, \vee, \neg\}$ that have constant depth and all gates have bounded fanin. These circuits constitute one of the weakest computational models in computational complexity. In an NC^0 proof system, each output bit depends on just $O(1)$ bits of the input, so to enumerate L as the range of an NC^0 function f, f must be able to do highly local corrections to an alleged proof while maintaining the global property that the output word belongs to L. Unlike locally-decodable error-correcting codes, the correction here must be deterministic and always correct. We also note that unlike verifiers that output a Boolean value based on the veracity of the input proof provided, we study proof systems that are expected to output a valid theorem for any proof given as input. It is easy to see that if the amount of computational resource allowed is at least that of AC^0, then these two notions coincide.

Since NC^0-computable proof systems are functions which shrink the input by at most a constant factor, every language with an NC^0 proof system is computable in nonuniform nondeterministic linear time.

A related line of research studies NC^0-computable functions in a cryptographic context [Hås87, AIK06, AIK08, CM01, MST06]. One of the main problems in this area is to construct pseudorandom generators which are computed by NC^0 circuits [AIK06, AIK08, CM01, MST06]. This question asks for NC^0-computable functions for which the range is hard to distinguish from a uniform distribution. In another thread [Vio11, Vio12], for a function f, one is interested in the uniform distribution over $\langle x, f(x)\rangle$. Sampling exactly from this distribution may be possible even if computing $f(x)$ is hard. However, it is shown that for some functions (such as detecting exact-Hamming-weight αn), getting even close to the uniform distribution via specific types of NC^0 circuits (d-local circuits) is not possible. In contrast, we are looking here at the related, but possibly easier problem of understanding which sets can appear at all as the range of NC^0-computable functions, irrespective of the distribution on the support.

Recall that we do not know a family of tautologies for which we can show even a super linear lower bound on proof length against polynomial time computable proof systems. Such a super linear lower bound would imply that TAUT is not in NTIME($O(n)$). If $\mathsf{NP} \neq \mathsf{coNP}$, then clearly TAUT is not in NTIME($O(n^k)$) for any constant k. Compared to showing that TAUT $\notin$ NTIME($O(n)$), it should be easier to show that TAUT does not have a uniform NC^0 proof system because NC^0 is a further restriction. Hence showing lower bounds for NC^0 computable proof systems is a possible starting point towards finding super linear lower bounds against general proof systems. This further justifies studying the NC^0 restriction.

Is NC^0 too strong a restriction? To answer this, we note that Cryan and Miltersen [CM01]

exhibit an NC^0-computable function whose range is NP complete. Thus, there are NP-complete languages that admit an NC^0 proof system. This suggests that the NC^0 restriction, in the context of proof systems, does not render the proof systems completely useless. An NP-complete language having an NC^0 proof system raises another natural question: Do we need the full power of AC^0 to construct proof systems for languages in NP? We observe that the AC^0 computable proof systems mentioned earlier for languages in NP have exactly one unbounded fanin $\wedge$ gate. If we disallow this unbounded fanin gate, how much of NP can we still capture? Disallowing the only unbounded fanin gate is equivalent to restricting the computational power of the verifier to NC^0 circuit families. Note that the non-deterministic time hierarchy theorem implies that there are languages in NP that do not have NC^0 proof systems. But this does not give us an explicit example.

1.1.2 Our results

Our aim is to study proof systems computable in NC^0, understand their capabilities, and discover connections (or the lack of them) to computational complexity classes. From here on, we use the term "NC^0 proof system" to mean a proof system that is computable by an NC^0 circuit family.

In Chapter 2, we give an introduction to constructing NC^0 computable proof systems. We initially focus on regular languages and develop ideas that we use later to construct NC^0 proof systems for a variety of natural problems. We start by showing that any regular language that has deterministic finite automata (DFA) with at most two states has an NC^0 proof system. Following this we show, in Theorem 2.9, a regular language that has a DFA with only 3 states but does not have even non-uniform NC^0 proof systems. We do this by looking at circuit structure and a careful counting argument to show a lower bound of $O(\log \log n)$ on the depth of any bounded fanin proof system generating this regular language. Thus two natural questions arise: (1) What regular languages have proof systems computable in NC^0? (2) How much computational power suffices to compute proof systems for all regular languages? We partially answer question 1 in Section 2.2.2 by giving sufficient conditions for a regular language to have a proof system computable in NC^0. We exhibit a general construction for NC^0 proof systems which works for all regular languages that are accepted by a strongly connected NFA. Our construction directly transforms this NFA into an NC^0 proof system.

We explore some closure and non-closure properties of NC^0 proof systems in 2.2.3 before showing that for every regular language, we can construct a proof system computable by circuits of bounded depth and poly log fanin in the output size (Theorem 2.29). We

refer to the class of circuits of bounded depth and poly log fanin in the output length as poly log AC^0. Hence Theorem 2.29 essentially answers question 2 stated above. We also observe that due to Theorem 2.9, the upper bound we obtain in Theorem 2.29 is asymptotically tight.

In Section 2.3.1, we give constructions for many non-regular languages. Among them are an NC^1 complete language (Proposition 2.30) and a P complete language (Proposition 2.34). We explore poly log AC^0 construction for proof systems in Section 2.3.2. We generalize the idea from proof of Theorem 2.29 and show a similar construction of poly log AC^0 for languages that have branching programs with certain properties. We push this technique further to show a poly log AC^0 proof system for languages corresponding to the 1 instances of threshold functions (Theorem 2.38).

We devote Chapter 3 to showing lower bounds. We start by generalize the counting argument used in Theorem 2.9 and show that languages such as 0^*1^*, Exact-Count^n_k and Threshold^n_k functions do not have NC^0 computable proof systems for certain values of n and k. Although we show lower bounds for more regular languages, it remains open to characterize the regular languages that indeed have proof systems that are computable in NC^0.

The simple counting argument we use in the proof of Theorem 2.9 fails for the language Majority - the set of all strings over $\{0, 1\}$ that have at least as many 1s as 0s. We devote the whole of subsection 3.3 to showing that Majority does not have proof systems computable in NC^0. We notice that the reason the previous counting argument does not work for Majority is because of the presence of input bits that could influence the value of a large number of output bits. We try to work around these "high-influence" input bits by hardwiring them to well chosen values such that we obtain a circuit that still generates a non trivial part of Majority. We do this iteratively till we are finally left with a small number of input bits, which are not hardwired, that generate a large number of possible output strings. We then argue that this is not possible unless the depth of the circuit is $\omega(1)$. We first give an intuitive idea of the proof in subsection 3.3 and then proceed to a full formal proof under Theorem 3.6. We use this to show Corollary 3.16 that the language of all pairs of graphs that are isomorphic to each other does not have an NC^0 proof system.

Given that there exists a regular language that does not admit an NC^0 proof system (Theorem 2.9) while there is an NP-complete language that does ([CM01]), the class of languages that admit an NC^0 computable proof system slices vertically across complexity classes. In other words, the complexity of generating exactly the strings of a language L and the complexity of deciding if a string belongs to L seem to be completely unrelated

to each other. This makes the study NC^0 computable proof systems even more intriguing.

In Chapter 4, we return to the question of whether TAUT has a proof system computable in NC^0. We believe that TAUT does not have NC^0 proof systems because otherwise $\mathsf{NP} = \mathsf{coNP}$. Although it seems like showing that no NC^0 proof system can capture TAUT should be easier than showing $\mathsf{NP} \neq \mathsf{coNP}$, we have not yet succeeded. So we ask the same question about 2TAUT - the set of all 2DNFs that are tautologies. The language 2TAUT is in non-deterministic logspace (NL), and hence may well have an NC^0 proof system. The standard NL algorithm for 2TAUT is via a reduction to the problem of $s-t$ connectivity STCONN in a related implication graph. So it is interesting to ask – does STCONN have an NC^0 proof system? We do not know yet. We first show, in Theorem 4.13, that a "reduction" from 2TAUT to STCONN exists in the proof systems context as well. i.e., we show that if STCONN has an NC^0 computable proof system, then so does 2TAUT. As a main result of Chapter 4, we show that USTCONN - the undirected analogue of STCONN, a language complete for L (logspace), has an NC^0 proof system. The idea is to look at an $s-t$ path as an exclusive-or (EXOR) of the (s,t) edge and a simple cycle involving s and t. We first demonstrate the proof technique on the special case of undirected grid graphs (Theorem 4.11). We then use the same high level idea to get Theorem 4.13. Our construction relies on a careful decomposition of even-degrees-only graphs (established in the proof of Theorem 4.14) that may be of independent interest.

We end Chapter 4 by partially answering the question of how much computational power do proof systems need to generate the languages in NP. i.e., do we need all of AC^0 to compute proof systems for languages in NP? For instance, by further restricting the fanin of $\wedge$ gates ($\vee$ gates)in an AC^0 circuit family to be bounded, we obtain the class SAC^0 (coSAC^0). Can we capture all of NP with these restricted classes? We show in Theorem 4.19 that there is a language A in NP such that any proof system for A will need to be at least as powerful as SAC^0 or coSAC^0. On the other hand, we show that languages from a large fraction of NP have proof systems that can be computed by SAC^0 or coSAC^0 circuit families.

In conclusion, the capabilities and weaknesses of NC^0 proof systems that we discover in our study show that the question of whether a language has an NC^0 proof system is unrelated to the computational complexity of their associated membership problem. The reason for this might be that proof systems computed by such restricted circuits are very sensitive to the encoding used. On the other hand, although NC^0 is a strong restriction on the circuit structure, showing that TAUT does not have proof systems computable in NC^0 seems quite difficult. In fact, we do not yet have a characterization of even regular languages that admit an NC^0 proof system. We show various languages that do not have

NC^0 proof systems, but essentially all of them are consequences of either Theorem 2.9 or Theorem 3.6. It is not clear if lower bound techniques which are used against restricted circuit classes (cf. [Weg87, Vol99]) are applicable to show lower bounds for NC^0 proof systems.

Most of the results in Chapters 2, 3 and 4 have appeared in [BDK+13] and [KLMS13].

1.2 Polynomial Identity Testing and Arithmetic Formulas

1.2.1 Background and Motivation

The Polynomial Identity Testing (PIT) problem is the most fundamental computational question that can be asked about polynomials: is the polynomial given by some implicit representation identically zero? The implicit representations of the polynomials can be arithmetic circuits, branching programs etc., or the polynomial could be presented as a black-box, where the black-box takes a query in the form of an assignment to the variables and outputs the evaluation of the polynomial on the assignment. PIT has a randomized polynomial time algorithm on almost all input representations, independently discovered by Schwartz and Zippel [Sch80, Zip79]. However, obtaining deterministic polynomial time algorithms for PIT remained open since then. In 2004, Impagliazzo and Kabanets [KI04] showed that a deterministic polynomial time algorithm for PIT implies lower bounds (either $\mathsf{NEXP} \not\subset \mathsf{P/poly}$ or permanent does not have polynomial size arithmetic circuits), thus making it one of the central problems in algebraic complexity. Following [KI04], intense efforts over the last decade have been directed towards derandomizing PIT (see for instance [SY10, Sax14]). The attempts fall into two categories: considering special cases ([Sax14]), and optimizing the number of random bits used in the Schwartz-Zippel test [BHS08, BE11].

The recent progress on PIT mainly focuses on special cases where the polynomials are computed by restricted forms of arithmetic circuits. They can be seen as following one of the two main lines of restrictions: 1. Shallow circuits based on alternation depth of circuits computing the polynomial. 2. Restriction on the number of times a variable is read by formulas (circuits with fanout 1) computing the polynomial.

The study of PIT on shallow circuits began with depth two circuits, where deterministic polynomial time algorithms are known even when the polynomial is given as a

black-box [BT88, KS01]. Further, there were several interesting approaches that lead to deterministic PIT algorithms on depth three circuits with bounded top fan-in [DS07, KS07]. However, progressing from bounded fan-in depth three circuits seemed to be a big challenge. In 2008, Agrawal and Vinay [AV08] explained this difficulty, showing that deterministic polynomial time algorithms for PIT on depth four circuits implies sub-exponential time deterministic algorithms for general circuits. There have been several interesting approaches towards obtaining black-box algorithms for PIT on restricted classes of depth three and four circuits, see [Sax14, SY10] for further details. Recently, Gupta, Kamath, Kayal and Saptharishi [GKKS13] showed that, over infinite fields, deterministic polynomial time algorithms for PIT on depth three circuits would also imply lower bounds for the permanent.

A formula computing a polynomial that depends on all of its variables must read each variable at least once (count each leaf labeled x as reading the variable x). The simplest such formulas read each variable exactly once; these are Read-Once Formulas ROFs, and the polynomials computed by such formulas are known as read-once polynomials (ROP). In the case of an ROP f presented by a read-once formula computing it, a simple reachability algorithm on formulas can be applied to test if $f \equiv 0$. Shpilka and Volkovich [SV08] gave a deterministic polynomial time algorithm for PIT on ROPs given as a black-box. Generalizing this to formulas that read a variable more than once, they obtained a deterministic polynomial time algorithm for polynomials presented as a sum of $O(1)$ ROFs. Anderson et al. [AvMV11] showed that if a read-k formula, with $k \in O(1)$, is additionally restricted to compute multilinear polynomials at every gate, then PIT on such formulas can be done in deterministic polynomial time. Note that not all read-k multilinear polynomials can be expressed as a sum of $O(1)$ read-once formulas. The result by [AvMV11] subsumes the result in [SV08] since a k-sum of read-once formulas is read-k and computes multilinear polynomials at every gate. Both [SV08] and [AvMV11] crucially exploit the multilinearity property of the polynomials computed under the respective models.

PIT algorithms check whether the polynomial computed by the circuit has at least one monomial. Natural generalizations/variants of this question are (1) MonCount: compute the number of monomials in the polynomial computed by a given circuit, and (2) ZMC: Decide whether a given monomial has zero coefficient in the polynomial computed by a given circuit. ZMC was first studied by Koiran and Perifel [KP07]. More recently, Fournier, Malod and Mengel [FMM12a] showed that ZMC and MonCount characterize certain levels of the counting hierarchy (CH, the hierarchy based on the complexity classes PP and $\mathsf{C}_{=}\mathsf{P}$). In fact, MonCount remains hard even if restricted to formulas. They also show that if the circuits compute multilinear polynomials, then these problems

become easier (equivalent to PP and PIT respectively), and that multilinearity checking itself is equivalent to PIT. All these results from [FMM12a] are in the non-black-box model, where the circuit is given explicitly in the input.

Since PIT on Read-k formulas appears easier than on general read formulas, naturally one could ask whether MonCount and ZMC become easier as well.

1.2.2 Our results

In Chapter 5, we try to extend the results of [SV08] and [AvMV11] in the following two ways:

- We attempt to remove the multilinearity requirement in [AvMV11]. We give a multilinearity and identity test for read-twice and read-thrice formulas. Note that both Read-twice and Read-thrice formulas can compute non-multilinear polynomials. Our tests are intertwined with a PIT algorithm for subformulas. We give a deterministic polynomial-time algorithm that simultaneously decides whether a Read-2 Formula is multilinear and whether it is identically zero (Theorem 5.5). It uses the sum-of-k-ROFs test from [SV08] on some subformulas as well as on some formulas obtained by transforming subformulas of the input formula. Thus it is inherently a non-blackbox algorithm; so is the polynomial-time algorithm from [SV08]. In Theorem 5.6, we give a multilinearity and identity test for read-thrice formulas. However, this does not gather as much information as the test for read-twice. Hence we feel that the multilinearity test for the read-twice case can potentially be extended to formulas that read variables more than twice.

- We attempt to extend the class considered in [SV08] to the class of polynomials of the form $\sum_{i=1}^{k} f_i g_i$ where the f_is and g_is are presented as ROFs and k is some constant. These are read-$2k$ polynomials, not necessarily multilinear.

  Over the ring of integers and the field of rationals, we give an efficient deterministic non-blackbox PIT algorithm for the case $k = 2$; the polynomial is $f_1 f_2 + g_1 g_2$ where f_1, f_2, g_1, g_2 are all read-once polynomials presented by ROFs. This class can also be seen as a special case of read-4 polynomials. Our algorithm exploits the structural decomposition properties of ROPs and combines this with an algorithm that extracts greatest common divisors of the coefficients in an ROP. The algorithm easily generalises to polynomials of the form $f_1 f_2 f_3 \cdots f_m + g_1 g_2 \cdots g_s$ where f_is and g_is are presented as ROFs, but m, s can be unbounded; that is, the class $\sum^{(2)} \cdot \prod \cdot \mathsf{ROF}$. Note that this class of polynomials includes non-multilinear polynomials and also

polynomials with no bound on the number of times variables are read. Thus it is incomparable with the classes considered in [SV08], [AvMV11]. This result is presented in Section 5.3, Theorem 5.7. We also sketch a way to modify the proof of Theorem 5.7 to work for polynomials over any field (Theorem 5.11).

Central to the PIT algorithm in [SV08] is a "hardness of representation" lemma showing that the polynomial $\mathcal{M}_n = x_1 x_2 \cdots x_n$, consisting of just a single monomial, cannot be represented as a sum of less than $n/3$ ROPs of a particular form (weakly 0-justified). More recently, a similar hardness of representation result appeared in [Kay12]: if $\mathcal{M}_n$ is represented as a sum of powers of low-degree (at most d) polynomials, then the number of summands is $\exp(\Omega(n/d))$. As is implicit in [Kay12], such a hardness of representation statement can be used to give a PIT algorithm. We analyze this connection explicitly, and show that the results in [Kay12] lead to a deterministic sub-exponential time algorithm for black-box PIT for sums of powers of polynomials with appropriate size and degree (Chapter 5, Section 5.4, Theorem 5.15).

A minor drawback of both these statements is that they consider a model that cannot even individually compute all monomials. One would expect any reasonable model of representing polynomials to be able to compute $\mathcal{M}_n$. In Section 5.5, we consider the restriction of read-once formulas to *constant-free* formulas that are only allowed leaf labels ax, where x is a variable and a is a field element. This model can compute any single monomial. We show (Theorem 5.18) that the elementary symmetric polynomial $\mathsf{Sym}_{n,d}$ of degree d cannot be written as a sum of powers of such formulas unless the number of summands is $\Omega(\log(n/d))$. This appears weak compared to the $n/3$ bound from [SV08], but this is to be expected since unlike in [SV08] where the ROPs could only be added, we allow sums of powers. We also consider $0-\mathsf{justified}$ read-once formulas with alternation depth (between + and $\times$) 3, and obtain a similar hardness-of-representation result for the polynomial $\mathcal{M}_n$ against sums of powers of polynomials computed by such formulas, showing that $n^{\frac{1}{2}-\epsilon}$ summands are needed (Theorem 5.20). Again, this appears weak compared to the $\exp(\Omega(n/d))$ bound from [Kay12], but unlike in [Kay12] where the degree of the inner functions is a parameter, our inner ROPs could have arbitrarily high degree.

In Chapter 6, we study the problems of ZMC and MonCount. We observe that even for monotone (no negative constants) read-twice formulas, MonCount is #P-hard. This further leads us to the investigation: where exactly does hardness for MonCount and ZMC begin? Further, translating the classes between NP and PSPACE down to classes below P, can we show that on restricted circuits, MonCount and ZMC are complete for the translated classes?

Starting with ROFs, we show (Theorem 6.2) that MonCount for ROFs is in the GapNC^1-hierarchy, i.e., the AC^0-closure of GapNC^1, where GapNC^1 is the class of Boolean problems that can be computed by arithmetic formulas over the integers with constants 0, 1, -1. The GapNC^1-hierarchy is an intriguing class that lies between NC^1 and DLOG and has been studied extensively in the last two decades; see for instance [All04]. We also show that ZMC for ROFs is in logspace (Theorem 6.13). It is straightforward to see that ZMC for ROFs is hard for $\mathsf{C_=NC}^1$, so this is almost tight. (The "gap" between Boolean NC^1, $\mathsf{C_=NC}^1$, GapNC^1 and DLOG is very small.)

Another equally natural and well-studied restriction is when the circuit is an algebraic branching program BP with edges labeled by the allowed constants or by variables. Evaluation of BPs on Boolean-valued inputs is complete for the arithmetic class GapL, the logspace analogue of the class GapP. The GapL hierarchy (the AC^0 closure of GapL) is known to be contained in $\log n$ depth threshold circuits TC^1 and hence in $\log^2 n$ depth Boolean circuits NC^2. Two restrictions on BPs, in order of increasing generality, are: (1) occur-once BPs, or OBPs, where each variable appears at most once anywhere in the BP; these subsume ROFs, and (2) multilinear BPs, or MBPs, where the polynomial computed at every node is multilinear. Again, deterministic algorithms are known for PIT on OBPs, [JQS10]. We show that MonCount for OBPs is in the GapL hierarchy (Theorem 6.10), while ZMC for OBPs and even MBPs is complete for the complexity class $\mathsf{C_=L}$ (Theorem 6.12). (As a comparison, a well-known complete problem for $\mathsf{C_=L}$ is testing singularity of an integer matrix [ABO99].)

A related problem explored in [FMM12a] as a tool to solving MonCount is that of checking, given a circuit C and monomial m, whether C computes any monomial that extends m. Denote this problem ExistExtMon. Though our algorithms for MonCount do not need this subroutine, we also show that for OBPs (and hence for ROFs), ExistExtMon lies in the GapL hierarchy (Theorem 6.14).

Most of the results in Chapter 5 and 6 appear in [MRS14a] and [MRS14b].

Chapter 2

Constructions of small depth proof systems

In this chapter, we introduce NC^0 proof systems. We start with some basic definitions in section 2.1. In section 2.2, we show constructions of NC^0 proof systems for various languages starting from regular to P-complete. In section 2.2.2, we give sufficient conditions for a regular language to have a proof system computable in NC^0. In section 2.2.3, we show that for every regular language L, there is a circuit family of depth $(\log \log n)$ that computes a proof system for L. Finally, section 2.3 discusses NC^0 proof systems for some non-regular languages and also proof systems computable by bounded depth and polylogarithmic fanin circuit families.

2.1 Preliminaries

A Boolean circuit C is a labelled directed acyclic graph with one or more designated output gates. Each leaf of C is labelled by either a variable or a Boolean value and each internal node is labelled by one of $\{\vee, \wedge, \neg\}$. The internal nodes are also called "gates" of the circuit. Each gate in a Boolean circuit computes a Boolean function of its inputs in a natural inductive way. Since the number of leaves for any circuit is fixed, circuits are by definition a non-uniform model of computation.

A family $(C_n)_{n\geq 1}$ of Boolean circuits is a collection of Boolean circuits that work on inputs of various lengths. For formal definitions and notation regarding circuits and circuit families, see [Vol99].

A proof system for a language $L \subseteq \{0,1\}^*$ is a function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that Range$(f) = L$.

A family of Boolean circuits $(C_n)_{n\geq 1}$ is said to be a proof system for a language $L \subseteq \{0,1\}^n$ if it satisfies the following conditions:

1. Output length: For some function $m : \mathbb{N} \longrightarrow \mathbb{N}$, and for all $n \geq 1$,
   - If $L \cap \{0,1\}^n \neq \emptyset$, then $C_n : \{0,1\}^{m(n)} \to \{0,1\}^n$.
   - Otherwise C_n is empty.
2. Soundness: For all n where C_n is non-empty, and for all words $x \in \{0,1\}^{m(n)}$, $C_n(x) \in L$.
3. Completeness: For all $y \in L \cap \{0,1\}^n$, there is a word $x \in \{0,1\}^{m(n)}$ such that $C_n(x) = y$; we say that x is a *proof* of the theorem "$y \in L$".

That is, the circuit family has as its range exactly the set L. For some functions $s, d : \mathbb{N} \longrightarrow \mathbb{N}$, each C_n has size $s(n)$, depth $d(n)$.

We say a family of Boolean circuits $\mathcal{P} = (C_n)_{n\geq 1}$ is a proof system for a function $f : \{0,1\}^* \longrightarrow \{0,1\}$ if $\mathcal{P}$ is a proof system for the language $L = f^{-1}(1)$.

We will call each individual circuit C_n in a proof system $\mathcal{P}$ a "proof circuit". The circuit family $\mathcal{P}$ is said to be a constant-depth proof system if the AND and OR gates have unbounded fan-in, and for some constant d and for all n, $d(n) \leq d$. A constant-depth proof system of polynomial size (for some constant c and for all n, $s(n) \leq n^c$) is said to be an AC^0 proof system. Note that the size bound is non-standard: it is measured in terms of the output length, not input length.

The circuit family is said to be an NC^0 proof system if the AND and OR gates have bounded fan-in, and for some constant d and for all n, $d(n) \leq d$. This, along with the fan-in bound, implies $s(n) \leq cn$ for some constant c.

If a function $f : \{0,1\}^* \longrightarrow \{0,1\}$ has a proof system of a particular type (constant-depth, AC^0, NC^0), then we say that f admits a proof system of that type.

If the circuit family is *uniform*, then we say that the proof system is *uniform*. Here, a uniform circuit family is a family whose direct connection language, i.e., a language describing the structure (nodes, wires, gates types) of the circuits in the family, is decidable. If the direct connection language is decidable in polynomial time, then the family is said

to be P-*uniform*. If the language is decidable in logarithmic time, then the family is said to be DLOGTIME-*uniform*. (For more formal definitions, we refer the reader to [Vol99].)

We remark that all upper bounds of NC^0 proof systems in this chapter will yield DLOGTIME-uniform proof systems, unless explicitly stated otherwise.

For a language $L \subseteq \{0,1\}^*$, we say that L admits an NC^0 proof system, or that L is enumerable in NC^0, if its characteristic function χ_L admits such a proof system. In other words, there is an NC^0 circuit family which produces as output all the strings in L and no other strings. As before, if $C(x) = y$, then we view x as a *proof* that $y \in L$. Note that a string $y \in L$ can have multiple proofs x such that $C(x) = y$.

We observe that uniform AC^0 proof systems do exist for every NP set. In fact, a more general statement is true. (Here uniformity only refers to computable direct connection languages.)

Proposition 2.1 (Folklore)**.**

1. *Every language in* NP *admits a uniform* AC^0 *proof system.*

2. *More generally, every computable language admits a uniform constant depth proof system.*

3. *Even more generally, every language admits a constant-depth proof system.*

Proof. (Sketch.) For an arbitrary language L with characteristic function $\chi_L = f$, let n be a length where L is non-empty. Pick an arbitrary $w \in L^{=n}$. Restricted to length n, f is computed by a constant-depth Boolean circuit (possibly of size exponential in n) D. The circuit C_n extends D: If on input x, $D(x) = 1$, then output x otherwise output w. Clearly C_n is also constant-depth, and its range is exactly $L^{=n}$, proving (3).

Now let the computable language L be decided by the deterministic Turing Machine M. The run-time of M with respect to the length of the input can be assumed to be computable. Also a default word $w \in L^{=n}$ can be found effectively if it exists. A proof of a word $y \in L$ is an encoding of an accepting sequence of configurations of M on input y. The correctness of such a sequence of configurations can be checked locally, essentially in two consecutive configurations only three letters (around the head position on the tape) can be different. If our circuit reads an input that is not such an encoding, then it outputs some default value $w \in L$ of the appropriate length as above. All of this can be done by a constant depth uniform circuit, proving (2).

Finally, let L be accepted in polynomial time by the nondeterministic Turing machine M. Proceeding as above, the checking circuit is of size polynomial in the output word, proving (1). □

As mentioned already in the introduction, Cryan and Miltersen [CM01] exhibit an NP-complete language that admits even a uniform NC^0 proof system. But it is quite easy to see that this is not the case for every NP language. Indeed, as noted earlier, for any uniform NC^0 proof system, $m(n) \in O(n)$ and the circuits C_n are also of size $O(n)$; each bit of the output depends on $O(1)$ bits of the input proof. Thus if L has NC^0 proof systems, then strings in L have linear-sized proofs that are locally verifiable. This leads to the following observation, which will be considerably strengthened in Chapter 3.

Proposition 2.2. *There are non-trivial languages in* NP *that do not admit any* DLOGTIME*-uniform* NC^0 *proof system.*

Proof. If a language L has a DLOGTIME-uniform NC^0 proof system, then it can be recognised in NTIME(n): given an input y, guess the linear-sized proof x, evaluate the circuit $C_{|y|}(x)$, and verify that its output is y. But by the nondeterministic time hierarchy theorem [Coo73] we know that NP is not contained in NTIME(n). □

2.2 Proof systems for regular languages

In this section, we will give a gentle introduction to constructing proof systems computable in NC^0. We will start by constructing proof systems for certain simple regular languages and show possible generalizations wherever possible. The regular languages we consider will be on the alphabet $\{0, 1\}$ unless stated otherwise.

2.2.1 Simple examples

Consider regular languages accepted by automata with at most one state. There are only two possible languages - $\{0, 1\}^*$ and $\emptyset$. The first language has a trivial NC^0 proof system that takes as input a binary string, and outputs it without any change. The second language also has a trivial NC^0 proof system that is empty.

We now look at the languages accepted by deterministic finite state automata (DFA) with exactly two states:

1. L_{end1} - All strings ending with a 1.

2. L_{OR} - All strings with at least one 1.

3. $L_{\oplus}$ - All strings that have an odd number of 1s.

4. $L_{\text{end-odd}}$ - All strings that end with an odd number of 1s.

Constructing an NC^0 proof system for L_{end1} can be done by simply hardwiring the last output bit to be 1 and the rest of the output bits are just copied from the input. This immediately generalizes to the following:

Lemma 2.3. *Let L be any language over a finite alphabet Σ and w a fixed string in Σ^*. Then L admits an NC^0 proof system if and only if $L \cdot \{w\}$ does.*

This can be further generalized to concatenation with finite sets: For any two languages L_1 and L_2, define the language $L_1 \cdot L_2 = \{w_1 w_2 | w_1 \in L_1 \wedge w_2 \in L_2\}$.

Lemma 2.4. *Let S be any finite language and L be any language - both over a finite alphabet Σ. If L admits an NC^0 proof system then, the languages $L \cdot S$ and $S \cdot L$ admit an NC^0 proof system.*

Proof. Let $\mathcal{P} = (C_n)_{n>0}$ be an NC^0 proof system for L. Let $S = \{s_1, s_2, \cdots, s_k\}$ where k is a constant independent of n. For each $i \in [k]$, let $|s_i| = \ell_i$. To generate the strings of length n in $L \cdot S$, we use the proof circuits $C_{n-\ell_1}, C_{n-\ell_2}, \cdots, C_{n-\ell_k}$ along with an input m of length $\lceil \log k \rceil$ bits. We output the concatenation of the string output by $C_{n-\ell_m}$ and s_m. To generate strings of length n in $s \cdot L$, we output the concatenation of s_m with the output of $C_{n-\ell_m}$.

In essence, the construction is a multiplexer that choses the correct strings from $O(1)$ many choices to get a length n string as output. The choice of the multiplexer is based on the input m. Since k is a constant and each output bit of the circuits from $\mathcal{P}$ is a function of only $O(1)$ many input bits, we conclude that our construction indeed gives an NC^0 circuit family. □

We now show a construction for L_{OR}:

Proposition 2.5. *The language L_{OR} admits an NC^0 proof system.*

Proof. The circuit $C_n : \{0,1\}^{2n-1} \to \{0,1\}^n$ takes as input bit strings $a = a_1 \dots a_n$ and $b = b_1 \dots b_{n-1}$, and outputs a sequence $w = w_1 \dots w_n$ where

$$(\text{for } 1 \le i \le n) \quad w_i = \begin{cases} a_i & \text{if } (b_{i-1} \vee a_i) = b_i \\ 1 & \text{otherwise.} \end{cases}$$

Here for notational convenience we assume that $b_0 = 0, b_n = 1$. Notice that if each b_i correctly encodes the OR of the prefix $a_1 \dots a_i$, then $b_n = 1$ ensures that at least one $w_i = a_i$ is 1. Otherwise if there is ever a discrepancy between the b_i's and the prefix OR's of a_j's, we introduce a 1 at the first such w_i (and maybe others too); thus w is indeed in L_{OR}. Since for each string $a \in L_{OR}$ there is a correct string b with $b_0 = 0$, $b_i = b_{i-1} \vee a_i$ and $b_n = 1$, every string in L_{OR} is produced by C_n. Thus C_n is an onto map from $\{0,1\}^{2n-1} \to L_{OR} \cap \{0,1\}^n$ completing the proof. □

Now we look at the language $L_\oplus$. The idea used in the proof of Proposition 2.5 of interpreting the input as a string that gives the prefix *OR*s of the intended output can be used again. But this time we interpret the input as a string of prefix EXORs to obtain the following:

Proposition 2.6. *$L_\oplus$ admits an NC^0 proof system.*

Proof. The circuit $C_n : \{0,1\}^{n-1} \to \{0,1\}^n$ takes as input a bit string $a = a_1 \dots a_{n-1}$ and outputs a sequence $w = w_1 \dots w_n$ where

$$(\text{for } 1 \le i \le n) \quad w_i = a_{i-1} \oplus a_i$$

Here we will hardwire $a_0 = 0$ and $a_n = 1$. Hardwiring a_0 and a_n this way ensures that the output string always has an odd parity. This is because the parity of the output is just $(0 \oplus a_1) \oplus (a_1 \oplus a_2) \oplus (a_2 \oplus a_3) \cdots (a_n \oplus 1) = 1$. Notice that any string with odd number of 1s can be produced by just giving as input the string that holds the prefix parities. □

This idea of taking prefix parities generalizes to the word problem over finite groups. The word problem for a finite monoid M with identity e is (membership in) the language: $\{\langle m_1, m_2, \dots, m_n\rangle \in M^* : \prod_{i=1}^n m_i = e\}$. We assume here that for some constant c depending only on M, each element of M is described by a bit string of exactly c bits.

Proposition 2.7. *The word problem for finite groups admits an NC^0 proof system.*

Proof. We describe the circuit $C_n \colon \{0,1\}^{cn-c} \to \{0,1\}^{cn}$. (Since the word problem contains only words of lengths divisible by c, we produce circuits only for such lengths.) Given

the encoding of a sequence $g_1, \ldots, g_{n-1}$, and assuming that $g_0 = g_n = e$, C_n produces the sequence $\langle h_1, \ldots, h_n \rangle$ where $h_i = g_{i-1}^{-1} g_i$. Thus $\prod h_i = e$ and the word is indeed in the language. Conversely, every word $\langle h_1, \ldots, h_n \rangle$ in the language is produced by this circuit on input $g_1, \ldots, g_{n-1}$ where $g_i = \prod_{j \leq i} h_j$. □

In proving Proposition 2.7, we used all the three group axioms: associativity, existence of an identity and existence of inverses. However, for the earlier example of L_{OR}, the *OR* operation is associative and has an identity, but not all elements have an inverse. Yet we were able to show that the language L_{OR} has an NC^0 proof system.

Now we tackle the language $L_{\text{end-odd}}$. We construct an NC^0 proof system for this by asking for a proof that is closely connected to the DFA on two states that recognizes it:

Proposition 2.8. *The language $L_{\text{end-odd}}$ has an NC^0 proof system.*

Proof. We use automaton $D = \{Q = \{q_0, q_1\}, \Sigma = \{0, 1\}, \delta, F = \{q_1\}, q_0\}$ where q_0 is the start state and δ is defined as follows:

δ	0	1
q_0	q_0	q_1
q_1	q_0	q_0

We first observe that for every pair of states p, q in D, there is a string $w_{p,q}$ of length exactly 2 such that $\hat{\delta}(p, w_{p,q}) = q$. These strings are listed below:

	q_0	q_1
q_0	00	01
q_1	10	11

We construct proof circuit P that takes as input a proof x. The proof x is interpreted as blocks consisting of a word w_i of length 2 sandwiched between two states s_{i-1} and s_i. The proof is expected to have the property that for any i, $\hat{\delta}(s_{i-1}, w_i) = s_i$. We allow the last block to have a word of length 1 to accommodate for odd length strings in L. We hardwire $s_0 = q_0$ and $s_{\lceil n/2 \rceil} = q_1$ since q_0 and q_1 are the start and final state respectively. So to output strings of length n, the input string is of length $n + \frac{n}{2} - 1$.

The output string P is constructed as follows: For any i, if indeed $\hat{\delta}(s_{i-1}, w_i) = s_i$, then we output w_i as is. Otherwise, we output w_{s_{i-1}, s_i}.

The idea behind the construction is that the output is always consistent with the sequence of intermediate states provided in the proof.

Soundness: Follows from the fact that the last state is hardwired to q_1 and that every string we output is consistent with the state sequence in the proof.

Completeness: For any string $y \in L$, a proof that produces y would be a string that encodes the string y along with the intermediate alternate states in an accepting run of the automaton D on y.

Constant depth: Verifying if $\hat{\delta}(s_{i-1}, w_i) = s_i$ depends on only 4 input bits. The transition function can be implemented in constant size. Hence the construction gives an NC^0 circuit family. □

So far, we have shown NC^0 proof systems for regular languages that have DFAs of at most two states. Looking at languages with a DFA of exactly three states recognizing it, we already arrive at our first example of a language that does not admit NC^0 proof systems:

Theorem 2.9. *The language* Exact-Or^n *on n bits, that consists of all strings with exactly one 1, does not admit an* NC^0 *proof system.*

Proof. Suppose there is such a proof system, namely an NC^0-computable function $f : \{0,1\}^m \longrightarrow \{0,1\}^n$. Let $R_i \subseteq [m]$ be the proof bit positions that have a path to the ith output bit. For each i, there is at least one setting of the R_i bits that places a 1 in the ith bit of the output (producing the output string e_i). All extensions of this setting must produce e_i. Therefore $|f^{-1}(e_i)| \geq 2^{m-|R_i|}$. Let $c = \max_{i=1}^{n} |R_i|$; by assumption, $c \in O(1)$. Then for each $i \in [n]$, $|f^{-1}(e_i)| \geq 2^{m-c}$. But the $f^{-1}(e_i)$ partition $\{0,1\}^m$. Hence

$$2^m = \sum_{i=1}^{n} |f^{-1}(e_i)| \geq \sum_{i=1}^{n} 2^{m-c} = n2^{m-c}.$$

Therefore $c \geq \log n$, so $\exists i \in [n] : |R_i| \geq \log n$, a contradiction. □

Note that the above proof implies that any proof system for Exact-Or^n needs depth $\Omega(\log \log n)$. We will generalize the above proof to obtain lower bounds for other languages in Chapter 3.

We now describe some generic constructions and closures. They are easy to see, but we state them explicitly for later use.

Lemma 2.10. *If* $A, B \subseteq \{0,1\}^*$ *admit* NC^0 *proof systems, then so does* $A \cup B$.

Proof. Let the proof systems for A and B be witnessed by circuit families C' and C'', with proof lengths $m'(n)$ and $m''(n)$ respectively. We construct the circuit family C for $A \cup B$, with proof length $m'(n) + m''(n) + 1$, as follows: C_n consists of a copy of C'_n and a copy of C''_n, and has an input x for C', and input y for C'', and an extra input bit b. It outputs the string $(C'_n(x) \wedge b) \vee (C''_n(y) \wedge \bar{b})$ where the combination with b and $\bar{b}$ is done for each bit position. □

Note that in the above proof, the depth of the circuit for $A \cup B$ is two more than the maximum depth of the circuits for A and B. Since union is associative, a union of k sets can be expressed as a binary tree of unions of depth $\lceil \log k \rceil$. Thus the union of k languages, each with an NC^0 proof system of depth d, has an NC^0 proof system of depth $d + 2\lceil \log_2 k \rceil$. In particular, we get the following nonuniform upper bounds.

Lemma 2.11. *Suppose $L \subseteq \{0,1\}^*$ has the property that there is a constant k, and for each n, $|L \cap \{0,1\}^n| \leq k$. That is, at each length, at most k strings of that length are in L. Then L admits a nonuniform NC^0 proof system.*

We can show that the complement of the languages considered in Lemma 2.11 have an NC^0 proof system:

Lemma 2.12. *Suppose $L \subseteq \{0,1\}^*$ has the property that there is a constant k, and for each n, $|L \cap \{0,1\}^n| \geq 2^n - k$. That is, at each length, at most k strings of that length are not in L. Then L admits a nonuniform NC^0 proof system.*

Proof. The circuit C for $\mathrm{OR}^{-1}(1)$ outputs all strings except the string of all 0s. We first generalize this to exclude any fixed string y from the output. This is done as follows: Let $y \in \{0,1\}^n$ be the string that is to be excluded from the output of our proof circuit. Take the output bits $w_1, \ldots, w_n$ of C and feed them to a layer of XOR gates that does a bit-by-bit XOR of w and y. The output of the XOR layer is our output string. Since C never outputs all 0s, the output after XOR-ing with y can never be y.

Now we push this further to exclude k strings.

Let $L^{=n} = \{0,1\}^n \setminus U$, where $U = \left\{u^1, u^2, \ldots, u^k\right\}$ and $u^1, \ldots, u^k \in \{0,1\}^n$ are the strings excluded from L.

The proof is by induction on $|U|$. The base case of $|U| = 1$ has already been shown.

Assume we have a proof circuit for $L^{=n} \setminus U$ for all U with $|U| < k$. Induction step: $|U| = k$. Let l be the first position where there is at least one string in U which has 0 at l and at least one string in U that has a 1 at l. Since $|U| > 1$, there exists such an l. Now partition

U into U^0 and U^1 based on whether a string has a 0 or a 1 at the l'th position. Now by the choice of l, $|U^0| < k$ and $|U^1| < k$. From the induction hypothesis we have a proof circuit C^0 for $\{0,1\}^n \setminus U^0$ and a proof circuit C^1 for $\{0,1\}^n \setminus U^1$. We want the language $L^{=n} = L^0 \cap L^1$. We construct proof circuit C for $L^{=n}$ that uses the outputs of C^0 and C^1, and takes an additional bit b as input. Let $s \in L$ be an arbitrary fixed string. Define $C(bx)$ where b is a bit as follows:

- $C(bx) = C^0(x)$ if $b = 0$ and $C^0(x)_l = 0$;
- $C(bx) = C^1(x)$ if $b = 1$ and $C^1(x)_l = 1$;
- $C(bx) = s$ otherwise.

□

2.2.2 Sufficient conditions

A simple observation relating deciding membership to verifying proofs is as follows:

Proposition 2.13. *Every language decidable in nonuniform* NC^0 *has a nonuniform* NC^0 *proof system.*

Proof. Let L be a language decidable in nonuniform NC^0. Let C be an NC^0 circuit family that decides it. We construct an NC^0 proof system D for L as follows: D outputs the input x if $C(x) = 1$ and otherwise outputs w where w is a fixed string in L. It is easy to see that D enumerates exactly the words accepted by C. □

We now use the ideas developed so far to give sufficient conditions for a regular language to admit an NC^0 proof system.

Our first sufficient condition abstracts the strategy used to show that OR has an NC^0 proof system. This strategy exploits the fact that there is a DFA for OR, where every useful state has a path to an "absorbing" final state. (Here, by useful we mean that the state q lies on some path from the start state to a final state. This is syntactic usefulness, and may not correspond to real usefulness for acceptance if for each such path through q there is also another accepting path avoiding q. But we show that it is useful for designing NC^0 proof systems.

Theorem 2.14. *Let L be a regular language accepted by an NFA $M = (Q, \Sigma, \delta, F, q_0)$. Let $F' \subseteq F$ denote the set of absorbing final states; that is, $F' = \{f \in F \mid \forall a \in \Sigma, \delta(f, a) = f\}$. Suppose M satisfies the following condition:*

For each $q \in Q$, if there is a path from q to some $f \in F$, then there is a path from q to some $f' \in F'$.

Then L has an NC^0 proof system.

Proof. We assume without loss of generality that all states of M are useful (they lie on some accepting path); if not, we remove non-useful states (and the hypothesized property continues to hold). The hypothesis is that from each state q, we can reach some absorbing final state via a word of length at most $k = |Q| - 1$. Pick any such word arbitrarily, pad it arbitrarily with a suffix so that its length is exactly k, and denote the resulting word as $\text{fin}(q)$ (i.e., $\text{fin}(q)$ "finalizes" q). Clearly, $\delta(q, \text{fin}(q)) \in F'$.

The proof consists of the word x broken into blocks of size k, with the remainder bits at the beginning. In addition, the proof provides the state of M after each block on some accepting run. So the total proof is $x^0, x^1, \ldots, x^N, q^1, \ldots, q^N$ where $N = \lfloor n/k \rfloor$, each $q^i \in Q$, $x^i \in \Sigma^k$ for $i \geq 1$, and $x^0 \in \Sigma^{<k}$ are the remainder bits.

The word w output by the proof system on such a proof is also broken into blocks in the same way, and each block is defined as follows:

$$
\begin{aligned}
w^0 &= x^0 \\
w^1 &= \begin{cases} x^1 & \text{if } q^2 \in \delta(q^0, x^0 x^1) \\ \text{fin}(\delta(q_0, x^0)) & \text{otherwise.} \end{cases} \\
\text{For } 2 \leq i \leq N, w^i &= \begin{cases} x^i & \text{if } q^i \in \delta(q^{i-1}, x^i) \\ \text{fin}(q^{i-1}) & \text{otherwise.} \end{cases}
\end{aligned}
$$

(Note that for string lengths less than $2k$, we only have $O(1)$ many strings in L. So strings of these lengths can be generated trivially by a circuit that takes as input a number $i \in [2k]$ and outputs the i'th word for that length according to some ordering fixed beforehand.)

To see that our construction gives us an NC^0 family, note that since $|Q|$ and $|\Sigma|$ are constant, the transition function δ can be implemented by a circuit of constant size. And since k is a constant, checking if $q^i \in \delta(q^{i-1}, x^i)$ can be done in NC^0. Thus the above can be implemented in NC^0. □

We now look at a subset of regular languages called star-free languages. A regular language is called star-free if the language has a regular expression that does not use any Kleene star, but is allowed to use union, intersection and complementation. For example the language OR can be captured by the expression $\phi^c\ 1\ \phi^c$ where c denotes complement.

The language Exact-Or is also star-free and can be written as $(\phi^c\ 1\ \phi^c\ 1\ \phi^c)^c \cap (\phi^c\ 1\ \phi^c)$. However, we know from 2.5 and 2.9 that OR has an NC^0 proof system and Exact-Or does not. Observe that the complements in the expression for OR are applied to the empty set, whereas those in Exact-Or are applied to non-empty sets too. Based on this, we formulate and prove the following sufficient condition for a star-free regular language to have an NC^0 proof system.

Definition 2.15. *Strict star-free expressions over an alphabet Σ are exactly the expressions obtained as follows:*

1. *ϵ, a for each $a \in \Sigma$, $\Sigma^* = \bar{\emptyset}$ are strict star free.*
2. *If r and s are strict star free, so is $(r \cdot s)$.*
3. *If r and s are strict star free, so is $(r + s)$.*

Theorem 2.16. *Let r be a strict star-free expression describing a language $L = L(r)$. Then L admits an NC^0 proof system.*

Proof. We first note that in a regular expression, $\cdot$ distributes over $+$. Hence it is possible to repeatedly apply this rule of distributivity to arrive at an expression that is of the form $s_1 + s_2 + \cdots + s_k$, where each s_i is simply a concatenation without any $+$. So we assume that we have a strict star-free regular expression in this form.

Now, if we can show that each of the expressions s_i has an NC^0 proof system, then, we can use the fact that NC^0 proof systems are closed under finite union (Lemma 2.10).

The following claim shows that this is indeed true:

Claim 2.17. *Let L be a language recognized by a strict star-free expression s that does not have a $+$. Then L admits NC^0 proof systems.*

Proof. The expression s must be of the form $w_1\ \bar{\emptyset}\ w_2\ \bar{\emptyset}\ \ldots\ w_{k-1}\ \bar{\emptyset}\ w_k$, where $w_i \in \Sigma^+$ for $1 < i < k$ and $w_1, w_k \in \Sigma^*$. Let $s = w_1\ \bar{\emptyset}\ w_2\ \bar{\emptyset}\ \ldots\ w_{k-1}\ \bar{\emptyset}\ w_k$. Note that if $w_1 \neq \epsilon$, then we can hardwire w_1 to be the first $|w_1|$ symbols in the output of our proof circuit. Similarly w_k can be hardwired at the end. Now for the central $\bar{\emptyset}\ w_2\ \bar{\emptyset}\ w_3\ \ldots\ w_{k-1}\ \bar{\emptyset}$ part: Notice that any minimal DFA for this expression will have a self-absorbing final state to which all states have a path. Hence Theorem 2.14 implies that we have an NC^0 proof system for this language. Using this NC^0 proof system, and hardwiring w_1 and w_k as prefix and suffix respectively, we obtain an NC^0 proof system for L. □

This completes the proof of Theorem 2.16. □

Theorem 2.14 essentially characterizes functions like OR. On the other hand, the parity function, that has an NC^0 proof system, cannot be recognized by any DFA or NFA with an absorbing final state. The strategy used in constructing the proof system for parity exploits the fact that the underlying graph of the DFA for parity is strongly connected. In this scenario, the idea of using prefix parities as the input can be seen as taking the state of the automaton after each symbol as input. Hardwiring the last bit to 1 can be seen as forcing the end state to be the accepting state of the automaton. The same idea is used in Proposition 2.8 where we ask for a string that encodes the intermediate states in an accepting run.

In the following result, we abstract this property and prove that strong connectivity in an NFA recognizer is indeed sufficient for the language to admit an NC^0 proof system.

Theorem 2.18. *Let L be accepted by NFA $M = (Q, \Sigma, \delta, F, q_0)$. If the directed graph underlying M is strongly connected, then L admits an NC^0 proof system.*

Proof. We use the term "walk" to denote a path that is not necessarily simple, and "closed walk" to denote a walk that begins and ends at the same vertex. The proof system circuit construction we describe below is applied only for those lengths where L is non-empty.

The idea behind the NC^0 proof system we will construct here is as follows: We take as input a sequence of blocks of symbols $x^1, x^2, \ldots, x^k$, each of length l and as proof, we take the sequence of states $q^1, q^2, \ldots, q^k$ that M reaches after each of these blocks, on some accepting run. Now we make the circuit verify at the end of each block whether that part of the proof is valid. If it is valid, then we output the block as is. Otherwise, if some x^i does not take M from q^{i-1} to q^i, then we want to make our circuit output a string of length l that indeed makes M go from q^{i-1} to q^i. So we make our circuit output a string of symbols which will first take M from q^{i-1} to q_0, then from q_0 to q^i. To ensure that this length is indeed l, we sandwich in between a string of symbols that takes M on a closed walk from q_0 to q_0. We now proceed to formally prove that closed walks of the required length always exist, and that this can be done in NC^0.

Define the following set of non-negative integers:

$$T = \{\, \ell \mid \text{there is a closed walk through } q_0 \text{ of length exactly } \ell \,\}$$

Since M is strongly connected, we know that T is non-empty. Let g be the greatest common divisor of all the numbers in T. Note that though T is infinite, it has a finite subset T' whose gcd is g.

Claim 2.19. *For every $p \in Q$, $\exists \ell_p, r_p \in \{0, 1, \ldots, g-1\}$ such that*

1. *the length of every path from q_0 to p is $\equiv \ell_p \pmod g$;*
2. *the length of every path from p to q_0 is $\equiv r_p \pmod g$.*

Proof. Let ℓ, ℓ' be the lengths of two q_0-to-p paths, and let r, r' be the lengths of two p-to-q_0 paths. Then there are closed walks through q_0 of length $\ell + r$, $\ell + r'$, $\ell' + r$, $\ell' + r'$, and so g must divide all these lengths. So $\ell = -r \pmod g = -r' \pmod g$, and $r = -\ell \pmod g = -\ell' \pmod g$. It follows that $\ell \equiv \ell' \pmod g$ and $r \equiv r' \pmod g$. □

From here onwards, for each $p \in Q$, by ℓ_p and r_p we mean the numbers as defined in the above claim.

Choose a subset S of states as follows:

$$S = \{\, q \in Q \mid \text{there is a walk from } q_0 \text{ to } q \text{ whose length is } 0 \bmod g \,\}$$

Claim 2.20. *For every $p \in S$, $\ell_p = r_p = 0$.*

Proof. By the definition of S, we have $\ell_p = 0$. Suppose $r_p \neq 0$. Let w be a word taking M from p to q_0. Appending this to any word w' that takes M from q_0 to p gives a closed walk through q_0 whose length is $0 + r_p \neq 0 \pmod g$. This contradicts the fact that g is the gcd of numbers in T. □

Claim 2.21. *There is a constant c_0 such that for every $K \geq c_0$, there is a closed walk through q_0 of length exactly Kg.*

Proof. This follows from Lemma 2.22 below. □

Note that if $g = 1$, then every state is in S, and for every state p, $l_p = r_p = 0$. Claim 2.21 then asserts that there are closed walks through q_0 of every possible length exceeding c_0.

Let $K = |Q|$. Now set $t = \lfloor \frac{K-1}{g} \rfloor$ and $\ell = t \cdot g$. Then for every $p \in S$, there is a path from q_0 to p of length $t'g$ on word $\alpha(p)$, and a path from p to q_0 of length $t''g$ on word $\beta(p)$, where $0 \leq t', t'' \leq t$. ($\alpha(p)$ and $\beta(p)$ are not necessarily unique. We can arbitrarily pick any such string.)

If for all accepting states $f \in F$, $\ell_f \not\equiv n \pmod g$, then $L^{=n} = \emptyset$, and the circuit C_n is empty.

Otherwise, let $r = n \pmod g$. There is at least one final state f such that $\ell_f \equiv r \pmod g$. Thus there is at least one string of length $t'g + r$, with $0 \leq t' \leq t$, that takes M from q_0 to f.

We now construct a proof circuit $C : \Sigma^n \times Q^n \longrightarrow \Sigma^n$. We consider the inputs of the proof circuit to be divided into blocks. We choose the block size to be a multiple of g, with the possible exception of the last block. In particular, we choose block size $cg = (2t + c_0)g$ where c_0 is the constant from Claim 2.21. The last block is of size $c'g + r$ for some $0 \leq c' \leq c$.

Let $k = \lfloor n/cg \rfloor$. Now the total proof is $x^1, \ldots, x^k, x^{k+1}, q^1, \ldots, q^k, q^{k+1}$ where each $q^i \in Q$, $x^i \in \Sigma^{cg}$ for $i \leq k$, and $x^{k+1} \in \Sigma^{c'g+r}$ for some $0 \leq c' < c$.

The word w output by the proof system on such a proof is also broken into blocks in the same way, and each block is obtained as follows:

1. For $1 \leq i < k$, if $q^i \in \delta(q^{i-1}, x^i)$, then $w^i = x^i$. Otherwise, w^i is obtained by concatenating $\beta(q^{i-1})$, a word u such that $q_0 \in \delta(q_0, u)$, and $\alpha(q^i)$. We need $|u| = (c - t' - t'')g$, and we know that $(c - t' - t'') \geq c_0 g$, and hence Claim 2.21 guarantees that such a word u exists.

2. If $q^{k+1} \in \delta(q^{k-1}, x^k x^{k+1})$ and $q^{k+1} \in F$, then let $w^k w^{k+1} = x^k x^{k+1}$.

   Otherwise, let $w^k w^{k+1}$ have as suffix a string of length $t'g + r$ in L, where $0 \leq t' \leq t$. By the choice of t we know that such a string exists. This leaves a prefix of length $(cg + c'g + r) - (t'g + r) = (c + c' - t)g$ with $(c + c' - t) \geq c_0 g$. We insert here a word u such that u takes q_0 to q_0; by Claim 2.21, such a word exists.

This completes the proof of the Theorem 2.18 except for Lemma 2.22, which is stated and proved below. □

Lemma 2.22 (Folklore)**.** *Let T be a set of positive integers with gcd g. There is a constant c_0 such that for every $K \geq c_0$, Kg can be generated as a non-negative integral combination of the integers in T.*

Proof. We prove the statement by induction on $|T|$. Let $T = \{m_1, m_2, \ldots, m_t\}$ be the given set.

Basis: If $t = 1$, then $g = m_1$ and $Kg = Km_1$, so set c_0 to 1.

Inductive Hypothesis: Assume the statement is true for all sets of size $t - 1$.

Inductive Step: T is a set of size t.

It suffices to prove the statement when $g = 1$; for larger g, let T' be the set $\{t/g \mid t \in T\}$. Then T' has gcd 1, and if we can generate all numbers beyond c_0 with T', then we can generate all Kg for $K \geq c_0$ with T. So now assume T has gcd 1.

Let g' denote the gcd of the subset R consisting of the first $t-1$ numbers. If $g' = 1$, then, even without using the last number m_t, we are already done by induction. Otherwise, let $m = m_t$. Then the numbers g', m are co-prime (because gcd for T is 1). By induction, there is a constant c' such that using only numbers from R, we can generate $K'g'$ for any $K' \geq c'$. Set $c = (c' + m)g'$. Consider any number $n \geq c$.

The numbers $0 < n - (c' + m - 1)g', n - (c' + m - 2)g', \ldots, n - (c' + 1)g', n - c'g'$ all have different residues modulo m.
(If not, suppose for some $0 \leq i < j \leq m-1$, $n - (c' + i)g' \equiv n - (c' + j)g' \pmod{m}$. Then $(j - i)g' \equiv 0 \pmod{m}$, and so m must divide $(j - i)g'$. Since $0 < j - i < m$, m does not divide $j - i$. But m is co-prime to g'. Contradiction.)
So for some $0 \leq i < m$, and for some non-negative integer a, $n - (c' + i)g' = am$. That is, $n = (c' + i)g' + am$. By the induction hypothesis, $(c' + i)g'$ can be generated using numbers in $R \subseteq T$. And $m \in T$. So n can be generated from T. □

Example 2.23. The following shows the construction of the proof circuit as in the proof of Theorem 2.18 for the regular language $L = (1^9 + 0^6)^*$. Consider the NFA for L shown in Figure 2.1. Using the same notation as in Theorem 2.18, we have $T = \{6, 9\}$. The greatest

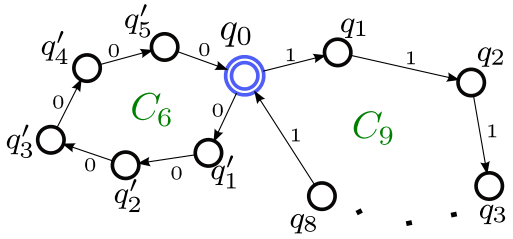


Figure 2.1: The strongly connected NFA from Example 2.23

common divisor of the numbers in T is $g = 3$. Then we have $S = \{q_3, q_6, q'_3\}$. It is easy to see that in our example, Claim 2.21 goes through for $c_0 = 2$. We choose the block length to be

$$\ell = \left\lfloor \frac{|Q| - 1}{g} \right\rfloor g + c_0 g + \left\lfloor \frac{|Q| - 1}{g} \right\rfloor g = 12 + 6 + 12 = 30.$$

This is chosen such that in the case of an input block that claims to take state p to state p' and does not have the correct proof, our proof system can output a set of at most 12 symbols to go from p to q_0, and then cycle around q_0 using kg symbols where $k > c_0$ is chosen appropriately and finally output at most 12 symbols to go from q_0 to p'. Note that in this way, for each pair of states $p, p' \in S$, we can produce a path $p \rightsquigarrow q_0 \rightsquigarrow q_0 \rightsquigarrow p'$ of length exactly 30. For example: Consider the pair q_3, q'_3. There is a path of length 6 from q_3 to q_0 and a path of length 3 from q_0 to q'_3. Since we want the total length of the path from q_3 to q'_3 to be exactly 30, we sandwich a closed walk of length 21 at q_0. It is easy to

see that such a closed walk exists in the NFA shown: two rounds of C_6 and one round of C_9.

Theorem 2.18 gives the following corollary:

Corollary 2.24. *For every $n > 1$, the language MOD_n=$\{ x \mid |x|_1 \equiv 1 \bmod n \}$ admits an NC^0 proof system.*

All the proof systems that we constructed for the regular languages till now are obtained by applying one of Theorems 2.14, 2.16, 2.18, in conjunction with a generic closure property.

Now that we have established certain sufficient conditions for a regular language to have NC^0 proof systems, we want to ask the opposite question - How much more resources are needed to construct proof systems for every regular language?

2.2.3 Closures and Non-closures

In this section, we identify a class of circuits that have sufficient power to compute proof systems for every regular language.

Theorem 2.14 and Theorem 2.18 suggest that there exists a connection between the structure of automatons and proof systems. To understand this better, we first explore the extent to which the structure of regular languages can be used to construct proof systems. At the base level, we know that all finite languages have NC^0 proof systems. Building regular expressions involves unions, concatenation, and Kleene closure. And the resulting class of regular languages is also closed under many more operations. We examine these operations one by one with respect to NC^0 proof systems:

Theorem 2.25. *Let C denote the class of languages with NC^0 proof systems. Then C is closed under*

1. *finite union (Lemma 2.10),*
2. *concatenation with finite sets (Lemma 2.4),*
3. *reversal,*
4. *fixed-length morphisms,*
5. *inverses of fixed-length morphisms,*
6. *fixed-length regular transductions computed by strongly connected (nondeterministic) finite-state automata.*

Proof. Closure under reversal is trivial.

Let h be a fixed-length morphism $h : \{0,1\} \longrightarrow \{0,1\}^k$ for some fixed k. Given a proof system (C_n) for L, a proof system (D_n) for $h(L)$ consists of n parallel applications of h to the each bit of the output of the circuit C_n. Given a proof system D'_n for L, a proof system C'_n for $h^{-1}(L)$ consists of n parallel applications of h^{-1} applied to disjoint k-length blocks of the output of the circuit D'_{kn}. C'_n needs additional input for each block to choose between possibly multiple pre-images.

If L has an NC^0 proof system (C_n) and h is a regular transduction computed by a strongly connected automaton M, the construction from 2.18 with the output w of C_n as input will produce a word $x \in L(M)$. A small modification allows us output the transduction $h(x)$ instead of x. This works provided there are constants k, ℓ such that each edge in M is labeled by a pair (a, b) with $a \in \{0,1\}^k$ and $b \in \{0,1\}^\ell$. □

We now observe some non-closures. We will crucially use the fact that the language Exact-Or does not admit NC^0 proof systems (See Theorem 2.9).

Theorem 2.26. *Let C denote the class of languages with NC^0 proof systems. C is not closed under*

1. *complementation,*
2. *concatenation,*
3. *symmetric difference,*
4. *cyclic shifts,*
5. *permutations and shuffles,*
6. *intersection,*
7. *quotients.*

Proof. To see complementation, note that Th_2^n has an automaton with an absorbing final state and hence using Theorem 2.14 has an NC^0 proof system. However, its complement Exact-Or $\cup\ 0^*$ does not have an NC^0 proof system as a consequence of generalizing Theorem 2.9 (see page 46). The languages denoted by the regular expressions 1, 0^*, 10^*, and the languages Th_1, Th_2 all have NC^0 proof systems. The language Exact-Or does not, but it can be written as $0^* \cdot 10*$ (concatenation), as $\mathrm{Th}_2 \Delta \mathrm{Th}_1$ (symmetric difference), as the result of cyclic shifts or permutations on 10^*, and as the shuffle of 1 and 0^*.

To see the last two non-closures, it is easier to use non-binary alphabets; the coding back to $\{0,1\}$ is straightforward. The languages $(0^*10^* \cup (0+1+a)^*a(0+1+a)^*)$ and $(0^*10^* \cup (0+1+b)^*b(0+1+b)^*)$ both have NC^0 proof systems (this follows from Theorem 2.14), but their intersection is Exact-Or. Also, consider the languages $A = a0^*$, $B_1 = \{xay \mid |x| = |y|, x \in \text{Exact-Or}, y \in 0^*\}$, $B_2 = \{xay \mid |x| = |y|, x \in (0+1)^*, y \in \mathrm{Th}_1\}$.

Then A and $B = B_1 \cup B_2$ have NC^0 proof systems but Exact-Or $= B \mid A$.
(A proof system for B is as follows: the input proof at length $2n+1$ consists of a word $w \in (0+1)^n$ and the sequence of n states $q_1, \ldots, q_n$ allegedly seen by an automaton M for Exact-Or on reading w. The circuit copies w into x. If q_{i-1}, w_i, q_i is consistent with M, then it sets y_i to 0, otherwise it sets y_i to 1. It can be verified that the range of this circuit is exactly $B^{=2n+1}$.) □

A natural idea is to somehow use the structure of the syntactic monoid (equivalently, the unique minimal deterministic automaton) to decide whether or not a regular language has an NC^0 proof system, and if so, to build one. Unfortunately, this idea collapses at once: the languages Exact-Or and Th_2 have the same syntactic monoid; by Theorem 2.9, Exact-Or has no NC^0 proof system; and by Proposition 2.14 Th_2 has such a proof system.

The next idea is to use the structure of a well-chosen (nondeterministic) automaton for the language to build a proof system; Theorem 2.14 and Theorem 2.18 do exactly this. They describe two possible structures that can be used. However, one is subsumed in the other; see Observation 2.27 below.

Observation 2.27. *Let L be accepted by an automaton with a universally reachable absorbing final state. Then L is accepted by a strongly connected automaton.*

Proof. Let M be the non-deterministic automaton with universally reachable and absorbing final state q. That is, q is an accepting state such that (1) q is reachable from every other state of M, and (2) there is a transition from q to q on every letter in Σ. Add ϵ-moves from q to every state of M to get automaton M'. Then M' is strongly connected, and $L(M') = L(M)$. □

A small generalisation beyond strongly connected automata is automata with exactly two strongly connected components. However, the automaton for Exact-Or is like this, so even with this small extension, we can no longer construct NC^0 proof systems. (In fact, we need as much as $\Omega(\log \log n)$ depth.)

Finite languages do not have strongly connected automata. But they are strict star-free and hence have NC^0 proof systems. Strict star-free expressions lack non-trivial Kleene closure. What can we say about their Kleene closure? It turns out that for any regular language, not just a strict-star-free one, the Kleene closure has an NC^0 proof system.

Theorem 2.28. *If L is regular, then L^* has an NC^0 proof system.*

Proof. Let M be an automaton accepting L, with no useless states. Adding ϵ moves from every final state to the start state q_0, and adding q_0 to the set of final states, gives an automaton M' for L^*. Now M' is strongly connected, so Theorem 2.18 gives the NC^0 proof system. □

Based on the above discussion and known (counter-) examples, we conjecture the following characterization. The structure implies the proof system, but the converse seems hard to prove.

Conjecture 1. *Let L be a regular language. The following are equivalent:*

1. *L has an NC^0 proof system.*
2. *For some finite k, $L = \bigcup_{i=1}^{k} u_i \cdot L_i \cdot v_i$, where each u_i, v_i is a finite word, and each L_i is a regular language accepted by some strongly connected automaton.*

An interesting question arising from this is whether the following languages are decidable:

$$\text{Reg-SCC} = \left\{ M \mid \begin{array}{l} M \text{ is a finite-state automaton; } L(M) \text{ is accepted by} \\ \text{some strongly connected finite automaton} \end{array} \right\}$$

$$\text{Reg-NC}^0\text{-PS} = \left\{ M \mid \begin{array}{l} M \text{ is a finite-state automaton; } L(M) \text{ has an } \mathsf{NC}^0 \text{ proof} \\ \text{system} \end{array} \right\}$$

(Instead of a finite-state automaton, the input language could be described in any form that guarantees that it is a regular language.)

It can be shown that Reg-SCC is indeed decidable by using the result of Igor Grunsky et al. in [GKP06]. The main result from [GKP06] shows that for every regular language L, there exists a constant c_L such that every NFA with more than c_L states that recognizes L contains at least 2 mergeable states. Moreover, c_L can be computed from a representation of L. The key observation is that merging two states in a strongly connected NFA results in a strongly connected NFA. So if there is a strongly connected NFA for L, then there is one with at most c_L states. Hence the problem of checking if a regular language L has a strongly connected NFA can be decided by first computing c_L and then checking if any NFA with at most c_L states accepts L.

2.2.4 An upper bound for all regular languages

We now establish the main result of this section. NC^0 is the restriction of AC^0 where the fanin of each gate is bounded by a constant. By putting a fanin bound that is $\omega(1)$

but $o(n^c)$ for every constant c ("sub-polynomial"), we obtain intermediate classes. In particular, restricting the fanin of each gate to be at most poly log n gives the class that we call poly log AC^0 lying between NC^0 and AC^0. We show that it is large enough to have proof systems for all regular languages. As mentioned earlier, Theorem 2.9 implies that this upper bound is tight.

The idea behind the construction is the following: Start with an automaton A that accepts the regular language. As proof that a string x is accepted by A, we expect to be given the state of A after it reads x_i and x_j for certain values of i, j. The values of (i, j) are chosen in a way so that any inconsistencies can be checked and corrected very locally. These values of i, j can be seen as forming a tree that we describe more precisely in the proof. For convenience, we convert the automaton accepting the language into a branching program, which are defined as follows:

A Branching program (BP) P_n on n inputs is a directed acyclic graph with a start node s and a sink node t and each edge labelled by a literal or a bit. The branching program P accepts an input x if there is a path from s to t where all edges labels take on the value 1. A family of branching programs is a collection of branching programs $(P_n)_{n\geq 1}$ - one for each input length.

Theorem 2.29. *Every regular language has a* poly log AC^0 *proof system.*

Proof. Let $A = (Q, \Sigma, \delta, q_0, F)$ be an automaton for L. We assume that $\Sigma = \{0, 1\}$, larger finite alphabets can be suitably coded. We unroll the computation of A on inputs of length n to get a layered branching program B with $n+1$ layers numbered 0 to n. The initial layer of B has just the start node s which behaves like q_0 in the automaton, while every other layer of the branching program has as many vertices as $|Q|$. Since A may have multiple accepting states, we add an extra layer at the end with a single sink node t, and connect all copies of accepting states at layer n to t by edges labeled 1. Note that B has the following properties:

- Length $l = n + 2$.
- Every layer except the first and last layer has width (number of vertices in that layer) $w = |Q|$.
- Edges are only between consecutive layers. These edges and their labelling are according to δ.
- All edges from layer $i - 1$ to layer i are labelled either x_i or $\overline{x}_i$.

- A word $a = a_1 \dots a_n$ is accepted by A if and only if B has a path from s to t (with $n + 1$ edges) with all edge labels consistent with a.

Any vertex $u \in B$ can be indexed by a two tuple (ℓ, p) where ℓ stands for the layer where u appears and p is the position where u appears within layer ℓ.

Represent the interval $(0, n + 1]$ as a binary tree T where

1. the root corresponds to the interval $(0, n + 1] = \{1, 2, \dots, n + 1\}$,
2. a node corresponding to interval $(i, j]$ has children corresponding to intervals $(i, \lceil \frac{i+j}{2} \rceil]$ (left child) and $(\lceil \frac{i+j}{2} \rceil, j]$ (right child), and
3. a node corresponding to interval $(k - 1, k]$ for $k \in [n + 1]$ is a leaf.

We call this the interval tree. For each interval $(i, j]$ in T, we provide a pair of states $\langle u, v \rangle$; these are intended to be the states q_i and q_j in the alleged accepting run ρ. (Note that the state sequence on ρ itself is now supposed to be specified at the leaves of T.)

Consider the interval tree T for $(0, n + 1]$ described above. The input to the proof system consists of a string $a \in \{0, 1\}^n$ and a pair of labels $\langle u, v \rangle$ for each node in the interval tree. The labels u, v point to nodes of B. For interval $(i, j]$, the labels are of the form $u = (i, p)$, $v = (j, q)$. Since i, j are determined by the node in T, the input only specifies the pair $\langle p, q \rangle$ rather than $\langle u, v \rangle$. That is, it specifies a pair of states from A. At the root node, the labeling is hardwired to be $\langle s, t \rangle$.

Given a word $a = a_1 \dots a_n$ and a labeling as above of the interval tree, we define feasibility and consistency as follows:

1. A leaf node $(k - 1, k]$ with $k \in [n]$, labeled $\langle p, q \rangle$, is
   (a) **feasible** if there exists an edge from $(k - 1, p)$ to (k, q) in B. (That is, there exists $b \in \Sigma$ such that $q \in \delta(p, b)$.)
   (b) **consistent** if there exists an edge from $(k - 1, p)$ to (k, q) in B labeled x_k if $a_k = 1$, labeled $\overline{x_k}$ if $a_k = 0$. (That is, $q \in \delta(p, a_k)$.)

   (The case $k = n + 1$ is simpler: feasible and consistent if p is a final state of A.)
2. An internal node $(i, j]$ labeled $\langle p, q \rangle$ is
   (a) **feasible** if there exists a path from (i, p) to (j, q) in B. (That is, there exists a word $b \in \Sigma^{j-i}$ such that $q \in \tilde{\delta}(p, b)$.)

(b) **consistent** if it is feasible, both its children are feasible, and the labels $\langle p', q' \rangle$ and $\langle p'', q'' \rangle$ of its left and right children respectively satisfy: $p = p'$, $q = q''$, $q' = p''$.

3. A node is **fully consistent** if all its ancestors (including itself) are consistent.

Since the label at the root of T is hardwired, the root node is always feasible. But it may not be consistent.

For each node $(i, j]$ in the interval tree, and each potential labeling $\langle p, q \rangle$ for this node, let $u = (i, p)$ and $v = (j, q)$. Define the predicate $R(u, v)$ to be 1 if and only if there is a path from u to v in B. (i.e., this potential labeling is feasible.) Whenever $R(u, v) = 1$, fix a partial assignment $w_{u,v}$ that assigns 1 to all literals that occur as labels along an arbitrarily chosen path from u to v). Note that $w_{u,v}$ assigns exactly $j-i$ bits, to the variables $x_{i+1}, \ldots, x_j$. We call $w_{u,v}$ the **feasibility witness** for the pair (u, v).

Let y be the output string of the proof system we construct. A bit y_k of the output y is computed as follows: Find the lowest ancestor of the node $(k-1, k]$ that is fully consistent.

- If the leaf node $(k-1, k]$ is fully consistent, output a_k.
- If there is no such node, then the root node is inconsistent. Since it is feasible, the word $w_{s,t}$ is defined. Output the kth bit of $w_{s,t}$.
- If such a node is found, and it is not the leaf node itself but some $(i, j]$ labeled $\langle p, q \rangle$, let $u = (i, p)$ and $v = (j, q)$. The word $w_{u,v}$ is defined and assigns a value to x_k. Output this value.

It follows from this construction that every word $a \in L$ can be produced as output: give in the proof the word a, and label the interval tree fully consistent with an $s - t$ path of B consistent with a (equivalently, an accepting run of A on a).

It also follows that every word y output by this construction belongs to L. On any proof, moving down from the root of the interval tree, locate the frontier of lowest fully consistent nodes. These nodes are feasible and correspond to a partition of the input positions, and the procedure described above outputs a word constructed by patching together the feasibility-witnesses for each part.

To see that the above construction can be implemented in depth $O(\log \log n)$ with $O(1)$ alternations, observe that each of the conditions - feasibility, consistency and equality of two labels depend on $O(\log w)$ bits. Hence depth of $O(\log \log n)$ and $O(1)$ alternations suffices for their implementation.

More formally, define the following set of predicates:

- $\textsc{Equal} : [w]^2 \longrightarrow \{0,1\}$ the Equality predicate on $\log w$ bits.
- For each $0 \le i < j \le n+1$, $\textsc{Feasible}_{i,j} : [w]^2 \longrightarrow \{0,1\}$ is the Feasibility predicate with arguments the labels (p,q) at interval $(i,j]$.
- For each $0 \le i < j+1 \le n+1$, $\textsc{Consistent}_{i,j} : [w]^6 \longrightarrow \{0,1\}$ is the Consistency predicate at an internal node, with arguments the labels at interval $(i,j]$ and at its children.
- For each $0 < k \le n+1$, $\textsc{ConsistentLeaf}_k : [w]^2 \times \Sigma \longrightarrow \{0,1\}$ is the Consistency predicate at leaf $(k-1,k]$ with arguments the label $\langle p,q\rangle$ and the bit a_k at the leaf.

All the predicates depend on $O(\log w)$ bits. So a naive truth-table implementation suffices to compute them in depth $O(\log w)$ with $O(1)$ alternations.

For any $0 < k \le n+1$, let the nodes on the path from $(k-1,k]$ to the root of the interval tree be the intervals $(k-1,k] = (i_0,j_0), (i_1,j_1], \ldots, (i_r,j_r] = (0,n+1]$. Note: $r \in O(\log n)$.

Given a labeling of the tree, the output at position k is given by the expression below. (It looks ugly, but it is just implementing the scheme described above. We write it in this detail to make the poly log AC^0 computation explicit.)

$$\begin{aligned} y_k &= \left[a_k \wedge \textsc{ConsistentLeaf}_k \wedge \bigwedge_{h=1}^{r} \textsc{Consistent}_{i_h,j_h} \right] \\ &\vee \left[(w_{s,t})_k \wedge \overline{\textsc{Consistent}_{0,n+1}} \right] \\ &\vee \left[\bigvee_{h=1}^{r} (w_{(i_h,p_h),(j_h,q_h)})_k \wedge \overline{\textsc{Consistent}_{i_{h-1},j_{h-1}}} \wedge \bigwedge_{g=h}^{r} \textsc{Consistent}_{i_g,j_g} \right] \end{aligned}$$

where the arguments to the predicates are taken from the tree labeling. This computation adds $O(1)$ alternations and $O(\log\log n)$ depth to the computation of the predicates, so it is in poly log AC^0. □

2.3 Proof systems for other languages

2.3.1 NC^0 proof systems

In this section, we construct NC^0 proof systems for a variety of languages. We start with an NC^1-complete problem, viz. reachability in bounded width directed acyclic graphs.

A layered graph with vertices arranged in layers from $0, 1, \ldots, L$ with exactly W vertices per layer (numbered from $0, \ldots, W-1$) and edges between vertices in layer i to $i+1$ for $i \in \{0, \ldots, L-1\}$ is a positive instance of reachability if and only if there is a directed path from vertex 0 at layer 0 to vertex 0 at layer L. A description of the graph consists of a layer by layer encoding of the edges as a bit vector. In other words it consists of a string $x = x^0 x^1 \ldots x^{L-1} \in (\{0,1\}^{W^2})^L$ where the x^i is indexed by $j, k \in \{0, \ldots, W-1\}$ and $x^i[j,k] = 1$ if and only if there is an edge from j-th vertex on the i-th layer to the k-th vertex on the $(i+1)$-th layer. The language $BWDR_W$ (where *BWDR* stands for Bounded Width Directed Reach) consists of those strings $x \in (\{0,1\}^{W^2})^L$ which describe a positive instance of reachability. Then we have:

Proposition 2.30. *For every fixed W, the language $BWDR_W$ admits an NC^0 proof system.*

Proof. The proof consists of a string $x \in (\{0,1\}^{W^2})^L$ which describes the graph and a string $v = v^1 \ldots v^{L-1} \in (\{0,1\}^V)^{L-1}$ representing a path. Here $V = \lceil \log W \rceil$ is the number of bits required to describe a vertex at a given layer in binary.

Given x, v we first replace each v^i which occurs in v and which represents a number greater than $W-1$ by the bit string consisting of V zeros. This requires a circuit of depth $O(\log(V)) = O(\log \log W)$. For the ease of notation we continue to refer to the modified v as v.

Next, for each $i \in \{0, \ldots, L-1\}$, we add the edge represented by (v^i, v^{i+1}) to the graph represented by x by setting $x^i[v^i, v^{i+1}] = 1$. This ensures that the graph contains the path represented by v, i.e. it is a positive instance. Clearly to address the appropriate bits of x^i we need a circuit of depth $O(\log V) = O(\log \log W)$. Finally, we output this modified x. It is easy to see that all positive instances will be output by this circuit for some inputs. Since W is a constant, we will obtain an NC^0 proof system. □

In the above proof, we crucially used the fact that the width W was a fixed constant. It is not clear how to capture directed reach in graphs with unbounded width. We show a connection between directed reach and the set of all tautologies expressed as 2DNFs in Theorem 4.1 of Chapter 4. However, we can show that the complement of connectedness has an NC^0 proof system. In the following proposition, we will define the language UnReach that is known to be NL-complete ([Imm88],[Sze88]) and show that it has an NC^0 proof system:

Proposition 2.31. *The language* UnReach *defined below has an NC^0 proof system under*

the standard adjacency matrix encoding.

$$\textsc{UnReach} = \left\{ A \in \{0,1\}^{n \times n} \middle| \begin{array}{l} A \text{ is the adjacency matrix of a directed graph } G \text{ with} \\ \text{no path from } s = 1 \text{ to } t = n. \end{array} \right\}$$

Proof. As proof, we take as input an adjacency matrix A and an n-bit vector X with $X(s) = 1$ and $X(t) = 0$ hardwired. Intuitively, X is like a characteristic vector that represents all vertices that can be reached by s.

The adjacency matrix B output by our proof system is:

$$B[i,j] = \begin{cases} 1 & \text{if } A[i,j] = 1 \text{ and it is not the case that } X(i) = 1 \text{ and } X(j) = 0, \\ 0 & \text{otherwise} \end{cases}$$

Soundness: No matter what A is, X describes an s,t cut since $X(s) = 1$ and $X(t) = 0$ and $\forall i, j, X(i) = 1 \wedge X(j) = 0 \implies B[i,j] = 0$. So any graph output by the proof system will not have a path from s to t.

Completeness: For any $G \in \textsc{UnReach}$, use the adjacency matrix of G as A and give input X such that $X(v) = 1$ for a vertex v if and only if v is reachable from s. □

The idea from the proof of Proposition 2.30 can be used for addition and comparison. Consider the function $f_+ : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^{n+1} \to \{0,1\}$ such that $f_+(a,b,s) = 1$ if and only if $A + B = S$ where a, b are the n-bit binary representations of the numbers A, B and s is the $(n+1)$-bit binary representation of S. Also consider the function $f_\leq : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ where $f_\leq(a,b) = 1 \iff A \leq B$, where again a, b are the n-bit binary representations of numbers A, B.

Proposition 2.32. *The language $L_+ = f_+^{-1}(1)$ admits an NC^0 proof system.*

Proof. The circuit $C_n : \{0,1\}^{3n} \to \{0,1\}^{3n}$ maps three strings $\alpha = \alpha_{n-1} \ldots \alpha_0$, $\beta = \beta_{n-1} \ldots \beta_0$ and $\gamma = \gamma_n \ldots \gamma_1$ (for notational convenience assume that $\gamma_0 = 0$) to strings a, b, s with the intent that γ will serve as the carry sequence in the grade-school addition of the two numbers α, β. Also, if we ever discover a discrepancy between the assumed carry sequence and the two numbers α, β we correct the error by altering α, β appropriately to yield a, b. Formally, for $0 \leq i \leq n-1$, if $\mathrm{Th}_2^3(\alpha_i, \beta_i, \gamma_i) = \gamma_{i+1}$ then $a_i = \alpha_i, b_i = \beta_i$ otherwise, set a_i, b_i arbitrarily under the constraint that $\mathrm{Th}_2^3(a_i, b_i, \gamma_i) = \gamma_{i+1}$. Also set $s_i = a_i \oplus b_i \oplus \gamma_i$. Set $s_n = \gamma_n$. The input-alteration ensures that the output is always in

the language, and for each word $\langle a, b, s\rangle$ in the language, the proof that gives the correct carry sequence ensures that the word is produced as output. □

Proposition 2.33. *The language $L_{\leq} = f_{\leq}^{-1}(1)$ admits an NC^0 proof system.*

Proof. The proof consists of four n-bit strings $\alpha, \alpha', \gamma, \beta$, with the intent that γ is the carry sequence for the sum of α, α' which yields β. Again in the proof we ensure that if the carry bits γ_i, γ_{i-1} are compatible with α, α' summing to β, then copy α_i, β_i to a_i, b_i respectively. Otherwise, set $a_i = 0, b_i = 1$ (which ensures that if $a_j = b_j$ for $j > i$ then $a < b$). As before, the input alteration guarantees soundness, and the proof with the correct carry bits guarantees completeness. □

We now consider a P-complete language, Grid Circuit Value. An instance consists of a planar circuit with vertices embedded in a square grid so that the circuit wires lie only along the grid edges and are directed to go only due east or due north. All possible wires are present. The gates can be arbitrary functions of the two inputs and two outputs. All inputs are present on the outer face of the circuit (i.e. on the southern and western boundaries). It is easy to see that this problem is contained in P. To see that it is P hard, we reduce the Circuit Value Problem to it under, say, $\mathsf{DLogspace}$ reductions. First make the circuit planar by using the usual cross-over gadget [Gol77] to remove all crossings. Now, embed the circuit in the grid by using a method similar to the one used in [ABC+09, CD06] to obtain the required embedding. Finally we replace all missing wires by altering the gates to ignore the value from any missing input and output an arbitrary value, say zero along all missing outputs.

Using the strategy of locally correcting the input if the proof shows an inconsistency, we can show the following:

Proposition 2.34. *The Grid Circuit Value Problem admits an NC^0 proof system.*

Proof. The proof consists of a string describing the circuit, that is, the truth tables of (both outputs) of a gate for each gate position and a value for each of the wires in the circuit. Since each truth table is for a 2-input and 2-output gate, it is represented by a truth table of 8 bits. Thus for a grid consisting of n vertices on each side, with m input variables, the input string is $(g, v) \in \{0,1\}^{8n^2} \times \{0,1\}^{2(n-1)n}$. The output of the circuit is a pair $(g', x, b) \in \{0,1\}^{8n^2} \times \{0,1\}^{2n-1} \times \{0,1\}$ with g' describing new truth tables obtained by copying those from g already consistent with v, and modifying the others to make them consistent with the values in v (this is always possible by setting one entry of each inconsistent truth-table). The string x describes the values (from v) corresponding to inputs and b the value of the output gate. □

Remark 2.35. As mentioned earlier, Cryan and Miltersen [CM01] (and in fact already Agrawal et al. [AAR98]) show that a certain NP-complete language admits an NC^0 proof system. The language they consider is just an encoding of 3-SAT: for each n, instances with n variables are encoded by an $M = 2^3\binom{n}{3}$ bit string, where each bit indicates whether the corresponding potential clause is present in the instance. A proof consists of an assignment to the propositional variables and a suggestion for a 3-CNF, which is modified by the proof system in order to be satisfied by the given assignment. The clause bit is flipped if (and only if) (1) it is on, and (2) the clause is not satisfied by the assignment. Since each potential clause has its reserved "indicator bit", checking whether the clause is satisfied by the assignment requires looking at exactly three fixed bits of the assignment. It is easy to see that this system generates exactly the satisfiable 3-SAT instances.

2.3.2 poly log AC^0 proof systems

While proving Theorem 2.29, we unrolled the computation of a w-state automaton on inputs of length n into a layered branching program BP of width w with $\ell = n + 2$ layers. The BP so obtained is nondeterministic whenever the automaton is. The BP has a very restricted structure which we exploited to construct the poly log AC^0 proof system.

We observe that some restrictions on the BP structure can be relaxed and still we can construct a poly log AC^0 proof system.

Definition 2.36. *A branching program for length-n inputs is* **structured** *if it satisfies the following:*

1. *It is layered: vertices are partitioned into $n + 1$ layers $V_0, \ldots, V_n$ and all edges are between adjacent layers $E \subseteq \cup_i (V_{i-1} \times V_i)$.*
2. *Each layer has the same size $w = |V_i|$, the width of the BP. (This is not critical; we can let $w = \max |V_i|$.)*
3. *There is a permutation $\sigma \in S_n$ such that for $i \in [n]$, all edges in $V_{i-1} \times V_i$ read $x_{\sigma(i)}$ or $\overline{x_{\sigma(i)}}$.*

Non-uniform automata [Bar89, BT88] give rise to branching programs that are structured with w the number of states in the automaton. For instance, the language $\{xx \mid x \in \{0,1\}^*\}$ is not regular. But if the input bits are provided in the order $1, m+1, 2, m+2, \ldots, m, 2m$ then it can be decided by a finite-state automaton. This gives rise to a structured BP where σ is the inverse of the above order. (eg $r_2 = m + 1$, $r_3 = 2$, $\sigma(m + 1) = 2$, $\sigma(2) = 3$.)

The idea behind the construction in Theorem 2.29 works for such structured BPs. It yields a proof system with depth $O(\log\log n + \log w)$. This means that for $w \in O(\operatorname{poly}\log n)$, we still get poly log AC^0 proof systems. Potentially, this is much bigger than the class of languages accepted by non-uniform finite-state automata. Formally,

Theorem 2.37. *Languages accepted by structured branching programs of width $w \in (\log n)^{O(1)}$ have* poly log AC^0 *proof systems.*

For the language MAJ of strings with at least as many 1s as 0s, and in general for threshold languages TH^n_k of strings with at least k 1s, we know that there are constant-width branching programs, but these are not structured in the sense above. It can be shown that a structured BP for MAJ must have width $\Omega(n)$ (a family of growing automata M_n for MAJ, where M_n is guaranteed to be correct only on $\{0,1\}^n$, must have $1 + n/2$ states in M_n). This is much much more than the poly log width bound used in the construction in Theorem 2.29. Nevertheless, we show below how we can modify that construction to get a poly log AC^0 proof system even for threshold languages.

Theorem 2.38. *For every n and $t \le n$, the language TH^n_t has a* poly log AC^0 *proof system.*

Proof. We follow the approach in Theorem 2.29: the input to the proof system is a word $a = a_1, \ldots, a_n$ and auxiliary information in the interval tree allowing us to correct the word if necessary. The labeling of the tree is different for this language, and is as follows. Each interval $(i, j]$ in the tree gets a label which is an integer in the range $\{0, 1, \ldots, j - i\}$. The intention is that for an input $a = a_1, \ldots, a_n$, the label of interval $(i, j]$ is **no more than** the number of 1s in the subword $a_{i+1} \ldots a_j$ formed at the leaves of the tree rooted at $(i, j]$. At a leaf node $(k-1, k]$, we do not give explicit labels; a_k serves as the label. At the root also, we do not give an explicit label; the label t is hard-wired. (We restrict the label of any interval $(i, j]$ to the range $[0, j - i]$, and interpret larger numbers as $j - i$.)

For any node u of T, let $l(u)$ denote the label of u. A node u with children v, w is **consistent** if $l(u) \le l(v) + l(w)$.

The output of our proof system $y_1, \ldots, y_n$ is constructed as follows:

- If all nodes on the path from $(k-1, k]$ to the root in T are consistent, then $y_k = a_k$.
- Otherwise, $y_k = 1$.

An example input and output are shown in Figure 2.2 for the proof system we constructed that generates the language $\text{TH}^n_{t=n/2}$. In the figure, the input is x along with the values for

the nodes in the interval tree constructed above x. The length of x is 16 and hence we hardwire the root node with label 8. The input x shown is providing a wrong count of 7 at the node corresponding to the interval $[8, 16]$. Our proof system outputs string y which has a 1 for every bit in the subtree under this node, hence making sure the output has at least eight 1s.

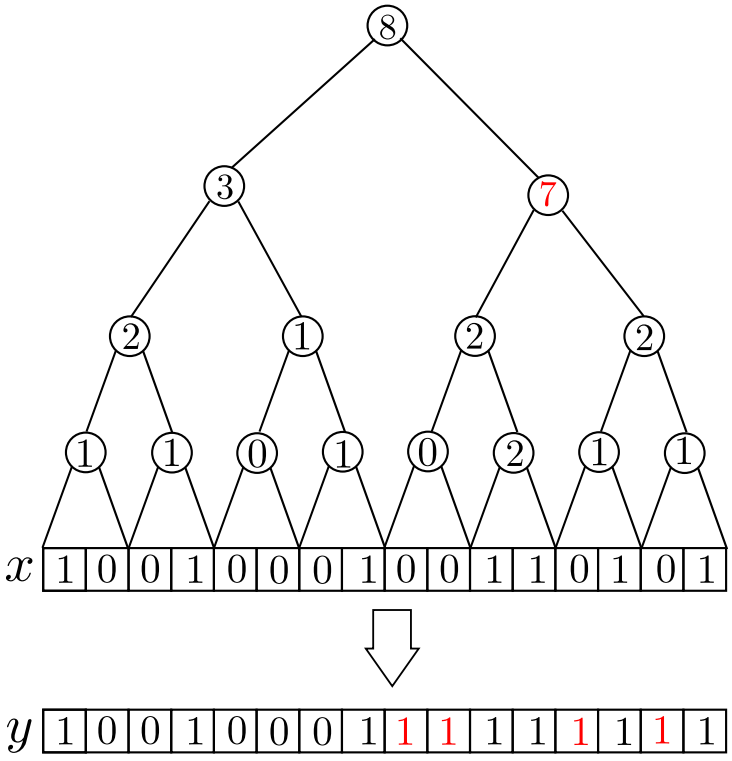


Figure 2.2: Example input/output of proof system for TH_8^{16}

In analogy with Theorem 2.29, we use here for each interval $(i, j]$ the feasibility witness 1^{j-i}, independent of the actual labels. Thus the construction forces this property: at a node u corresponding to interval $(i, j]$ labelled $\ell(u)$, the subword $y_{i+1}, \ldots, y_j$ has at least $\min\{\ell(u), j-i\}$ 1s. Thus, the output word is always in TH_t^n. Every word in TH_t^n is produced by the system at least once, on the proof that gives, for each interval other than $(0, n]$, the number of 1s in the corresponding subword.

As before, the $\text{CONSISTENT}_{i,j}$ predicate at a node depends on 3 labels, each of which is $O(\log n)$ bits long. A truth-table implementation is not good enough; it will give an AC^0 circuit. But the actual consistency check only involves adding and comparing $m = \log n$ bit numbers. Since addition and comparison are in AC^0, this can be done in depth $O(\log m)$ with $O(1)$ alternations. Thus the overall depth is $O(\log \log n)$. □

Corollary 2.39. *For every n and $t \le n$, EXACT-COUNT_t^n has a* poly log AC^0 *proof system.*

Proof. We follow the same approach as Theorem 2.38. We redefine **consistent** as follows: For any node u of T, let $l(u)$ denote the label of u. A node u with children v, w is consistent if $l(u) = l(v) + l(w)$. Let the output of our proof system be $y_1, \ldots, y_n$. The construction is as follows:

- If all nodes on the path from $(k-1, k]$ to the root in T are consistent, then $y_k = a_k$.
- Otherwise, let $u = (p, q]$ be the topmost node along the path from $(k-1, k]$ to the root that is not consistent. We output $y_k = 1$ if $k - p \leq l(u)$, 0 otherwise.

That is, for $u = (i, j]$ labeled $\ell(u)$, if $L = \min\{\ell(u), j - i\}$, use feasibility witness $1^L 0^{j-i-L}$. □

Combining the statements of Theorem 2.9 and Corollary 2.39, we can see that the depth of any proof system for Exact-Or has to be $\Theta(\log \log n)$. A depth upper bound of $O(\log \log n)$ for Exact-Or can be seen more directly as follows: Take as input a number $i \in [n]$. Output 1 at the i'th output position and 0s elsewhere. Depth $\log \log n$ suffices for this construction because each output bit y_j is 1 if and only if $i = j$.

2.4 Conclusion

We developed techniques to construct proof systems computable by NC^0 circuit families for a variety of languages.

The main open question that arises at this point is a combinatorial characterization of all languages that admit NC^0 proof systems. Our generic results from Section 2.2.2 and 2.2.3 can be seen as a first step towards such a characterization for regular languages. We believe that further progress essentially depends on developing strong lower bound techniques. We start studying lower bound techniques in the next chapter.

Our construction from Theorem 2.29 can be generalized to work for languages accepted by growing-monoids or growing-non-uniform-automata with poly-log growth rate (see eg [BLM93]). Can we obtain good upper bounds for linearly growing automata?

Chapter 3

Lower bounds on depth of proof systems

In this chapter we show explicit examples of languages that do not admit NC^0 proof systems, some of them even regular. In section 3.1, we generalize the counting technique used in Theorem 2.9 to derive lower bounds on the depth of proof systems for other languages. We then show, in section 3.2, that a similar counting technique works for Majority, the set of all strings with at least as many 1s as 0s, if we restrict the proof circuits further to have the property that each input bit is connected to only $O(1)$ output bits. To show a lower bound on depth for Majority against proof systems without this restriction, we use a more involved counting done over many stages. This is described in section 3.3.

3.1 Generalizing Exact-Or

We have already seen an example (Theorem 2.9) of a language that cannot be generated by an NC^0 proof system. We now generalize the technique used in the proof of Theorem 2.9 to derive a criterion which implies non-constant lower bounds for the depth of an enumerating circuit family.

Theorem 3.1. *Let L be a language and $\ell, t\colon \mathbb{N} \to \mathbb{N}$ functions. Suppose for each length n where $L^{=n}$ is non-empty, there is a set W of $t(n)$ distinct strings $W = \{w_1, \ldots, w_{t(n)}\} \subseteq L^{=n}$ satisfying the following: For each $w \in W$, there exists a set $S = \{i_1, \ldots, i_{\ell(n)}\} \subseteq [n]$ of $\ell(n)$ positions such that if x is in $L^{=n}$, and if x agrees with w on S, then x equals w. That is, the bits of w in positions indexed by S fix all the remaining bits. Then the depth of any*

bounded fan-in circuit family that enumerates L is at least $\log\log t(n) - \log \ell(n)$.

Proof. Let $f : \{0,1\}^{m(n)} \longrightarrow \{0,1\}^n$ be a depth-$d(n)$-circuit enumerating the length n members of L, and let $\ell(n)$ and $t(n)$ be as in the statement of the theorem. Denote the resulting words $w_1, \ldots, w_{t(n)}$.

For each of the w_j the following holds: The $\ell(n)$ crucial bits have paths to at most $r(n) = \ell(n)2^{d(n)}$ bits of the proof. Thus there is a setting to $r(n)$ bits of the proof, all extensions of which generate the same output w_j. Hence $|f^{-1}(w_j)| \geq 2^{m(n)-r(n)}$.

Now we just count the number of proofs. As there are $m(n)$ proof bits,

$$2^{m(n)} = \text{number of proofs} \geq \sum_{j=1}^{t(n)} \text{number of proofs for } w_j \geq t(n)2^{m(n)-r(n)}$$

and hence

$$2^{r(n)} \geq t(n); \quad \ell(n)2^{d(n)} = r(n) \geq \log t(n); \quad d(n) \geq \log\log t(n) - \log \ell(n).$$

□

Using this theorem, we can show that several functions are not enumerable in constant depth.

Exact Counting. Consider the function Exact-Count^n_k on n bits: it evaluates to 1 if and only if exactly k of the input bits are 1. (Exact-Or^n is precisely Exact-Count$^n_1$.) For each length n there are exactly $\binom{n}{k}$ words in Exact-Count^n_k. And whenever k bits of a word are set to value 1, then all remaining bits are bound to take the value 0. So for Exact-Count^n_k the parameters $t(n)$ and $\ell(n)$ defined in the theorem above take the values $\binom{n}{k}$ and k, respectively, which yields a lower bound of

$$d(n) = \log\log\binom{n}{k} - \log k \geq \log\log\left(\frac{n^k}{k^k}\right) - \log k = \log(\log n - \log k)$$

on the depth of an enumerating circuit family. For $k(n)$ sub-linear in n this gives an unbounded function; thus for every sub-linear $k(n)$, Exact-Count^n_k does not admit an NC^0 proof system. Note that for a constant k, this language is even regular.

The threshold functions $\neg\mathrm{Th}^n_{k+1}$ and dually Th^n_{n-k} for sub-linear k. Let Th^n_a be the function that evaluates to 1, if and only if at least a of the n inputs are set to 1. The lower bounds for these languages are derived precisely by the same argument given above for $\textsc{Exact-Count}^n_k$. So they also yield the same set of parameters.

In more detail: Strings in Th^n_{n-k} (or $\neg\mathrm{Th}^n_{k+1}$) can have at most k 0s (at most k 1s, respectively). There are $t(n) = \binom{n}{k}$ ways of choosing $l(n) = k$ positions from $[n]$; for each such choice, setting the bits in the chosen positions to 0 (1, resp.) forces all other bits to be 1 (0, resp.).

The language 0^*1^* and iterations. First consider 0^*1^*, whose members consist of a (possibly empty) block of 0's followed by a (possibly empty) block of 1's. The $n+1$ length-n members of 0^*1^* are in 1-1 correspondence to the members of $\textsc{Exact-Count}^{n+1}_1$ via the NC^0 mapping $w_1 \dots w_n \longmapsto x_1 \dots x_{n+1}$, where $x_i := w_{i-1} \oplus w_i$, with the convention that $w_0 := 0$ and $w_{n+1} := 1$. Thus an NC^0 proof system of 0^*1^* would directly yield one for $\textsc{Exact-Count}^{n+1}_1$, which we have shown to be impossible. The parameters from the theorem are $\ell(n) = 2$ (two consecutive bits with different values or simply $w_1 = 1$ or $w_n = 0$) and $t(n) = n + 1$. By the same argument, for sub-linear k, the languages consisting of either exactly or up to k alternating blocks of 0's and 1's do not admit NC^0 proof systems.

Majority: The language Majority (same as $\mathrm{Th}^n_{n/2}$) consists of those words which have at least as many 1's as 0's. We show in the following section that if we restrict the power of proof systems to an even weaker class than NC^0, then a simple counting argument gives us a lower bound on the depth of such restricted circuits

3.2 Constant influence

Motivated by their investigation into NC^0 cryptography [AIK06, AIK08], Applebaum et al. [AIK09] investigate cryptography with constant input locality. As a related question we ask which languages can be proven by circuits that have the property that every input bit is connected to at most $O(1)$ many output bits.

In the remainder of this section, we look at proof circuits that are NC^0 like before, but with the added restriction that each input bit can only influence at most $O(1)$ many output bits. We show that $\textsc{Exact-Count}^n_k$ and Th^n_k do not have proof circuits of small depth with

this added restriction even when k is allowed to be linear in n.

The proof is based on the following observation about any proof circuit for Exact-Count$_k^n$: Let C be a proof circuit for Exact-Count$_k^n$. For an output gate i of C we denote by $\sup(i)$ the set of all input gates of C that have a path to i. For a set S of output gates of C we let $\sup(S) = \bigcup_{i \in S} \sup(i)$. Now, for any set of output positions $S \subseteq [n]$, $|S| = k$ and $i \in [n], i \notin S$, we have $\sup(S) \cap \sup(i) \neq \emptyset$. If this were not true, then we could obtain $(k+1)$ 1s in the output by setting the bits in $\sup(S)$ to get k 1s corresponding to the positions in S, and by setting the $\sup(i)$ to get a 1 in the ith output position.

The above can be generalized to the following:

Lemma 3.2. *For $n \geq c2^d + k$, the language Exact-Count$_k^n$ does not have a proof circuit of depth d with each input bit influencing at most c output bits.*

Proof. Suppose such a circuit exists, take any output position $i \in [n]$. We know that $|\sup(i)| \leq 2^d$. Let T be the set of all output bits j for which $\sup(i) \cap \sup(j) \neq \emptyset$. $|T| \leq c2^d$. Now if $n \geq c2^d + k$, then we can find a set $S \subseteq [n]$ of output positions such that $|S| = k$ and $S \cap T = \emptyset$. This implies that $\sup(S) \cap \sup(i) = \emptyset$, contradicting the observation made above. □

Corollary 3.3. *The language Exact-Count$_{n/2}^n$ does not have a proof system of constant depth and constant influence.*

A similar observation as above holds for threshold functions as well: Let C be a proof circuit for Th$_k^n$. Then, for any subset of output positions $S \subseteq [n]$, $|S| = n - k$ and any $i \notin [n]$, we have $\sup(S) \cap \sup(i) \neq \emptyset$. If this were not true, then we can force C to output $n - k + 1$ 0s by setting the support of S and the support of i such that we get 0s in all the S positions and position i.

Lemma 3.4. *The function Th$_k^n$ does not have a proof circuit of depth d with each input bit influencing at most c output bits if $n > k \geq c2^d$.*

Proof. Suppose such a circuit exists, call it C. Take any output position $i \in [n]$. We know that $|\sup(i)| \leq 2^d$. Let T be the set of all output bits which have a support bit in $\sup(i)$. $|T| \leq c2^d$. Now since $k \geq c2^d$, we can find a set of output positions $S \subseteq [n]$ with $|S| = n - k$ such that $S \cap T = \emptyset$. Since T was all the bits that are influenced by $\sup(i)$, we have $\sup(S) \cap \sup(i) = \emptyset$. The above observation can be used to conclude that C can be forced to output a string that has more than $n - k$ 0s. □

We have the following immediate corollary:

Corollary 3.5. *Majority does not have a proof circuit family with constant depth and constant influence.*

Majority does not admit an NC^0 proof system. But this does not follow from an extension of the techniques described so far, and requires a completely different and significantly more non-trivial approach. We describe this in the following section.

3.3 Majority does not admit NC^0 proof systems

The language MAJ consists of all 0-1-words that contain at least as many 1's as 0's. The language EXMAJ consists of all 0-1-words w that contain exactly $\lceil |w|/2 \rceil$ 1's. Clearly, EXMAJ $\subseteq$ MAJ. If w is in EXMAJ, and if a single bit in w is flipped *from 1 to 0*, then the resulting string w' is not in MAJ. We will exploit this to show that MAJ does not admit an NC^0 proof system.

Intuitive Idea

Assume that there is an NC^0 proof system for MAJ. The idea of the proof is that there are two types of inputs: inputs that influence a linear number of outputs – call these the high-fanout inputs, and inputs that influence a sublinear number of outputs – these are the low-fanout inputs. (Note our non-standard use of the term *fanout* which refers to the number of output bits an input is connected to instead of the number of wires leaving the gate.) Since every output is connected to a constant number of inputs, there can only be a constant number of high-fanout inputs. So nearly all inputs are of low-fanout. However, by Corollary 3.5, we know that not all input positions can be low fanout.

We will try to find an output x_i whose value can be changed by manipulating only the set S of low-fanout inputs connected to x_i. Also, since low-fanout inputs are only connected to a sublinear number of outputs, we can assume that S is connected to less than $n/2$ of the outputs. So we can find a word w in MAJ that has a 1 at every position that depends on the input bits of S and assign the remaining outputs in such a way that we even get a word w in EXMAJ. Since this is a valid word in MAJ, the proof system needs to generate it, hence there is an assignment of the inputs that outputs w.

But now we can modify the input bits in S and toggle x_i to the value 0. Toggling the input bits in S only affects output bits that were assigned to 1, hence this might flip additional

bits from value 1 to value 0. But the word generated in this way by the proof system has fewer 1's than w and hence is not in MAJ. It follows there is no NC^0 proof system for MAJ.

Formalising this idea is a bit more complicated. It turns out that we need a finer gradation of what we consider high-fanout. We will define a decreasing function $g : \mathbb{N} \longrightarrow \mathbb{N}$, and at stage e, we consider an input connected to more than $g(e)$ output bits as high-fanout. Say that X_e is the set of high-fanout inputs at stage e. If we can find an output x_i as above, we will have obtained a contradiction. But we may not immediately succeed in finding such an x_i, because it may be the case that settings to the high-fanout bits X_e fix each output x_i. We then carefully fix a small set R_e of output bits and an assignment w^e to these bits in a way such that each output outside of R_e can be toggled without changing the input setting to X_e. At this point, we look for a string in EXMAJ compatible with w^e, and try to obtain a contradiction by toggling a carefully chosen output. At any stage e, we say that an assignment for a subset of the input positions as "compatible" with w^e if in the positions R_e, the output is w^e. If we still cannot obtain a contradiction, we move on to the next stage. Finally, we show that if we complete stage c for some suitably chosen constant c, then we get a different kind of contradiction: a few high-fanout input bits completely determine many output bits. A simple counting argument shows that this cannot happen for MAJ.

To make this argument rigorous, we define a certain assertion Π_e concerning stage e. This assertion states that there is a setting w^e to a set R_e of output bits satisfying 4 properties: (1) R_e is small, (2) assignments to X_e compatible with w^e do not fix any output bit outside R_e, (3) the forbidden set F_{e+1}, consisting of output bits sharing a low-fanout input with some bit in R_e, is small, and (4) every non-forbidden-output is connected to at least one input that will enter the high-fanout set at stage $e + 1$. Then the above argument can be rephrased as: Π_0 is true, $\Pi_e \Rightarrow \Pi_{e+1}$, but at least one of $\Pi_0, \ldots, \Pi_{c-1}$ is not true. This is obviously a contradiction.

With this idea in mind, we now state and prove our theorem.

Theorem 3.6. *The language* MAJ *does not admit an* NC^0 *proof system.*

Proof. Assume that (C_n) is an NC^0 family of circuits enumerating MAJ. Let d be the maximal depth of the circuits C_n and let $c \leq 2^d$ be the maximum number of input bits connected to the same output bit. It is easy to see that no projection (each output bit is either some input bit or the negation of some input bit) can be a proof system for MAJ. Hence $c \geq 2$.

Let In and Out denote the sets of input and output bits of C_n, respectively. For a set A of

nodes of the circuit, define the sets

$$\begin{aligned} \mathrm{Out}(A) &:= \{y \in \mathrm{Out} \mid y \text{ is connected to some } x \in A\} \\ \mathrm{In}(A) &:= \{x \in \mathrm{In} \mid x \text{ is connected to some } y \in A\} \end{aligned}$$

For a singleton $\{x\}$ whose only element is an input/output bit, we simply write $\mathrm{Out}(x)$ or $\mathrm{In}(x)$.

Define functions f, g as follows:

$$\begin{aligned} f(e) &:= \begin{cases} 1 & \text{for } e = 0 \\ c^{5f(e-1)+1} & \text{for } e > 0 \end{cases} \\ g(e) &:= \frac{cn}{f(e)} \end{aligned}$$

Clearly, f is an increasing function, and g is a decreasing function. Note that f does not depend on the value of n. All arguments in the proof will work for a choice of $n \geq 4 \cdot f(c)$. We use g to define the high-fanout set at each stage;

$$X_e := \left\{ x \in \mathrm{In} \;\middle|\; |\mathrm{Out}(x)| > g(e) \right\}$$

Note that for each e, $X_{e-1} \subseteq X_e$. Also, since there are at most cn input-output-connections in circuit C_n, and since each input bit in X_e contributes more than $g(e)$ input-output connections, we obtain

$$|X_e| \cdot g(e) < (\text{number of input-output connections in } C_n) \leq cn = f(e) \cdot g(e).$$

Thus the function $f(e)$ yields an upper bound for the size of X_e.

We now state an assertion concerning the circuit C_n, for a parameter e; call this assertion Π_e.

Assertion 1 (Π_e). *There exists a set $R_e \subseteq$ Out, and a setting w^e to R_e, satisfying the following properties.*

1. $|R_e| \leq 2^{f(e)+1}$.

2. *for each $y \in \mathrm{Out} \setminus R_e$, for each assignment $q : X_e \to \{0,1\}$ compatible with w^e, and for each value $b \in \{0,1\}$, there is a legal configuration of the circuit extending $w^e \cup q$ and setting y to b.*

3. *Let* $F_{e+1} := R_e \cup \text{Out}(\text{In}(R_e) \setminus X_{e+1})$. *($F_{e+1}$ denotes the set of* forbidden *outputs.) Then* $|F_{e+1}| \leq \frac{n}{c^3}$.

4. $\forall y \in \text{Out} \setminus F_{e+1}$, $\text{In}(y) \cap (X_{e+1} \setminus X_e) \neq \emptyset$.

We prove the theorem by contradiction. Assuming that (C_n) enumerates MAJ, we will show that for all sufficiently large n, the following statements hold:

(A). Π_0 is true.

(B). $\Pi_0, \Pi_1, \ldots, \Pi_{c-1}$ are not simultaneously true.

(C). For all $1 \leq e < c$, $\Pi_{e-1} \implies \Pi_e$.

With these statements established in Lemmas 3.7, 3.8, and 3.9 below, we reach a contradiction, and the proof of the main theorem is complete. □

Proof of the Statements (A)-(C)

Lemma 3.7 (Statement (A))**.** Π_0 *is true.*

Proof. Note that $X_0 = \emptyset$. Define $R_0 = \emptyset$, $w^0 = \epsilon$. Then $F_1 = \emptyset$. Properties 1,3 are trivial. Property 2 holds because no bit in MAJ is fixed; each y can take values 0 and 1.

It remains to show Property 4. Suppose Property 4 fails; that is, there is an output bit y with no connections to X_1. Then the neighbourhood of y, defined as $N(y) = \text{Out}(\text{In}(y))$, has size at most $c \times g(1) = n/c^4 < n/2$. So there exists a string z in EXMAJ with only 1s at members of $N(y)$, and hence a configuration β of the circuit compatible with z. By changing the input settings in β only in $\text{In}(y)$, we can set output y to 0. Since $\text{In}(y)$ reached only positions set to 1 in β, the change strictly decreases the number of 1s in the output. Thus C_n outputs a string not in MAJ, a contradiction. Hence Property 4 must hold. □

Lemma 3.8 (Statement (B))**.** $\Pi_0, \Pi_1, \ldots, \Pi_{c-1}$ *are not simultaneously true.*

Proof. Assume to the contrary that for each $e \in \{0, 1, \ldots, c-1\}$, Π_e is true. Define $F = \cup_{e=1}^{c} F_e$, and let $G := \text{Out} \setminus F$ denote the remaining output bits.

Consider any output bit $y \in G$. For each $e \in [c]$, y is not in F_e, so by property 4 in Π_{e-1}, $\text{In}(y)$ has a bit in $X_e \setminus X_{e-1}$. Thus $\text{In}(y)$ has at least c bits in X_c. But $\text{In}(y)$ has at most c bits overall, so $\text{In}(y)$ is in fact completely contained in X_c.

By property 3 of each Π_e, we know that $|G| \geq n - c(n/c^3) = n - n/c^2 \geq 3n/4$. As remarked earlier, $f(e)$ is an upper bound on $|X_e|$, and so $|X_c| < f(c)$. We saw above that for each $y \in G$, $\text{In}(y) \subseteq X_c$. Thus, in legal configurations of the circuit, the assignment to G is determined by the assignment to X_c. But there are less than $2^{f(c)}$ distinct assignments to X_c, while there are at least $\binom{|G|}{\frac{n}{4}} \geq 2^{n/4}$ distinct assignments to G corresponding to strings in MAJ. For sufficiently large n, this is impossible. □

Lemma 3.9 (Statement (C))**.** *For all* $1 \leq e < c$, $\Pi_{e-1} \Longrightarrow \Pi_e$.

Proof. To show that Π_e holds, we first describe a procedure that extends R_{e-1} and w^{e-1} to R_e and w_e, and then show that the extension satisfies properties 1 to 4 of Π_e. The immediate objective of the extension procedure is to satisfy property 2 of Π_e; control the input bits in $X_e \setminus X_{e-1}$ by restricting the output to a configuration that does not allow the X_e part of the input to fix further output bits.

Set $R = R_{e-1}$ and $w = w^{e-1}$. Define the set Q as follows.

$$Q := \left\{ q \in \{0,1\}^{X_e} \;\middle|\; q \text{ is compatible with } w \right\}.$$

Perform the Prune procedure described below.

The Prune Procedure: Perform the following step as long as possible.

> Find a partial assignment $q \in Q$, a position y in $\text{Out} \setminus R$, and a value $b \in \{0,1\}$ such that all configurations of the circuit extending q set y to b. Add y to R, set y to $\overline{b}$ in w making w incompatible with q, and remove from Q all assignments (including q) that are incompatible with w.

After the Prune procedure terminates, set R_e to the resulting R, and w^e to the resulting w.

The four claims below show that this choice of R_e and w^e satisfies Π_e. First, we state a simple but important observation: After every step in the Prune procedure, Q satisfies

$$Q = \left\{ q \in \{0,1\}^{X_e} \;\middle|\; q \text{ is compatible with } w \right\}.$$

In connection with the first property stated in Π_e and proven below, this implies that Q is not empty as long as $n > 2^{f(e)+2}$.

Claim 3.10 (Property 1 of Π_e holds)**.** $|R_e| \leq 2^{f(e)+1}$.

Proof. We start with $|Q| = 2^{|X_e|}$ and $R = R_{e-1}$. Each time we add a position to R, we discard at least one element from Q. So $|R_e| \leq |R_{e-1}| + 2^{|X_e|}$.

Using the fact that $f(e)$ is an upper bound for $|X_e|$, the property 1 of Π_{e-1}, and the definition of f, we get: $|R_e| \leq |R_{e-1}| + 2^{|X_e|} \leq 2^{f(e-1)+1} + 2^{f(e)} \leq 2^{f(e)+1}$. □

Claim 3.11 (Property 2 of Π_e holds)**.** *For each $y \in \text{Out} \setminus R_e$, for each assignment $q : X_e \to \{0,1\}$ compatible with w^e, and for each value $b \in \{0,1\}$, there is a legal configuration of the circuit extending $w^e \cup q$ and setting y to b.*

Proof. Recall that, the way the Prune procedure is defined, all settings $q : X_e \longrightarrow \{0,1\}$ compatible with w^e are in Q, and none of them determine the bit at any position $y \in \text{Out} \setminus R_e$. Hence for any such y, and any value b, it is possible to extend $q \cup w^e$ and set the bit at position y to b. □

Claim 3.12 (Property 3 of Π_e holds)**.** *For sufficiently large n, $|F_{e+1}| \leq n/c^3$.*

Proof. Recall that $F_{e+1} := R_e \cup \text{Out}(\text{In}(R_e) \setminus X_{e+1})$.

$$\begin{aligned}
\frac{n}{c^3} - |F_{e+1}| &\geq \frac{n}{c^3} - (|R_e| + c \cdot |R_e| \cdot g(e+1)) \\
&= \frac{n}{c^3} - |R_e|\left(1 + c \cdot \frac{nc}{f(e+1)}\right) \\
&\geq \frac{n}{c^3} - 2^{f(e)+1} \cdot \left(1 + \frac{nc^2}{f(e+1)}\right) \quad \text{using Claim 3.10} \\
&= n \cdot \left(\frac{1}{c^3} - \frac{c^2 2^{f(e)+1}}{f(e+1)}\right) - 2^{f(e)+1} \\
&= \delta_e n - 2^{f(e)+1}
\end{aligned}$$

It suffices to show that $\delta_e > 0$, because then we can choose a sufficiently large n and ensure that $\delta_e n$ exceeds $2^{f(e)+1}$. (Note, for $e \leq c$, δ_e and $f(e)$ are constants independent of n.)

$$\begin{aligned}
\delta_e > 0 &\iff \delta_e c^3 f(e+1) > 0. \\
\delta_e c^3 f(e+1) &= f(e+1) - c^5 2^{f(e)+1} \\
&= c^{5f(e)+1} - c^5 2^{f(e)+1} \\
&\geq 2^{f(e)+1} c^{4f(e)} - c^5 2^{f(e)+1} \quad \text{since } c \geq 2 \\
&= 2^{f(e)+1}\left[c^{4f(e)} - c^5\right] \\
&> 0 \quad \text{since } e \geq 1 \text{ and } c \geq 2, 4f(e) \geq 5.
\end{aligned}$$

□

Alternative proof. Using Claim 1, the definitions of X_{e+1}, g and f, and the facts that $c \geq 2$ and $1 \leq e \leq c-1$ along with the choice of $n \geq 4f(c)$ we get

$$\begin{aligned}
|F_{e+1}| &\leq |R_e| + |\operatorname{Out}(\operatorname{In}(R_e) \setminus X_{e+1})| \\
&\leq 2^{f(e)+1} + g(e+1) \cdot 2^{f(e)+1} \cdot c \\
&= 2^{f(e)+1} + \frac{cn}{f(e+1)} \cdot 2^{f(e)+1} \cdot c \\
&\leq c^{f(c-1)+1} + \frac{c^2 n 2^{f(e)+1}}{c^{5f(e)+1}} \\
&\leq \frac{f(c)}{c^4} + \frac{c^2 n}{c^6} \cdot \frac{2^{f(e)+1}}{c^{f(e)+1}} \\
&\leq \frac{n}{c^4} + \frac{n}{c^4} = \frac{n}{c^3}
\end{aligned}$$

□

Claim 3.13 (Property 4 of Π_e holds)**.** *For sufficiently large n, $\forall y \in \operatorname{Out} \setminus F_{e+1}$, $\operatorname{In}(y) \cap (X_{e+1} \setminus X_e) \neq \emptyset$.*

Proof. Suppose the claim does not hold. That is, suppose there exists a $y \in \operatorname{Out} \setminus F_{e+1}$ such that $\operatorname{In}(y) \cap (X_{e+1} \setminus X_e) = \emptyset$. But note that $\operatorname{In}(y) \cap \operatorname{In}(R_e) \subseteq X_{e+1}$; otherwise y would have been in F_{e+1} by definition. Putting these together, we conclude that $\operatorname{In}(y) \cap \operatorname{In}(R_e) \subseteq X_e$. Generalising the corresponding argument used in establishing statement (A), we will now show that this is not possible.

Consider the e-neighbourhood of y defined as $U := \operatorname{Out}(\operatorname{In}(y) \setminus X_e)$. Since $y \notin F_{e+1}$, the sets U and R_e are disjoint. We have that $|U| \leq n/c^4$, since by definition of f, for $e > 0$, $f(e) \geq c^6$, and hence $|U| \leq c \cdot g(e) \leq c^2 n / f(e) \leq n/c^4$. Together with Claim 3.10, for sufficiently large n, $|R_e \cup U| < n/2$. Hence there exists a string z in ExMaj with only 1s at positions in U, and agreeing with w^e at positions in R_e. Hence there is a configuration β of the circuit compatible with z; let this configuration restricted to X_e be α. (Thus β extends $\alpha \cup w^e$.) We have already established property 2 of Π_e. Applying this to y and α with $b = 0$, we conclude that there is another configuration γ, also extending $\alpha \cup w^e$, such that γ has a 0 at y.

Now change the input settings of β only at positions in $\operatorname{In}(y) \setminus X_e$ to match the settings in γ. This change can affect only the output bits in U. In particular, it changes output y from 1 to 0. Since U had only 1s in β, the change strictly decreases the number of 1s in the output. Thus C_n outputs a string not in Maj, a contradiction. □

With Claims 3.10, 3.11, 3.12, 3.13, Lemma 3.9 is established. □

We note here that the lower bound on depth that we just showed for Majority is only $\omega(1)$, while the upper bound on depth for the proof system that we constructed for Majority (Theorem 2.38) has a depth of $O(\log \log n)$. Closing this gap between the lower bound and the upper bound is an interesting open problem.

Recall, from section 3.1, that we generalized Theorem 2.9 to Exact-Count$_k^n$ only for sub-linear k. We will now show that the Majority lower bound from Theorem 3.6 can be used to show that Exact-Count$_k^n$ does not have NC^0 proof systems even for $k = n/2$.

We first make an observation for languages, like Majority, that are derived from monotone boolean functions. Recall that a function f is monotone if whenever $f(x) = 1$ and y dominates x (that is, $\forall i \in [n]$, $x_i = 1 \Rightarrow y_i = 1$), then it also holds that $f(y) = 1$. For such a function, a string x is a *minterm* if $f(x) = 1$ but x does not dominate any z with $f(z) = 1$. Minterms(f) denotes the set of all minterms of f. Clearly, Minterms(f) $\subseteq f^{-1}(1)$. The following lemma observes that for any monotone function f, constructing a proof system for a language that sits in between Minterms(f) and $f^{-1}(1)$ suffices to get a proof system for $f^{-1}(1)$.

Lemma 3.14. *Let $f : \{0,1\}^* \longrightarrow \{0,1\}$ be a monotone boolean function and let $L = f^{-1}(1)$. Let $L_n = L \cap \{0,1\}^n$. Let L' be a language such that for each length n, (Minterms$(L) \cap \{0,1\}^n) \subseteq (L' \cap \{0,1\}^n) \subseteq L_n$. If L' has a proof system of depth d, size s and a alternations, then L has a proof system of depth $d+1$, size $s+n$ and at most $a+1$ alternations.*

Proof. Let C be a proof circuit for L' that takes input string x. We construct a proof system for L using C and asking another input string $y \in \{0,1\}^n$. The i'th output bit of our proof system is $C(x)_i \vee y_i$. □

From here on, for any language A, we let UpClose(A) denote the language $B = \{y : \exists x \in A, |x| = |y|, \forall i, x_i = 1 \implies y_i = 1\}$.

Lemma 3.15. *The following languages do not have NC^0 proof systems.*

1. ExMaj*, consisting of strings x with exactly $\lceil |x|/2 \rceil$ 1s.*
2. ExMajEven*, consisting of all even length strings in* ExMaj
3. EqualOnes $= \{xy \mid x, y \in \{0,1\}^*, |x| = |y|, |x|_1 = |y|_1\}$.

Proof. 1. To show that ExMaj does not have NC^0 proof systems, note that:

- From 3.6, the language MAJ does not have NC^0 proof systems
- Minterms(MAJ) = EXMAJ.
- Lemma 3.14 now implies EXMAJ does not have an NC^0 proof system.

By the same argument, EXMAJ restricted to even-length strings, call it EXMAJEVEN, has no NC^0 proof systems.

2. We will show that if EQUALONES has an NC^0 proof system, then so does the language EXMAJEVEN. Consider the slice

$$\text{EQUALONES}^{=2n} = \{xy \mid |x| = |y| = n; \quad x \text{ and } y \text{ have an equal number of 1s}\}.$$

If x, y are length-n strings, then $xy \in \text{EQUALONES}^{=2n}$ if and only if $xy' \in \text{EXMAJEVEN}$, where y' is the bitwise complement of y. Thus a depth d proof system for EQUALONES implies a depth $d+1$ proof system for EXMAJEVEN.

□

Corollary 3.16. *The language* $\text{GI} = \{G_1, G_2 \mid \text{Graph } G_1 \text{ is isomorphic to graph } G_2\}$ *does not have an* NC^0 *proof system. Here we assume that* G_1 *and* G_2 *are specified via their 0-1 adjacency matrices, and that 1s on the diagonal are allowed (the graphs may have self-loops).*

Proof. Let G_1, G_2 be n-node isomorphic graphs with adjacency matrices A_1, A_2. Then (A_1, A_2) is in $\text{GI}^{=2n^2}$. Let y_b be the string appearing on the diagonal of A_b. Then $y_1 y_2 \in \text{EQUALONES}^{=2n}$.

Conversely, for each $xy \in \text{EQUALONES}^{=2n}$ where $|x| = |y| = n$, the pair $(\text{Diag}(x), \text{Diag}(y))$ is in $\text{GI}^{=2n^2}$. (For an n-bit vector w, $\text{Diag}(w)$ is the $n \times n$ matrix with w on the diagonal and zeroes elsewhere.)

Thus a depth d proof system for G implies a depth d proof system for EQUALONES.

□

3.4 Conclusion

In this chapter, we showed various languages that do not have even non-uniform NC^0 proof systems. All lower bounds seem to develop either on the fact that EXACT-OR does

not have NC^0 proof systems or that MAJ does not. We do not yet have a characterization of even regular languages with respect to NC^0 proof systems.

For MAJ, we have given a proof system with $O(\log \log n)$ depth and $O(1)$ alternations in Theorem 2.38, and we have shown that $\omega(1)$ depth is needed in Section 3.3. Can this gap between the upper and lower bounds be closed?

To answer the main question of characterizing languages with NC^0 proof systems, a possible tool is from Agrawal's result on constant-depth isomorphisms [Agr10]. If we have an NC^0 isomorphism between two languages A and B, and B admits an NC^0 proof system, then so does A. The proofs for A are taken to be the proofs for B, then we simulate the proof system for B, and to the obtained word in B we apply the isomorphism from B to A and enumerate an element from A.

In fact, our work seems to bear further interesting connections to recent examinations on isomorphism of complete sets for the class NP. This work was started in the nineties in a paper by Agrawal et al. [AAR98] where it was shown that (1) every language complete for NP under AC^0 reductions is in fact already complete under (non-uniform) NC^0 reductions (this is called "gap theorem" in [AAR98]), and (2) that all languages complete for NP under AC^0 reductions are (non-uniformly) AC^0 isomorphic (that is, the reduction is an AC^0 bijection). This was later improved to uniform AC^0 isomorphisms [Agr10]. It follows from a result in [AAI+01] that this cannot be improved to P-uniform NC^0 isomorphisms. Using our results on proof systems, we obtain a very simple direct proof:

Proposition 3.17. *There are sets A and B that are NP complete under NC^0 reductions but not NC^0 isomorphic.*

Proof. Let A be the NP-complete set from [CM01] that admits an NC^0 proof system, cf. Remark 2.35. A is NP complete under AC^0 reductions, hence by the gap theorem of [AAR98], is also NP-complete under NC^0 reductions.

Let B be the disjoint union of A and the language EXACT-OR.
i.e., $B = (\{0\} \circ A) \cup (\{1\} \circ \text{EXACT-OR})$. Then B is complete for NP under NC^0 reductions because A reduces to B in NC^0.

If now A and B are NC^0 isomorphic, then we obtain an NC^0 proof system P for B. From the proof system P, we can obtain an NC^0 proof system for EXACT-OR as follows: Let string $y = y_1 y_2 \cdots y_n$ be the output of P. If $y_1 = 1$, we output y as is. Else, we output 10^{n-1}. Hence we arrive at a contradiction using Theorem 2.9.

□

Chapter 4

2TAUT and NC^0 proof systems

In this chapter we try to understand NC^0 proof systems better in the context of 2TAUT - the language of all 2DNFs that are tautologies. The main goal is to understand the relationship between TAUT and NC^0 proof systems. However, the language 2TAUT has more structure because of the connection with implication graphs. In particular we know that 2TAUT and directed graph reachability are equivalent in computational complexity. In the following section, we will show that 2TAUT is at least as easy as directed graph reachability in terms of proof systems as well.

4.1 Directed Reachability and 2TAUT

Let UNSAT denote the language of all unsatisfiable formulas. Recall that deciding if a 2CNF formula F belongs to UNSAT can be done in polynomial time. This is because one can build the following "implication graph" G from F: G is a directed graph with twice as many vertices as number of variables in F - one vertex for each possible literal. Each clause $(\ell_i \vee \ell_j)$ of F can be written as two implications: $\overline{\ell_i} \implies \ell_j$ and $\overline{\ell_j} \implies \ell_i$. An edge (ℓ_i, ℓ_j) is present in G if the implication $\ell_i \implies \ell_j$ is in the formula F. i.e., for every term $(\ell_i \vee \ell_j)$ in the formula F, G will have two edges namely - $(\overline{\ell_i}, \ell_j)$ and $(\overline{\ell_j}, \ell_i)$. Let vertices u_i be associated with literals x_i and vertices v_i be associated with the literal $\overline{x_i}$. Now F is unsatisfiable if and only if there exists an i such that there is a path from u_i to v_i and a path from v_i to u_i. Checking existence of such paths can be done in polynomial time and hence deciding if F is unsatisfiable is in P. Let 2UNSAT denote all 2CNF formulas that are unsatisfiable. Note that since 2TAUT = $\{F | \overline{F} \in$ 2UNSAT$\}$, deciding if a 2DNF formula F is in 2TAUT is also in non-deterministic logspace and hence in P.

Recall that the encoding used by [CM01] in their construction of an NC^0 proof system for the NP-Complete problem E3SAT has exactly $2^3\binom{n}{3}$ bits - one bit for each possible clause (See Remark 2.35). Using a similar idea, we can encode 2CNF formulas on n variables by a bit string of $2n(2n-1)$ bits - one bit for each possible clause (each clause has two literals. For the first literal, we have $2n$ possible choices and for the second we have $2n-1$ possible choices). Note that here we are allowing for trivial clauses like $(x_i \vee \overline{x}_i)$. We will also assume that if the bit corresponding to a clause $l_i \vee l_j$ is 1, then the bit corresponding to the clause $l_j \vee l_i$ is also 1 and vice versa. This encoding would correspond to the adjacency matrix of the implication graph on $2n$ vertices described previously but without the diagonal entries. However, for convenience, we will generate adjacency matrices of the implication graphs along with the diagonal entries included. In other words, we allow for self-loops in the graph. This is equivalent to allowing clauses such as $(x_i \vee x_i)$ in the formula. Hence we work with this encoding of $4n^2$ bits for the remainder of this section. It is easy to go from the $4n^2$ bit encoding to the $2n(2n-1)$ bit encoding: merely hide the diagonal entries from being output.

Let STConn be the language of all n vertex graphs with a path from vertex 1 to vertex n. In the following, we will show a reduction between proof systems generating 2TAUT and STConn. More precisely, we show that if STConn has an NC^0 proof system, then so does 2TAUT.

Define the following languages:

$$\text{STConn}_n = \left\{ A \in \{0,1\}^{n\times n} \middle| \begin{array}{c} A \text{ is the adjacency matrix of a directed graph } G \text{ where} \\ \text{vertices } s = 0,\ t = n-1 \text{ are in the same connected} \\ \text{component.} \end{array} \right\}$$

$$\text{STConn} = \bigcup_{n>0} \text{STConn}_n$$

When handling graphs, throughout this section, we write $u \rightsquigarrow v$ to denote "$\exists$ a path from u to v" where u and v are vertices of the graph being used.

We will show that a proof system for the set STConn can be used to construct a proof system for 2UnSat with only a constant blowup to the depth of the proof system:

Theorem 4.1. *If* STConn *has an* NC^0 *proof system, then* 2UnSat *has an* NC^0 *proof system.*

Proof. Let Q be a proof system computable in NC^0 for STConn.

We first show that using output of circuits from Q, we can generate the language GoodConn

defined as follows:

$$\textsc{GoodConn} = \bigcup_i \textsc{Good}_i$$

where $\textsc{Good}_i$ is defined as:

$$\textsc{Good}_i = \{G \subseteq \textsc{STConn}_{2n+2} | \exists i \in [n], \exists \text{ simple path } 0 \to i \rightsquigarrow \bar{i} \to 2n+1\}$$

where $\bar{i}$ is short for $n + i$.

Lemma 4.2. *If* STConn *has an* NC^0 *proof system, then so does* GoodConn.

Proof. We construct proof system $\mathcal{GC}$ computable in NC^0 for GoodConn by using proof system $\mathcal{Q}$ of STConn. Let $Q \in \mathcal{Q}$ be the proof circuit that outputs adjacency matrices of graphs in $\textsc{STConn}_{2n+2}$. We number the vertices of the graph output by Q as $0, 1, 2, \ldots, (2n+1)$. Let $s = 0$ and $t = 2n + 1$ as shown in Figure 4.1. Let the adjacency matrix output by Q be H.

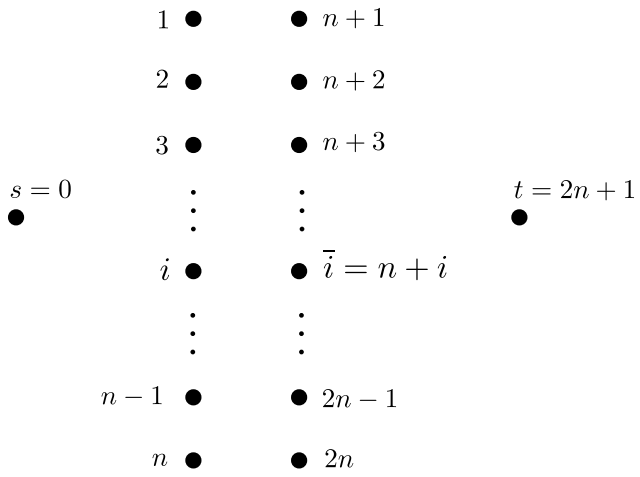


Figure 4.1: Vertex numbering

Construction: We will construct proof circuit $P \in \mathcal{GC}$ that outputs every graph in GoodConn on $2n + 2$ vertices. P takes the following as input: H and another adjacency matrix B. P outputs every edge in H, every edge in B and also some additional edges determined by the following rules:

1. If $H[s, t] = 1$, then include path $s \to 1 \to n + 1 \to t$.

2. For each $i \in [n]$, if $H[s, \bar{i}] = 1$, then include path $s \to i \to \bar{i} \to t$.

3. For each $i \in [n]$, if $H[i,t] = 1$, then include path $s \to i \to \bar{i} \to t$.

4. For each $i \in [n]$, if $H[s,i] = 1$, then add edge $(\bar{i} \to t)$.

5. For each $i, j \in [n]$ such that $i \neq j$, if $H[s,i] = H[\bar{j},t] = 1$, then add edge $(\bar{j}, \bar{i})$.

Intuitively, the vertices from 1 to n represent the literals $x_1, \cdots, x_n$ and the vertices from $n+1$ to $2n$ represent the literals $\bar{x}_1, \cdots, \bar{x}_n$. The idea behind the construction is to force a simple path $s \to i \rightsquigarrow \bar{i} \to t$ for some $i \in [n]$. However, if the graph H output by P was such that it had a simple path $s \to i \rightsquigarrow \bar{i} \to t$ and no other edges, then none of the rules apply. The extra input B is simply for upward closure (like in Lemma 3.14) to guarantee completeness.

Soundness: Let G be a graph output by P. We need to show that G is in GoodConn. It suffices to show the following:

Claim 4.3. *$\exists i \in [n]$ such that $G \in \text{Good}_i$*

Proof. Let $G = P(H, B)$. Since G has all edges from H and $H \in \text{STConn}_{2n+2}$, G has a simple path ρ from s to t. Consider the last edge (u, t) in ρ. We have the following cases:

- Case $u = s$: Rule 1 implies $G \in \text{Good}_1$.
- Case $u = i$ for some $i \in [n]$: Rule 3 implies $G \in \text{Good}_i$.
- Case $u = \bar{i}$ for some $i \in [n]$: Consider the first edge (s, v) in ρ. We have three cases:
  - Case $v = \bar{j}$ for some $j \in [n]$: Rule 2 implies $G \in GOOD_j$.
  - Case $v = i$: Straightforward to see $G \in GOOD_i$.
  - Case $v = j$ for some $j \in [n]$ and $j \neq i$: Rule 4 and Rule 5 together imply $G \in GOOD_j$. An example of this case is shown in Figure 4.2.

□

Completeness: We need to show that any graph $G \in$ GoodConn can be produced by P. We observe the following:

Observation 4.4. *Let $J \in \text{STConn}_{2n+2}$ be a graph that has a simple path $s \to i \rightsquigarrow \bar{i} \to t$ for some $i \in [n]$ and no other edges. If B is all 0s, then $P(J, B)$ is exactly J.*

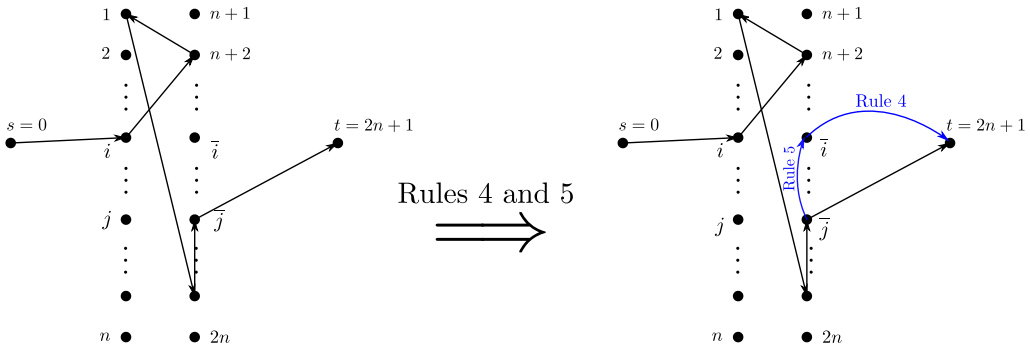


Figure 4.2: Example of Rules 4 and 5

From definition of GoodConn, $G \in \text{GoodConn}$ implies $G \in \text{Good}_i$ for some $i \in [n]$. Hence there exists a simple path ρ that proceeds as $s \to i \rightsquigarrow \bar{i} \to t$ in G. Consider a graph J with path ρ and no other edges. Then J is an output of Q. Let B be the $(2n+2) \times (2n+2)$ adjacency matrix of G. Then $P(J, B) = G$.

$\mathcal{GC}$ is computable in NC^0:

Lemma 4.5. *If Q is computable in NC^0, then $\mathcal{GC}$ is computable in NC^0.*

Proof. Assume Q is computable in NC^0. It suffices to show that each output in a proof circuit $P \in \mathcal{GC}$ is a function of only $O(1)$ of its input bits. P uses the output H of a proof circuit $Q \in Q$. Let A be the output adjacency matrix of P. To see that the rules determining the edges can be implemented in NC^0, we explicitly write down the circuit for each type of edge:

- (Rule 1) $A[s, 1] = H[s, t] \vee H[s, n+1] \vee H[1, t] \vee H[s, 1] \vee B[s, 1]$.
- (Rules 2 and 3)
  - For all $i \in [n] \setminus \{1\}$, $A[s, i] = H[s, \bar{i}] \vee H[i, t] \vee H[s, i] \vee B[s, i]$.
  - For all $i \in [n] \setminus \{1\}$, $A[i, \bar{i}] = H[s, \bar{i}] \vee H[i, t] \vee H[i, \bar{i}] \vee B[i, \bar{i}]$
- (Rules 2,3 and 4) For all $i \in [n]$, $A[\bar{i}, t] = H[s, i] \vee H[s, \bar{i}] \vee H[i, t] \vee H[\bar{i}, t] \vee B[\bar{i}, t]$.
- (Rule 5) For all $i, j \in [n]$, $i \neq j$, $A[\bar{j}, \bar{i}] = (H[s, i] \wedge H[\bar{j}, t]) \vee H[\bar{j}, \bar{i}] \vee B[\bar{j}, \bar{i}]$.

For all edges $e = (u, v)$ that do not fall under any of the above types, $A[e] = H[e] \vee B[e]$.

Thus each output bit is a function of at most 4 bits from H and 1 bit from B. H is the output of the proof circuit Q. Hence if proof system Q is an NC^0 circuit family, then so is $\mathcal{P}$. □

This completes the proof of Lemma 4.2. □

We define the following languages which have paths in the reverse direction as GoodConn:

$$\text{Good}_i^R = \{G \subseteq \text{STConn}_{2n+2} | \exists i \in [n], \exists \text{ simple path } 2n+2 \to \bar{i} \rightsquigarrow i \to 0\}$$

$$\text{GoodConn}^R = \bigcup_i \text{Good}_i^R$$

where the R in the superscript means "reversed".

Note that GoodConn^R can be generated using proof circuits from $\mathcal{GC}$ and either numbering the vertices of the output graph in reverse or by reversing the direction of each edge in the output graph. Let $\mathcal{GC}^R$ be such a proof system. The following is an easy observation:

Observation 4.6. *$\mathcal{GC}^R$ is computable in NC^0 if and only if $\mathcal{GC}$ is computable in NC^0.*

We now show how to use the proof system for GoodConn to obtain a proof system for 2TAUT.

Lemma 4.7. *If* GoodConn *has an NC^0 proof system, then so does* 2TAUT.

Proof. We will construct proof system $\mathcal{P}$ that generates all 2CNF formulas that are unsatisfiable and hence for 2TAUT. The idea is to take two graphs: G_1 with a path from some $i \in [n]$ to $\bar{i}$, and G_2 with a path from $\bar{j}$ to j for some $j \in [n]$, and combine them together to get a graph with path $i \rightsquigarrow \bar{i} \rightsquigarrow i$. When the vertices from 1 to n are interpreted as positive literals and vertices from $n+1$ to $2n+1$ are interpreted as negated literals, we get an implication graph of a formula that is not satisfiable. We will describe the construction of a proof circuit $P \in \mathcal{P}$ that generates every such formula on n variables. As mentioned before, the encoding we use has $4n^2$ bits and represents the adjacency matrix of the implication graph. Equivalently, each bit represents a clause in the formula.

Construction: P takes the following as input:

- Adjacency matrix A_1 of a graph $G_1 \in$ GoodConn on $2n+2$ vertices.
- Adjacency matrix A_2 of a graph $G_2 \in \text{GoodConn}^R$ on $2n+2$ vertices.
- $(2n+2) \times (2n+2)$ Adjacency matrix B.

The guarantee that the first two input adjacency matrices come from GoodConn and GoodConn^R respectively is achieved by using the outputs of the appropriate proof circuit $P_1 \in \mathcal{GC}$ and $P_2 \in \mathcal{GC}^R$ as constructed before.

For convenience, we will make P compute a graph on $2n+2$ vertices. For the final output graph, we do not output the vertices s and t or any edges that involve s or t.

P outputs a graph that has all edges in G_1, all edges in G_2, all edges indicated by B and some additional edges determined by the following "Stitch" rule:

- Stitch rule: If $(\bar{i}, t) \in G_1$ and $(t, \bar{j}) \in G_2$, then add edges $(\bar{i}, \bar{j})$ and (j, i).

Soundness: We need to show that any graph G output by P is an implication graph for a formula in 2UnSat. It suffices to show that there exists an i such that there is a path $i \rightsquigarrow \bar{i}$ and $\bar{i} \rightsquigarrow i$. Putting together these two paths results in a walk starting at i and ending at i via $\bar{i}$. For convenience, we will refer to this as a path (although strictly a walk) and write $i \rightsquigarrow \bar{i} \rightsquigarrow i$.

Claim 4.8. *For any graph G output by a circuit $P \in \mathcal{P}$, $\exists i \in [n]$ such that there is a path $i \rightsquigarrow \bar{i} \rightsquigarrow i$.*

Proof. Let $G = P(G_1, G_2, B)$. Since $G_1 \in$ GoodConn, there exists an $i \in [n]$ such that a path ρ_1: $s \to i \rightsquigarrow \bar{i} \to t$ is in G_1. Similarly a path ρ_2: $t \to \bar{j} \rightsquigarrow j \to s$ exists in G_2. We now have two cases:

1. If $i = j$, then since G contains all edges in G_1 and G_2, G has a path ρ: $i \rightsquigarrow \bar{i} \rightsquigarrow i$.
2. If $i \neq j$, then the Stitch rule forces the edges $(\bar{i}, \bar{j})$ and (j, i). Together with ρ_1 and ρ_2, we get a path ρ:$i \overset{\rho_1}{\rightsquigarrow} \bar{i} \to \bar{j} \overset{\rho_2}{\rightsquigarrow} j \to i$. An example of this case is shown in Figure 4.3.

□

Completeness: Take any formula $F \in$ 2UnSat on n variables. Let the implication graph of F be G. We will show that G is produced by P. We know that in G, $\exists i \in [n]$ such that there is a simple path ρ: $i \rightsquigarrow \bar{i} \rightsquigarrow i$. We can think of ρ as two parts: $i \rightsquigarrow \bar{i}$ and $\bar{i} \rightsquigarrow i$. Let the first part be ρ_1 and the second part be ρ_2. Define graph G_1 on $2n$ vertices to contain the path ρ_1 and no other edges. Similarly let G_2 be a graph on $2n$ vertices with the path ρ_2 and no other edges. Construct graph G_1' on $2n+2$ vertices as follows: G_1' is exactly

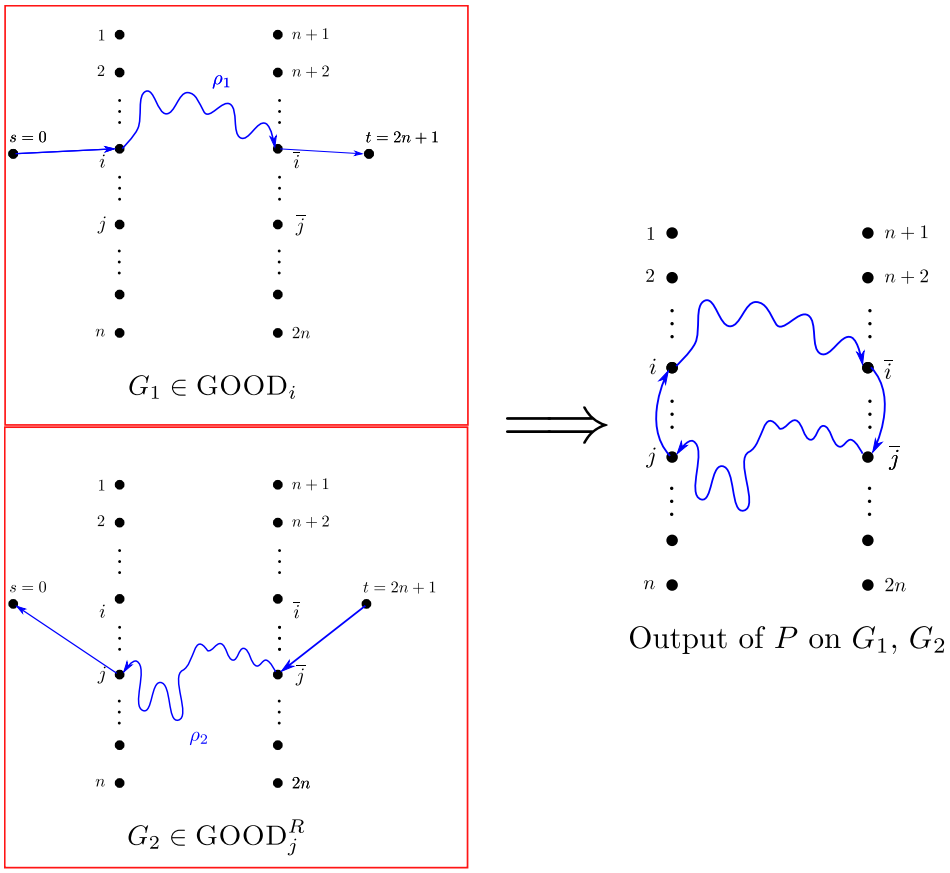


Figure 4.3: Effect of Stitch Rule

G_1 with two additional vertices $s = 0$ and $t = 2n + 2$ and edges (s, i) and $(\bar{i}, t)$. Clearly $G_1' \in$ GoodConn. Similarly construct G_2' from G_2 such that $G_2' \in \text{GoodConn}^R$. Adjacency matrix A_1 of G_1' is produced as output by proof system P_1 on some input and adjacency matrix A_2 of G_2' is produced as output by proof system P_2 on some input. Let B be the adjacency matrix of G. It is easy to see that $P(A_1, A_2, B) = G$.

$\mathcal{P}$ is computable in NC^0:

Lemma 4.9. *If $\mathcal{GC}$ is computable in NC^0, then $\mathcal{P}$ is computable in NC^0.*

Proof. We need to show that every output bit of a proof circuit $P \in \mathcal{P}$ is a function of at most $O(1)$ many input bits. P uses outputs A_1 and A_2 of proof circuits $P_1 \in \mathcal{GC}$ and $P_2 \in \mathcal{GC}^R$. Let output adjacency matrix of P be A. P incorporates only one rule and this can be expressed formally as follows:

- For all $i, j \in [n]$, $A[\bar{i}, \bar{j}] = (A_1[\bar{i}, t] \wedge A_2[t, \bar{j}]) \vee A_1[\bar{i}, \bar{j}] \vee A_2[\bar{i}, \bar{j}] \vee B[\bar{i}, \bar{j}]$

- For all $i, j \in [n]$, $A[j,i] = (A_1[\bar{i}, t] \wedge A_2[t, \bar{j}]) \vee A_1[j,i] \vee A_2[j,i] \vee B[j,i]$

For all edges $e = (i, j)$ that do not look like $(\bar{i}, \bar{j})$ or (j, i), $A[i, j] = A_1[i, j] \vee A_2[i, j] \vee B[i, j]$. Hence each output bit is a function of at most two bits from A_1, two bits from A_2 and one bit from B.

Combining Observation 4.6 and the fact that A_1 and A_2 are outputs of proof circuits $P_1 \in \mathcal{GC}$ and $P_2 \in \mathcal{GC}^R$, we have the lemma. □

This completes proof of Lemma 4.7. □

Combining Lemma 4.2 and Lemma 4.7 completes the proof of Theorem 4.1. □

Intuitively, reachability and connectedness are global properties. Hence intuition suggests that STConn should not have NC^0 proof systems. However, we have not been able to show this. In the following section, we study USTConn - the undirected analogue of STConn and show that USTConn has a proof system computable in NC^0.

4.2 Undirected Reachability

In this section, we show that the set USTConn of all undirected graphs with a path between two fixed vertices s and t has a proof system computable in NC^0.

Correcting an input that has no s-t path can be done locally by just adding the edge (s, t) to the output. However, detecting that the input does not have an s-t path with only local checks seems difficult. Always adding the (s, t) edge without checking for the absence of an $s - t$ path does not give us completeness. For this reason, we interpret the input differently.

The idea is as follows: We will first construct an NC^0 proof system C for the language Cycles of all undirected graphs that are a union of edge disjoint cycles. Consider a circuit $C \in C$ that outputs graphs on n vertices. The proof system we construct for USTConn takes the adjacency matrix of the graph G output by C and does an EXOR with the edge (s, t) to obtain a graph G' (i.e., if the edge (s, t) was present, we remove it and if the edge (s, t) was not present, we add it). Note that if G contained a cycle with the (s, t) edge, then removing this edge leaves an s-t path in the resultant graph. If G did not contain an (s, t) edge, then the EXOR with (s, t) results in a graph with an s-t path of length 1. To

get completeness, we take the upward closure of the graph G' by computing a bitwise OR with another input adjacency matrix (just like Lemma 3.14).

We first demonstrate this idea on the grid graph analogue of USTConn. In the grid graph case, generating the language of all grid graphs that look like edge disjoint cycles becomes very easy and hence helps in understanding the main idea better.

Define the following languages:

$$\textsc{GridCycles} = \left\{ A \in \{0,1\}^{2n(n-1)} \middle| \begin{array}{l} A \text{ is the adjacency matrix of an undirected grid graph} \\ G \text{ on } n^2 \text{ vertices that is a union of edge disjoint cycles.} \end{array} \right\}$$

$$\text{GRID} = \left\{ A \in \{0,1\}^{2n(n-1)} \middle| \begin{array}{l} A \text{ is the adjacency matrix of an undirected grid graph} \\ G \text{ on } n^2 \text{ vertices that has a path from } (1,1) \text{ to } (n,n). \end{array} \right\}$$

We think of a grid graph with n^2 vertices as having n rows and n columns with a vertex at the intersection of a row and column. Edges are of exactly two types: (1) Horizontal edges - edges between two vertices that are adjacent and in the same row and (2) Vertical edges - edges between two vertices are adjacent and in the same column.

We call the empty area created by the intersection of two consecutive rows and two consecutive columns as a "cell". In other words, a cell is the area enclosed by the 4-cycles of the grid graph. A grid graph on n^2 vertices will have exactly $(n-1)^2$ cells.

We first construct an NC^0 proof system for GridCycles.

Theorem 4.10. *The language* GridCycles *has an* NC^0 *proof system.*

Proof. We will construct a proof circuit that outputs every $G \in$ GridCycles on n^2 vertices.

The idea is that the set of all 4-cycles in a grid graph forms a generating set for GridCycles over addition modulo 2. For any cycle C, the 4-cycles that generate C are given by the cells that are enclosed by C.

Construction: Our proof system takes as input one bit for each cell. We will hardwire 0 for cells beyond the boundary of the grid graph. This way we ensure that all edges have two values adjacent to them.

We output an edge e if the two bits adjacent to e are different.

Soundness: Let k be the number of cells assigned 1 by the input. We proceed by induction on k. Base case: $k = 0$. Output graph will have no edges and hence is a union of edge disjoint cycles trivially. Induction step: $k > 0$. Take any input a with k 1s. Let G be the output of the proof system on input a. Pick any cell c that is set to 1 by the input. Let a' be the string obtained from a with 0 assigned to c and remaining values unchanged. Let G' be the graph output on a'. From induction hypothesis $G' \in$ GridCycles. Now we look at how G' changes when we switch on c. If all the four cells adjacent to c are assigned 0, then clearly assigning 1 to c will just add another edge disjoint cycle (the cycle around c) to G' to give G and hence $G \in$ GridCycles. Else if some of the cells adjacent to c are 1, then by construction, when c is assigned 1, only those edges of the 4-cycle around c are output that are not present in any other cycle. The key observation here is that when two cycles intersect at an edge, removing the edge results in a bigger cycle. Hence we always obtain edge disjoint cycles. Hence $G \in$ GridCycles in this case too.

Completeness: Take any graph $G \in$ GridCycles. Assign 1 to all cells that are enclosed within a cycle. It is easy to see that our proof system will produce G on this assignment.

Depth: Each output bit looks at exactly two input bits. Hence the construction is in NC^0.

□

Now we use the proof system for GridCycles to construct a proof system for GRID.

Theorem 4.11. GRID *has an* NC^0 *proof system.*

Proof. The idea is to add (modulo 2) an s-t path with graphs from GridCycles. Throughout this proof we will consider grid graphs on n^2 vertices. For any grid graph H and a path ρ in a grid graph, we define the grid graph $G' = H \oplus \rho$ as follows: Let $G' = (V', E')$ such that

$$e \in E' \iff (e \in H \wedge e \notin \rho) \vee (e \notin H \wedge e \in \rho)$$

Construction: Our proof system takes as input a graph $H \in$ GridCycles. We output the adjacency matrix of the graph $G' = H \oplus \rho$ where ρ is some fixed simple path from s to t. For convenience, let us fix ρ to be the path that starts at s and proceeds horizontally along the first row till $(1, n)$ and then proceeds vertically along the last column to reach t). Let $GRID'$ be the language generated by this proof system. We will show that GRID =

UpClose(GRID′) and hence obtain a proof system for GRID using Lemma 3.14. An example input and output are shown in Figure 4.4. In the figure, the output edges are indicated by the thick blue edges, while the dotted line segments merely indicate the grid. The input bits are shown in their respective cells. We observe that our construction so far

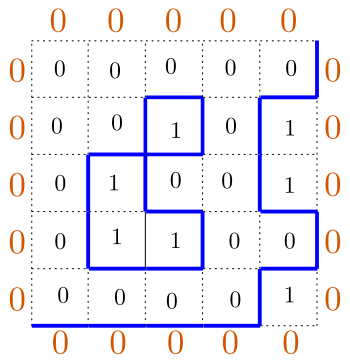


Figure 4.4: Example input and output

is sound:

Observation 4.12. *Let H be any graph from* GridCycles *and σ any simple path from s to t. Then, $H \oplus \sigma$ has a simple path between s and t.*

A formal proof for this observation proceeds by induction on number of cycles in H and uses the fact that in any graph, if vertices s and t have degree one and all other vertices have even degree, then the graph consists of zero or more cycles and an $s - t$ path, all pairwise disjoint. We do not prove this observation formally here since we show soundness for the general graph case later.

So the $\oplus$ operation does not disconnect s and t. Hence all graphs in GRID′ have an s-t path. Also, every graph G that has a simple s-t path σ and no other edges can be written as $H \oplus \rho$ for some $H \in$ GridCycles. To see this, consider a graph $H = G \oplus \rho$. It is easy to see that H is indeed in GridCycles.

However, GRID′ does not have any graph that has "loose edges" - edges that are not part of any cycle and not part of an s-t path. Hence, GRID can be seen as UpClose(GRID′). Now we just use Lemma 3.14 to obtain a proof system or GRID. □

Now we generalize Theorem 4.11 to the language Undirected Reachability, which is known to be in (and complete for) L ([Rei08]). Our proof system will output adjacency matrices of all graphs that have a path between s and t, and of no other graphs. In the process, we generalize Theorem 4.10 to give an NC^0 proof system for the set of all undirected graphs that are a union of edge-disjoint cycles.

Define the following languages:

$$\text{USTConn} = \left\{ A \in \{0,1\}^{n \times n} \middle| \begin{array}{c} A \text{ is the adjacency matrix of an undirected graph } G \\ \text{where vertices } s = 1, t = n \text{ are in the same connected} \\ \text{component.} \end{array} \right\}$$

$$\text{Cycles} = \left\{ A \in \{0,1\}^{n \times n} \middle| \begin{array}{c} A \text{ is the adjacency matrix of an undirected graph } G = \\ (V, E) \text{ where } E \text{ is the union of edge-disjoint simple} \\ \text{cycles.} \end{array} \right\}$$

(For simplicity, we will say $G \in$ USTConn or $G \in$ Cycles instead of referring to the adjacency matrices.)

Theorem 4.13. *The language* USTConn *has an* NC^0 *proof system.*

Proof. We will need an addition operation on graphs: $G_1 \oplus G_2$ denotes the graph obtained by adding the corresponding adjacency matrices modulo 2. If A is a collection of graphs and $B = \text{UpClose}(A)$, then B is the collection of super-graphs obtained by adding edges. Note that (undirected) reachability is monotone and hence UpClose(USTConn) = USTConn.

Let $L_1 = \{G = G_1 \oplus (s,t) | G_1 \in \text{Cycles}\}$ and $L_2 = \text{UpClose}(L_1)$. We show:

1. $L_2 = \text{USTConn}$.
2. If L_1 has an NC^0 proof system, then L_2 has an NC^0 proof system.
3. If Cycles has an NC^0 proof system, then L_1 has an NC^0 proof system.
4. Cycles has an NC^0 proof system.

Proof of 1: We show that $L_1 \subseteq \text{USTConn} \subseteq L_2$. Then applying upward closure, $L_2 = \text{UpClose}(L_1) \subseteq \text{UpClose}(\text{USTConn}) = \text{USTConn} \subseteq \text{UpClose}(L_2) = L_2$.

$L_1 \subseteq$ USTConn: Any graph $G \in L_1$ looks like $G = H \oplus (s,t)$, where $H \in$ Cycles. If $(s,t) \notin H$, then $(s,t) \in G$ and we are done. If $(s,t) \in H$, then s and t lie on a cycle C and hence removing the (s,t) edge will still leave s and t connected by a path $C \setminus \{(s,t)\}$.

USTConn $\subseteq L_2$: Let $G \in$ USTConn. Let ρ be an s-t path in G. Let $H = (V,E)$ be a graph such that $E =$ edges in ρ. Then, $G \in \text{UpClose}(\{H\})$. We can write H as $H' \oplus (s,t)$ where $H' = H \oplus (s,t)$. If $\rho = (s,t)$, then $E(H')$ is empty and hence $H' \in$ Cycles. Else $\rho \neq (s,t)$, and then $H' = H \oplus (s,t) = \rho \cup (s,t)$ since ρ is a simple path, and hence $H' \in$ Cycles. Either way, $H' \in CYCLES$ and so $H \in L_1$. Hence $G \in L_2$.

Proof of 2: Note that Minterms(USTConn) is exactly the set of graphs where the edge set is a simple s-t path. We have seen that $L_1 \subseteq$ USTConn. As above, we can see that $H \in$ Minterms(USTConn) $\implies H \oplus (s,t) \in$ Cycles $\implies H \in L_1$. Statement 2 now follows from Lemma 3.14.

Proof of 3: Let A be the adjacency matrix output by the NC^0 proof system for Cycles. The proof system for L_1 outputs A' such that $A'[s,t] = \overline{A[s,t]}$ and rest of A' is same as A.

Proof of 4: This is of independent interest, and is proved in Theorem 4.14 below.

This completes the proof of theorem 4.13. □

We now construct NC^0 proof systems for the language Cycles.

Theorem 4.14. *The language* Cycles *has an* NC^0 *proof system.*

Proof. To design an NC^0 proof system for Cycles, we try to follow the technique we used for GridCycles. While generating graphs in GridCycles, we observed that the 4-cycles of a grid graph form a generating set that is also local when used for generating edges. We want to identify such a generating set for Cycles.

Let T be a family of graphs. We say that an edge e is *generated* by a sub-family $\mathcal{S} \subseteq T$ if the number of graphs in $\mathcal{S}$ which contain e is odd. We say that the family T generates a graph G if there is some sub-family $\mathcal{S} \subseteq T$ such that every edge in G is generated by $\mathcal{S}$, and no other edge is generated. We first observe that to generate every graph in the set Cycles, we can set T to be the set of *all* triangles. Given any cycle, it is easy to come up with a set of triangles that generates the cycle; namely, take any triangulation of the cycle. Therefore, if we let T be the set of all triangles on n vertices, it will generate every graph in Cycles. Also, no other graph will be generated because any set $\mathcal{S} \subseteq$ Cycles generates a set contained in Cycles (see Lemma 4.16 below). This immediately gives a proof system for Cycles: given a vector $x \in \binom{n}{3}$, we will interpret it as a subset $\mathcal{S}$ of triangles. We will output an edge e if it is a part of odd number of triangles in $\mathcal{S}$. Finally, because of the properties observed above, any graph generated in this way will be a graph from the set Cycles.

Unfortunately, this is not an NC^0 proof system because to decide if an edge is generated, we need to look at $\Omega(n)$ triangles. For designing an NC^0 proof system we need to come up with a set of triangles such that for any graph $G \in$ Cycles, every edge in G is a part of $O(1)$ triangles.

So on the one hand, we want the set of triangles to generate every graph in Cycles, and

on the other hand we need that for any graph $G \in$ Cycles, every edge in G is a part of $O(1)$ triangles. We show that such a set of triangles indeed exists.

Thus our task now is to find a set of triangles $T \subseteq$ Cycles such that:

1. Every graph in Cycles can be generated using triangles from T. i.e.,

$$\text{Cycles} \subseteq \text{Span}(T) \triangleq \left\{ \sum_{i=1}^{|T|} a_i t_i \mid \forall i, a_i \in \{0,1\}, t_i \in T \right\}$$

2. Every graph generated from triangles in T is in Cycles; $\text{Span}(T) \subseteq$ Cycles.

3. $\forall u, v \in [n]$, the edge (u, v) is contained in at most 6 triangles in T.

Once we find such a set T, then our proof system asks as input the coefficients a_i which indicate the linear combination needed to generate a graph in Cycles. An edge e is present in the output if, among the triangles that contain e, an odd number of them have coefficient set to 1 in the input. By property 3, each output edge needs to see only $O(1)$ input bits and hence the circuit we build is NC^0. We will now find and describe T in detail.

Let the vertices of the graph be numbered from 1 to n. Define the length of an edge (i, j) as $|i - j|$. A triple $\langle i, j, k \rangle$ denotes the set of all graphs on n vertices that have exactly one triangle on vertices (u, v, w) where $|u - v| = i$, $|v - w| = j$, and $|u - w| = k$ and no other edges. So each graph in $\langle i, j, k \rangle$ is on n vertices but has exactly three edges that form a triangle with lengths i, j and k. We now define the set

$$T = \bigcup_{i=1}^{n/2} \langle i, i, 2i \rangle \cup \langle i, i+1, 2i+1 \rangle$$

Although the graphs in T have all n vertices, we sometimes refer to them as triangles when it is convenient.

Observation 4.15. *It can be seen that $|T| \leq \frac{3}{2}n^2$. This is linear in the length of the output, which has $\binom{n}{2}$ independent bits.*

We now show that T satisfies all properties listed earlier in reverse order.

T satisfies property 3: Take any edge $e = (u, v)$. Let its length be $l = |u - v|$. e can either be the longest edge in a triangle or one of the two shorter ones. If l is even, then e can be the longest edge for only 1 triangle in T and can be a shorter edge in at most 4 triangles in T. If l is odd, then e can be the longest edge for at most 2 triangles in T and can be a shorter edge in at most 4 triangles. Hence, any edge is contained in at most 6 triangles.

T **satisfies property 2**: To see this, note first that $T \subseteq$ Cycles. Next, observe the following closure property of cycles:

Lemma 4.16. *For any $G_1, G_2 \in$ Cycles, the graph $G_1 \oplus G_2 \in$ Cycles.*

Proof. A well-known fact about connected graphs is that they are Eulerian if and only if every vertex has even degree. The analogue for general (not necessarily connected) graphs is Veblen's theorem [Veb12], which states that $G \in$ Cycles if and only if every vertex in G has even degree.

Using this, we see that if for $i \in [2]$, $G_i \in$ Cycles and if we add the adjacency matrices modulo 2, then degrees of vertices remain even and so the resulting graph is also in Cycles. □

It follows that Span$(T) \subseteq$ Cycles.

T **satisfies property 1**: We will show that any graph $G \in$ Cycles can be written as a linear combination of graphs in T. Define, for a graph G, the parameter $d(G) = (l, m)$ where l is the length of the longest edge in G and m is the number of edges in G that have length l. For graphs $G_1, G_2 \in$ Cycles, with $d(G_1) = (l_1, m_1)$ and $d(G_2) = (l_2, m_2)$, we say $d(G_1) < d(G_2)$ if and only if either $l_1 < l_2$ holds or $l_1 = l_2$ and $m_1 < m_2$. Note that for any graph $G \in$ Cycles with $d(G) = (l, m)$, $l \geq 2$ (unless G is the graph with no edges).

Claim 4.17. *Let $G \in$ Cycles. If $d(G) = (2, 1)$, then $G \in T$.*

Proof. It is easy to see that G has to be a graph that has a triangle with edge lengths 1, 1 and 2 and no other edges. All such graphs are contained in T by definition. □

Lemma 4.18. *For every $G \in$ Cycles with $d(G) = (l, m)$, either $G \in T$ or there is a $t \in T$, and $H \in$ Cycles such that $G = H \oplus t$ and $d(H) < d(G)$.*

Proof. If $G \in T$, then we are done. So now consider the case when $G \notin T$:

Let e be a longest edge in G. Let C be a cycle that contains e. Pick $t \in T$ such that e is the longest edge in t. G can be written as $H \oplus t$ where $H = G \oplus t$. From Lemma 4.16 and since $T \subseteq$ Cycles, we know that $H \in$ Cycles. Let t have the edges e, e_1, e_2. Any edge present in both G and t will not be present in H. Since $e \in G \cap t$, $e \notin H$. Lengths of e_1 and e_2 are both less than l since e was the longest edge in t. Hence the number of times an edge of length l appears in H is reduced by 1 and the new edges added (if any) to H (namely e_1 and e_2) have length less then l. Hence if $m > 1$, then $d(H) = (l, m-1) < d(G)$. If $m = 1$, then $d(H) = (l', m')$ for some m' and $l' < l$, and hence $d(H) < d(G)$. □

By repeatedly applying Lemma 4.18, we can obtain the exact combination of triangles from T that can be used to give any $G \in$ Cycles.

Figure 4.5 illustrates such a repeated application of Lemma 4.18 on an example graph G shown at the top. The numbers in black next to vertices indicate vertex numbers and the numbers on the edges indicate the edge lengths. The final decomposition S has all graphs belonging to T. In Figure 4.6, the graphs in S are superimposed on each other. Any edge that appears twice in the graph formed by the superimposition does not exist in G since G is a sum modulo 2 of the graphs from S.

A more formal proof will proceed by induction on the parameter $d(G)$ and each application of Lemma 4.18 gives a graph H with a $d(H) < d(G)$ and hence allows for the induction hypothesis to be applied. The base case of the induction is given by Lemma 4.17. Hence T satisfies property 1.

Since T satisfies all three properties, we obtain an NC^0 proof system for Cycles, proving the theorem.

□

4.3 Pushing the Bounds

We know that any language in NP has AC^0 proof systems. In the bounded fanin model, this corresponds to $O(\log n)$ depth. A natural question to ask is if depth $\Omega(\log n)$ is necessary. The class obtained by restricting AC^0 by not allowing $\wedge$ gates ($\vee$ gates) to have unbounded fanin and forcing all negations to be applied to leaves is called SAC^0 (coSAC^0). The class SAC^i was defined in [BCD+89]. It has been known that SAC^0 is not closed under complement (see [Ven91]) and hence $\mathsf{SAC}^0 \subsetneq \mathsf{AC}^0$.

The following theorem implies that there is a language in NP for which any proof system generating it requires power at least that of SAC^0 or coSAC^0.

Theorem 4.19 (Srikanth Srinivasan (private communication))**.** *There is a language A in* NP *such that any bounded-fanin proof system for A needs $\Omega(\log n)$ depth.*

Proof. Let $A \subseteq \{0,1\}^*$ be an error correcting code of constant rate (For each n, A has $2^{\Omega(n)}$ strings of length n) and linear distance (the Hamming distance between two words of length n is $\Omega(n)$) that can be efficiently computed. Such codes are known to exist. See for example [Jus72]. Suppose there is a proof system $C_n : \{0,1\}^m \longrightarrow \{0,1\}^n$ of depth d that outputs exactly the strings in A. Assume that C is non-degenerate. i.e., for every

input position i, $\exists x \in \{0,1\}^m$ such that $C(x) \neq C(x \oplus e_i)$. Note that $m \in \Omega(n)$ since A has constant rate ($|A \cap \{0,1\}^n| = 2^{\Omega(n)}$). Note that since each output bit is a function of at most 2^d input bits, it must be the case that there exists an input position i such that x_i is connected to at most $O(2^d)$ output positions. For this i, let x be an input such that $C(x) \neq C(x \oplus e_i)$. But since $C(x)$ and $C(x \oplus e_i)$ are both codewords in A, they must differ in at least $\Omega(n)$ positions since A is has linear distance. This implies that x_i is connected to at least $\Omega(n)$ output positions and this is true for all i. Hence $d = \Omega(\log n)$. □

However, we note that proof systems for a big fragment of NP do not require the full power of AC^0. In particular, for every language in NP, an extremely simple padding yields another language with simpler proof systems.

Theorem 4.20. *Let L be any language in* NP.

1. *If L contains 0^*, then L has a proof system where negations appear only at leaf level, $\wedge$ gates have unbounded fanin, $\vee$ gates have $O(1)$ fanin, and the depth is $O(1)$. That is, L has a* coSAC^0 *proof system.*

2. *If L contains 1^*, then L has a proof system where negations appear only at leaf level, $\vee$ gates have unbounded fanin, $\wedge$ gates have $O(1)$ fanin, and the depth is $O(1)$. That is, L has an* SAC^0 *proof system.*

3. *The language $(\{1\} \cdot L \cdot \{0\}) \cup 0^* \cup 1^*$ has both* SAC^0 *and* coSAC^0 *proof systems.*

Proof. Let L be a language in NP. Then there is a family of uniform polynomial-sized circuits (C_n), where each C_n has $q(n)$ gates, n standard inputs x and $p(n)$ auxiliary inputs y, such that for each $x \in \{0,1\}^n$, $x \in L \iff \exists y : C_n(x,y) = 1$. We use this circuit to construct the proof system. The input to the proof system consists of words $x = x_1 \dots x_n$, $y = y_1 \dots y_{p(n)}$, $z = z_1 \dots z_{q(n)}$. The intention is that y represents the witness such that $C_n(x,y) = 1$, and z represents the vector of values computed at each gate of C_n on input x, y. There are two ways of doing self-correction with this information:

- Check for consistency: Check that every gate $g_i = g_j \circ g_k$ satisfies $z_i = z_j \circ z_k$. Output the string w where $\langle w \rangle = \langle x \rangle \wedge (\bigwedge_{i=1}^{q(n)} [z_i = z_j \circ z_k])$. If even one gate is inconsistent, w equals 0^*, otherwise w is the input x that has been certified by y, z; hence w is in $L \cup 0^*$. Every string in L can be produced by giving witness y and consistent z. The expression shows that this is a coSAC^0 circuit.

- Look for an inconsistency: Find a gate $g_i = g_j \circ g_k$ where $z_i \neq z_j \circ z_k$. Output the string w where $\langle w \rangle = \langle x \rangle \vee (\bigvee_{i=1}^{q(n)} [z_i \neq z_j \circ z_k])$. If even one gate is inconsistent, w

equals 1^*, otherwise w equals the input x that has been certified by y, z; hence w is in $L \cup 1^*$. Every string in L can be produced by giving suitable y, z. The expression shows that this is an SAC^0 circuit.

□

4.4 Discussion

Although we expected USTConn to not admit NC^0 proof systems, a different way of looking at paths in terms of cycles enabled us to detect if the input was faulty. The generating set used in the proof system for USTConn is of independent interest and potentially has applications in other areas.

In the case of directed graphs and STConn, it seems that a natural thing to try would be to use a directed version of the generating set T used in proof of Theorem 4.13. The problem with one such approach is that certain cycles might need multiple copies of the same triangle. Such a need for multiplicities essentially means that the input cannot be just a bit string anymore, but will have to be a string of natural numbers. Bounding the multiplicity has proven to be quite difficult so far.

Are there NC^0 proof system for STConn or 2TAUT? The results of Chapter 2 and 3 make it clear that it is not easy to guess one way or the other. We observe that the minterms of 2TAUT cannot have an NC^0 proof system:

Proposition 4.21. *The language M consisting of the minterms of* 2TAUT *(under the $4n^2$ bits encoding described earlier) does not have an NC^0 proof system.*

Proof. Note that in any minterm $m \in M$, if the terms $C_i = (x_i \wedge x_i)$ and $C'_i = (\overline{x}_i \wedge \overline{x}_i)$ are present, then no other terms can be present. This is because $C_i \vee C'_i$ is already a tautology and hence if any other term was present, then the resulting formula would not be a minterm.

We construct proof system for Exact-Or $\cup\, 0^*$ using a proof system for $\mathcal{P}$ of M. Let $P \in \mathcal{P}$ output a formula $F \in M$ on n variables. We construct output string y of length n as follows: $y_i = 1$ if and only if both C_i and C'_i are present in F. It is easy to see that we only output strings that have at most one 1. This is exactly the language $\neg\mathrm{Th}^n_2 =$ Exact-Or $\cup 0^*$. Using the lower bound established in Chapter 3 Section 3.1, we conclude that M cannot have an NC^0 proof system. □

This observation about minterms of 2TAUT does not help in guessing if 2TAUT has an NC^0 proof system. This is because we know that the language L_{OR} has an NC^0 proof system while the language Exact-Or consisting of its minterms does not.

The consequences of Taut having an NC^0 proof system suggest that showing that Taut does not have proof systems computable by such restricted models as NC^0 should be easy. But we are yet to show such a result. In fact, even showing that Taut does not have NC^0 proof systems which are further restricted to force each input bit to only influence $O(1)$ many output bits is still open.

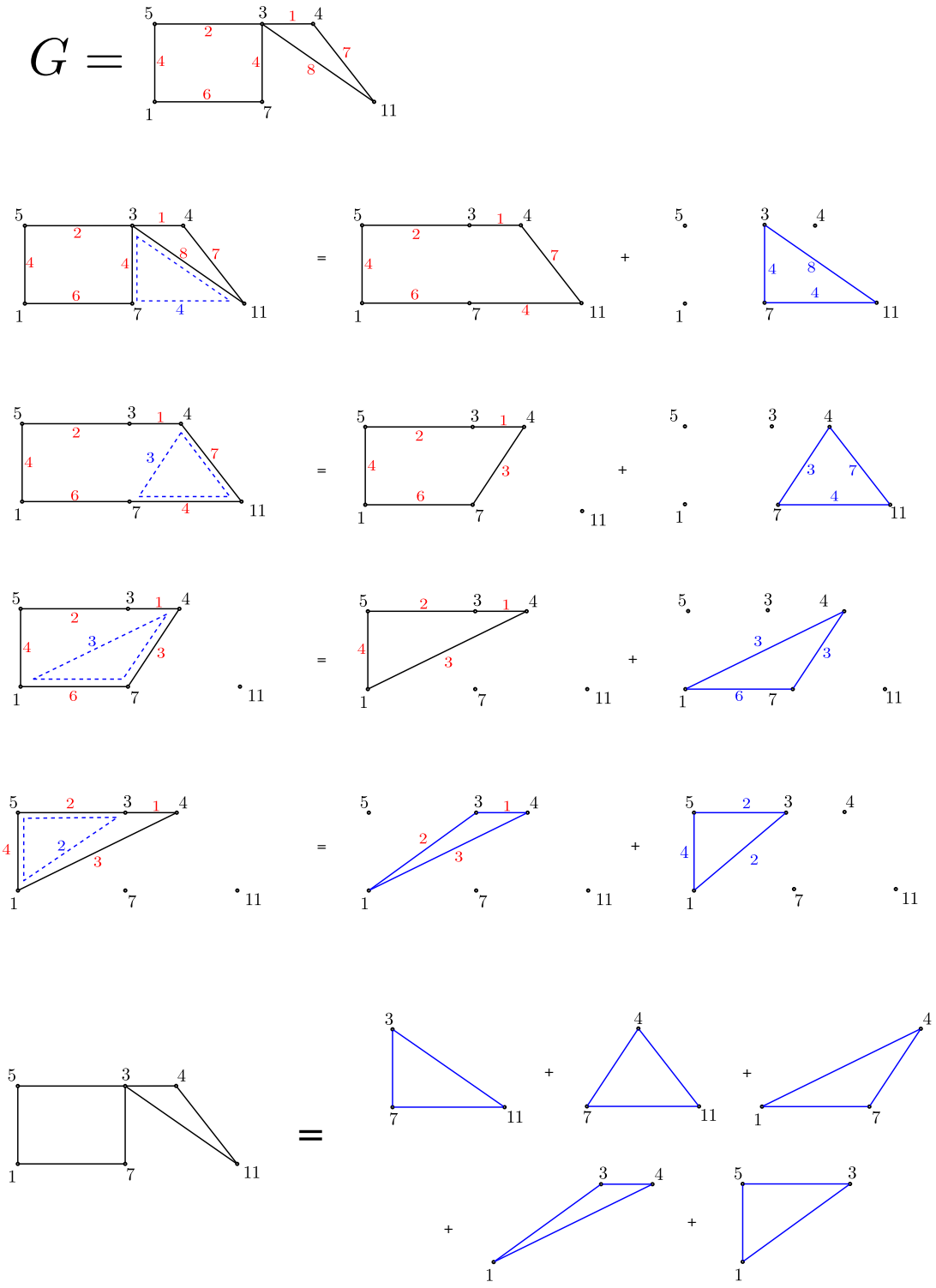


Figure 4.5: Decomposition of G into graphs from T

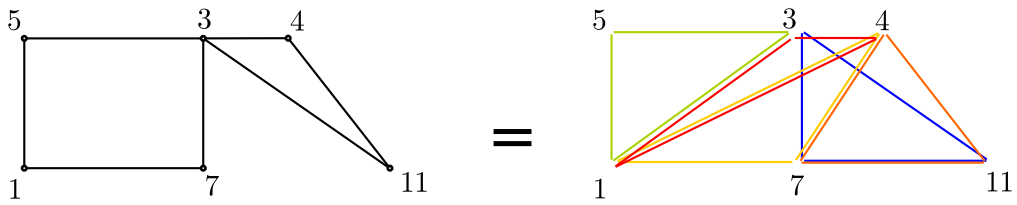


Figure 4.6: Decomposition of G into graphs from T (superimposed)

Chapter 5

Arithmetic Circuits and PIT

In this chapter we study the problem PIT of checking if an input arithmetic circuit is identically zero. We give deterministic polynomial time algorithms for PIT for certain restricted classes of polynomials. We start with defining the basic terms and notation.

5.1 Preliminaries

Circuits, formulas, polynomials.

Let $X = \{x_1, \ldots, x_n\}$ be a set of variables. An *arithmetic circuit* C over a ring R is a directed acyclic graph with internal nodes labeled + or $\times$ and leaves labeled from $X \cup R$. Every node has in-degree zero or two, and there is exactly one node of out-degree zero, called the output gate. Unless otherwise stated, we consider R to be the ring of integers $\mathbb{Z}$, and we allow only the constants $\{-1, 0, 1\}$ in the circuits. An *arithmetic formula* F is an arithmetic circuit where fan-out for every gate is at most one.

The depth of a circuit is the length of a longest root-to-leaf path. The alternation depth of the formula is the maximum number of maximal blocks of + and $\times$ gates on any root-to-leaf path in the formula. In the literature on identity testing, depth is used to mean alternation-depth. However we differentiate between these, as is done in uniform circuit complexity literature, because bounded fanin is crucial to some of our algorithms. Note that converting a circuit to a bounded fanin circuit increases only the depth, not the size or the alternation depth.

Every node in C computes a polynomial in $R[x_1, \ldots, x_n]$ in a natural way. Let g be a gate in a circuit (or formula) C. We denote by p_g the polynomial computed at gate g of C.

We denote by p_C the polynomial p_r, where r is the output gate of C. For a gate g, let $\mathsf{subvar}_g \stackrel{\Delta}{=} \{x_i \mid x_i \text{ is a leaf in the subtree rooted under } g\}$.

A *read-once* arithmetic formula (ROF for short) is an arithmetic formula where each variable occurs at most once as a label. More generally, in a *read-k* arithmetic formula a variable occurs at most k times as a label.

We say that an ROF is constant-free (denoted CF-ROF) if the labels at the leaves are of the form ax for $x \in X$ and $a \in \mathbb{F} \setminus \{0\}$. We call polynomials computed by such formulas constant-free ROPs, denoted CF-ROP.

For a polynomial $f \in \mathbb{F}[x_1, x_2, \cdots, x_n]$, a set $S \subseteq [n]$ and an assignment a, let $f_{S \to a_S}$ denote the polynomial on variables $\{x_i : i \notin S\}$ obtained from f by setting $x_j = a_j$ for $j \in S$. Using notation from [SV08], for a polynomial f, $\mathsf{var}(f)$ denotes the set of variables that f depends on non-trivially. We say that f is 0-justified if for all $S \subseteq \mathsf{var}(f)$, $\mathsf{var}(f|_{S \to 0}) = \mathsf{var}(f) \setminus S$. So an ROF F is 0-justified if and only if for every $x_i \in \mathsf{var}(f)$, the coefficient of the monomial x_i is non-zero.

Complexity Classes.

For all the standard complexity classes, the reader is referred to [AB09]. We provide below the definitions of some of the less-familiar complexity classes that are used in this chapter. Let $f = (f_n)_{n \geq 0}$ be a family of integer valued functions $f_n : \{0,1\}^n \to \mathbb{Z}$. f is in the complexity class GapL exactly when there is some nondeterministic logspace machine M such that for every x, $f(x)$ equals the number of accepting paths of M on x minus the number of rejecting paths of M on x. $\mathsf{C_=L}$ is the class of languages L such that for some $f \in \mathsf{GapL}$, for every x, $x \in L \Leftrightarrow f(x) = 0$. The GapL hierarchy, at an intuitive level, can be seen as classes of functions built over bit access to other functions, with a deterministic logspace machine at the base and access to GapL functions at the first level. It is known to be contained in NC^2. Due to technical subtleties in the definition of the GapL hierarchy, the reader is referred to [AO94, ABO99] or [AV11].

GapNC^1 denotes the class of families of functions $(f_n)_{(n \geq 0)}$, $f_n : \{0,1\}^n \to \mathbb{Z}$, where f_n can be computed by a uniform polynomial size log depth arithmetic circuit. This equals the class of functions computed by uniform polynomial-sized arithmetic formulas ([BCGR92]). $\mathsf{C_=NC}^1$ is the class of languages L such that for some GapNC^1 function family $(f_n)_{n \geq 0}$, and for every x, $x \in L \iff f_{|x|}(x) = 0$. The GapNC^1 hierarchy comprises of languages accepted by polynomial-size constant depth unbounded fanin circuits (AC^0) with oracle access to bits of GapNC^1 functions. It follows from the results of [HAB02]

that the hierarchy is contained in DLOG.

Notation.

A monomial is represented by the sequence of degrees of the variables. For instance, over x_1, x_2, x_3, the monomial x_1^2 is represented as $(2,0,0)$, and the monomial x_1x_3 is represented as $(1,0,1)$. For a degree sequence $m = (d_1, \ldots, d_n)$ we denote the monomial $\prod_{i=1}^{n} x_i^{d_i}$ by X^m. For any set $S \subseteq [n]$, we denote by m_S the multilinear monomial $\prod_{i \in S} x_i$. For a monomial m and polynomial p, $\mathsf{coeff}(p, m)$ denotes the coefficient of m in p. [statement S] is a Boolean 0-1 valued predicate that takes value 1 exactly when the statement S is true.

We use the following short-forms for some computational problems used in this chapter.

PIT: Short for Polynomial Identity Test. Given an arithmetic circuit C over $\mathbb{Z}$, test if the polynomial p_C computed by C is identically zero or not.

MLIN: Short for 'Multilinear', the problem is to decide if a given arithmetic circuit C computes a polynomial that is multilinear or not.

ExistExtMon: Short for "Exists an Extension to a given Monomial". Given an arithmetic circuit C, and a monomial m, test if there is a monomial M with non-zero coefficient in p_C such that M extends m; that is, $m|M$.

Note that for a polynomial p, when the input monomial is a single variable x_i, ExistExtMon reduces to checking if the partial derivative of p with respect to x_i is identically zero.

The following propositions list some of the known results we use in this chapter.

Proposition 5.1 ((follows from [BCGR92, HAB02])). *Evaluating an arithmetic formula where the leaves are labelled* $\{-1, 0, 1\}$ *is in* DLOG *(even* GapNC^1*).*

Proposition 5.2 ([SV08]). *Given k ROFs in n variables, there is a deterministic (non black-box) algorithm that checks whether they sum to zero or not. The running time of the algorithm is $n^{O(k)}$.*

Proposition 5.3 ([AvMV11]). *Given as input a read-k arithmetic formula F on n variables of size s that has the property that the polynomial computed by every subformula is multilinear, there is a deterministic algorithm that checks if F is identically zero. The running time of the algorithm is $s^{O(1)}n^{k^{O(k)}}$ (for constant read, this running time is polynomial in n).*

The following result can be obtained by easy reductions to known log-space complete problems [CM87].

Proposition 5.4 ([CM87])**.** *The following problems are in* DLOG*:*

1) Given a formula F, a gate $g \in F$, and a variable x, check whether $x \in \mathsf{subvar}_g$.

2) Given a rooted tree T, and two nodes u, v, find lowest common ancestor (LCA) of u and v.

5.2 Multilinearity and identity tests

In this section we consider read-restricted formulas, and the problems of testing multilinearity MLIN and testing identically zero PIT on read-2 and read-3 formulas. One way to test multilinearity is to take second order partial derivatives with respect to each variable and then check for identity on each of the resulting polynomials as described in [FMM12b]. The problem with this approach, when applied to read restricted formulas, is that computing partial derivatives as given in [FMM12b] could increase the number of times a variable is read and hence require computing PIT on a higher read formula.

In the following we show how to compute MLIN and PIT for the read-2 case:

Read-2 Formulas

The individual degree of a variable in a polynomial p computed by read-twice formula F is bounded by two. Thus, multilinearity testing boils down to testing if the second order partial derivative of F with respect to x_i is zero for every variable x_i. Note that the second-order partial derivative of F with respect to x_i is a polynomial in $n-1$ variables; thus MLIN reduces to n instances of PIT on $n-1$ variables. Our approach is to use the inductive structure of a read-twice polynomial to test these partial derivatives for zero, using the knowledge of multilinearity of gates at the lower levels. As an aid in this computation, we also check, for each gate g and each variable x, whether x survives in p_g. We say that x survives in g if $\mathsf{ExistExtMon}(g, x) = 1$ (this is equivalent to saying $x \in \mathsf{var}(g)$).

Theorem 5.5. *For read-twice formulas, the problems* PIT*,* $\mathsf{ExistExtMon}(\phi, x)$*, and* MLIN *(where ϕ is the input formula and x is a single variable in it) are in P.*

Proof. Let ϕ be the given read-twice formula on variables $x_1, \ldots, x_n$, with S internal nodes. Without loss of generality, assume that ϕ is strictly alternating. That is, inputs to

a + gate are either leaves or are × gates, and inputs to a × gate are either leaves or are + gates.

We proceed by induction on the structure of the formula ϕ.

We iteratively compute, for each gate g in ϕ and each variable $x \in X$, the following set of 0-1 valued functions:

$$\begin{aligned} \mathsf{PIT}(g) = 1 &\quad\Leftrightarrow\quad p_g \equiv 0 \\ \mathsf{MLIN}(g) = 1 &\quad\Leftrightarrow\quad p_g \text{ is multilinear} \\ \mathsf{ExistExtMon}(g, x) = 1 &\quad\Leftrightarrow\quad x \text{ survives in } p_g \text{ (i.e., } p_g \text{ has a monomial } m \text{ that contains } x) \end{aligned}$$

The base case is when ϕ is a single variable or a constant. That is, ϕ consists of a single gate g that is labelled $L \in \{x_1, \ldots, x_n\} \cup \{0, +1, -1\}$. Then $\mathsf{PIT}(g) = 1$ if and only if $L = 0$, $\mathsf{MLIN}(g) = 1$ always, and $\mathsf{ExistExtMon}(g, x) = 1$ if and only if $L = x$.

Now assume that for every gate u below the root gate of ϕ, the above functions have been computed and stored as bits. Let f be the root gate of ϕ. We show how to compute these functions at f. The order in which we compute them depends on whether f is a × or a + gate.

First, consider $f = g \times h$. We compute the functions in the order given below.

1. $\mathsf{PIT}(f)$: f is identically zero if and only if at least one of g, h is. Thus $\mathsf{PIT}(f) = \mathsf{PIT}(g) \vee \mathsf{PIT}(h)$.

2. $\mathsf{MLIN}(f)$: If f is identically zero, then it is vacuously multilinear. Otherwise, for it to be multilinear, it must be the product of two (non-zero) multilinear polynomials in disjoint sets of variables. Thus

$$\mathsf{MLIN}(f) = \quad \mathsf{PIT}(f) \vee \left[\mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h) \wedge \left(\bigwedge_{x \in X} [\neg \mathsf{ExistExtMon}(g, x) \vee \neg \mathsf{ExistExtMon}(h, x)] \right) \right]$$

   Note that the $\mathsf{PIT}(f)$ term is necessary, since f can be multilinear even if, for instance, g is not, provided $h \equiv 0$.

3. $\mathsf{ExistExtMon}(f, x)$: x appears in p_f if and only if $p_f \not\equiv 0$ and x appears in at least

one of p_g, p_h. Thus

$$\mathsf{ExistExtMon}(f, x) = \neg\mathsf{PIT}(f) \wedge [\mathsf{ExistExtMon}(g, x) \vee \mathsf{ExistExtMon}(h, x)]$$

Next, consider $f = g + h$. We compute the functions in the order given below.

1. $\mathsf{MLIN}(f)$: Since f is read-twice, a non-multilinear monomial in g cannot get cancelled by a non-multilinear monomial in h; that would require at least 4 occurrences of some variable. Thus, f is multilinear only if both g and h are. The converse is trivially true. Thus $\mathsf{MLIN}(f) = \mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h)$.

2. $\mathsf{ExistExtMon}(f, x)$: This is the non-trivial part of this proof; we defer the description to a bit later.

3. $\mathsf{PIT}(f)$: Once we compute the functions above, this is straightforward:

$$\mathsf{PIT}(f) = [f(0) = 0] \wedge \bigwedge_{x \in X} \neg\mathsf{ExistExtMon}(f, x)$$

   Checking if $f(0) = 0$ is feasible; see Proposition 5.1.

We now complete the description for computing $\mathsf{ExistExtMon}(f, x)$ when $f = g + h$. If x survives in neither g nor h, then it does not survive in f. But if it survives in exactly one of g, h, it cannot get cancelled in the sum, so it survives in f. Thus

$$\begin{aligned} \mathsf{ExistExtMon}(g, x) = 0 \wedge \mathsf{ExistExtMon}(h, x) = 0 &\implies \mathsf{ExistExtMon}(f, x) = 0 \\ \mathsf{ExistExtMon}(g, x) \oplus \mathsf{ExistExtMon}(h, x) = 1 &\implies \mathsf{ExistExtMon}(f, x) = 1 \end{aligned}$$

So now assume that x survives in both g and h. We can write the polynomials computed at g, h as $p_g = \alpha x + \alpha'$ and $p_h = \beta x + \beta'$, where α', β' do not involve x; and we know that $\alpha \not\equiv 0, \beta \not\equiv 0$. We want to determine whether $\alpha + \beta \equiv 0$.

Since x appears in V_g and V_h, and since f is read-twice, we conclude that x is read exactly once each in g and in h. Hence α, β also do not involve x.

We construct a formula computing α as follows: In the sub-formula rooted at g, let ρ be the unique path from x to g. For each + gate u on the path ρ, let u' be the child of u not on ρ; replace u' by the constant 0. Thus we retain only the parts that multiply x; that is, we compute αx. Setting $x = 1$ gives us a formula G computing α. A similar construction with the formula rooted at h gives a formula H computing β. Set $F = G + H$. Note that F

is also a read-twice formula, and it computes $\alpha+\beta$. Thus in this case $\mathsf{ExistExtMon}(f, x) = 1 \Leftrightarrow \mathsf{PIT}(F) = 0$, so we need to determine $\mathsf{PIT}(F)$.

Let Y denote the set of variables appearing in F; $Y \subseteq X \setminus \{x\}$. Partition Y:

A: variables occurring only in G; B: variables occurring only in H;
C: variables occurring in G and H.

If $A \cup B = \emptyset$, then $Y = C$, and each variable in F appears once in G and once in H. That is, both G and H are read-once formulas. We can now determine $\mathsf{PIT}(F)$ in time polynomial in the size of F using Proposition 5.2.

If $A \cup B \neq \emptyset$, then let $y \in A$. If y survives in G, it cannot get cancelled by anything in H, so it survives in F and $F \not\equiv 0$. Similarly, if any $y \in B$ survives in H, then $F \not\equiv 0$. We briefly defer how to determine this and complete the high-level argument. If no $y \in A$ survives in G, and no $y \in B$ survives in H, then let $F' = G' + H'$ be the formula obtained from F, G, H by setting variables in $A \cup B$ to 0. Clearly, the polynomial computed remains the same; thus $\alpha + \beta = p_F = p_F|_{A \cup B \leftarrow 0} = p_{F'}$. But F' satisfies the previous case (with respect to F', $A' \cup B' = \emptyset$), and so we can use Proposition 5.2 as before to determine $\mathsf{PIT}(F') = \mathsf{PIT}(F)$.

What remains is to describe how we determine whether a variable $y \in A$ survives in G. (The situation for $y \in B$ surviving in H is identical.) We exploit the special structure of G: there is a path ρ where all the + gates have one argument 0 and the path ends in a leaf labeled 1. Let $\mathcal{T} = \{T_1, \ldots, T_\ell\}$ be the subtrees hanging off the $\times$ gates on ρ; let u_i be the root of T_i. Note that each $T_i \in \mathcal{T}$ is a sub-formula of our input formula ϕ, and hence by the iterative construction we know the values of the functions PIT, MLIN, ExistExtMon at gates in these sub-trees. In fact, we already know that $\mathsf{PIT}(u_i) = 0$ for all i, since we are in the situation where $\alpha \not\equiv 0$, and $\alpha = \prod_{i=1}^{\ell} p_{u_i}$. Hence, if y appears in just one sub-tree T_i, then $\mathsf{ExistExtMon}(G, y) = \mathsf{ExistExtMon}(u_i, y)$. If y appears in two sub-trees T_i, T_j, then $\mathsf{ExistExtMon}(G, y) = \mathsf{ExistExtMon}(u_i, y) \vee \mathsf{ExistExtMon}(u_j, y)$. □

A question that arises naturally here is whether this algorithm is optimal, or whether the PIT problem for read-twice formulas is in some class smaller than P. Note that the input formula F can be re-structured into an equivalent log-depth formula F', as described in [Bus87, BCGR92]. If the resulting formula is also read-twice, then it appears that the above algorithm can be applied to F', with a careful implementation to keep track of partial values, to yield an upper bound in NC. However, we have not examined the possibility of such an implementation, because it is not at all clear that the depth restructuring does actually preserve the number of times a variable is read.

Read-3 Formulas

The algorithm in the previous subsection crucially uses the PIT algorithm from [SV08] for k-sum-of-ROFs. A stronger result due to [AvMV11] gives PIT algorithms for read-k formulas that compute multilinear polynomials at each node. Using this algorithm instead of [SV08], we obtain poly-time PIT and MLIN tests for read-thrice (as opposed to read-twice) formulas. However, we pay a cost: we can no longer check at every node g whether a variable survives at g (the bit $\mathsf{ExistExtMon}(g, x)$). We can compute this information only at nodes g where all descendants compute multilinear formulas. The fact that we can compute $\mathsf{ExistExtMon}(g, x)$ everywhere in the read-twice case may be of independent interest (it seems to be a useful fact where enumerating monomials is concerned). In the following, we prove that for read-3 formulas, PIT and MLIN are in P:

Theorem 5.6. *Given a read-thrice formula F with leaves labeled by variables from $X = \{x_1, \ldots, x_n\}$ or constants from $\{-1, 0, 1\}$ and nodes labeled $+$ or $\times$, there is an efficient deterministic algorithm that decides if F computes the identically zero polynomial, and if not, whether it computes a multilinear polynomial.*

Proof. Algorithm Idea: We proceed bottom-up, processing nodes of the formula, collecting as much information as possible/necessary about the polynomial computed at each node. The type of information collected for a node g could be: $\mathsf{MLIN}(g)$, $\mathsf{PIT}(g)$, $\mathsf{ExistExtMon}(g, x)$ for $x \in X$.

For nodes g computing multilinear polynomials, we will compute all this information.

For nodes where we detect non-multilinearity (and hence know that the polynomial is not identically zero), we will not compute any additional information.

We repeatedly use collected information to prune the formula. For instance, we ensure that no leaf is labeled 0 by moving the zeroes up (replace $g + 0$ by g; $g \times 0$ by 0). We ensure that for each non-leaf node g, $\mathsf{var}(g) \neq \emptyset$ (replace a node adding or multiplying constants by a leaf labeled with the resulting value). Note that the resulting formulas can have any constants from $\mathbb{Z}$ at the leaves.

Further, for nodes g where we determine that the identically zero formula is computed, we will cut away the subformula rooted at g, replacing it by a leaf labelled zero, and then eliminate the zero-leaf as discussed above. Thus a node that is processed and not deleted necessarily computes a non-zero polynomial.

We will also maintain the following property: for nodes g where we determine that a multilinear formula is computed, the subformula rooted at g computes multilinear formulas

at each node.

Assume that we have a pruned formula. At a leaf, the required information is trivial to compute. Consider a node f where the information has been computed at the children of f.

Case 1: $f = g \times h$**.** $\mathsf{PIT}(f) = \mathsf{PIT}(g) \vee \mathsf{PIT}(h)$. However, by the pruning we have described above, we know that $g, h \not\equiv 0$ and so $f \not\equiv 0$, $\mathsf{PIT}(f) = 0$.

Since $f \not\equiv 0$, if either of g, h is non-multilinear then so is f. If both g and h are multilinear, then f is multilinear if and only no variable survives in both g and h. Thus we can compute $\mathsf{MLIN}(f)$ from the information at g, h:

$$\mathsf{MLIN}(f) = \mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h) \wedge \bigwedge_{x \in X} (\neg \mathsf{ExistExtMon}(g, x) \vee \neg \mathsf{ExistExtMon}(h, x))$$

When f is multilinear, we need to compute the auxiliary information as well. Note that we have already ensured that all nodes below g and h compute multilinear polynomials, so this property is already true for f. For any $x \in X$, $\mathsf{ExistExtMon}(f, x) = \mathsf{ExistExtMon}(g, x) \vee \mathsf{ExistExtMon}(h, x)$.

Case 2: $f = g + h$**.** Computing $\mathsf{MLIN}(f)$: Since the formula is read-thrice, if any one of g, h (say g) is not multilinear and hence has an x^2 term for some $x \in X$, then this term cannot be cancelled by the other summand (say h) since h has at most one occurerence of x. So f is not multilinear. If g, h are both multilinear, then so is $g + h$. Thus

$$\mathsf{MLIN}(f) = \mathsf{MLIN}(g) \wedge \mathsf{MLIN}(h)$$

Computing $\mathsf{PIT}(f)$: If f is not multilinear, then $f \not\equiv 0$ and so $\mathsf{PIT}(f) = 0$. In this case, we do not compute any further information about f.

But if f is multilinear, we still need to check if $f \equiv 0$. We have already ensured that all nodes in the sub-formulas rooted at g, h and hence in the sub-formula rooted at f compute multilinear polynomials. And the sub-formula is read-thrice. So using Proposition 5.3, we can test whether $f \equiv 0$.

Computing the remaining information: If we detect that a multilinear f is identically 0, we replace the subformula rooted at f by 0 and move the constants up as far as possible.

If we detect that f is multilinear but $f \not\equiv 0$, then we need to compute the bits $\mathsf{ExistExtMon}(f, x)$. By multilinearity, $f(X) = Ax + B$ where A, B do not use x.

We want to know if $A \equiv 0$ (this is equivalent to $\mathsf{ExistExtMon}(f, x) = 0$). A is computed by the formula $f|_{x=1} - f|_{x=0}$. We have already ensured that all nodes in the sub-formulas rooted at g, h and hence in the sub-formula rooted at f compute multilinear polynomials. Thus the formula for A is multilinear and reads every variable at most 6 times. Using Proposition 5.3, we can test whether $A \equiv 0$.

□

From now on, it is more convenient for us to allow leaves to be labeled by forms $ax+b$ for some $x \in X$ and some $a, b \in \mathbb{F}$. This does not change the class of polynomials computed, even when restricted to ROFs. Henceforth we assume that ROFs are of this form.

5.3 Identity testing for $\sum^{(2)} \cdot \prod \cdot$ROPs over $\mathbb{Z}$ or $\mathbb{Q}$

In this section we show that PIT can be solved efficiently for formulas presented in the form $f_1 f_2 \dots f_m + g_1 g_2 \dots g_s$, where each of the f_i, g_j is an ROF over the field of rationals.

Theorem 5.7. *Given Read-Once Formulas computing each of the polynomials $f_1, f_2, \cdots, f_r$, $g_1, g_2, \dots, g_s \in \mathbb{Q}[x_1, \dots, x_n]$, checking if $f_1 \cdot f_2 \cdots f_r \equiv g_1 \cdot g_2 \cdots g_s$ can be done in deterministic polynomial time.*

A crucial ingredient in our proof is the following structural characterization from [RS11, RS13] and its constructive version; this is a direct consequence of the characterisation of ROPs given in [SV08].

Lemma 5.8 ([RS13])**.** *Let $f \not\equiv 0$ be an ROP. Then exactly one of the following holds:*

1. *$k \geq 1$, there exist ROPs $f_1, \dots, f_k$, with $\mathsf{var}(f_i) \cap \mathsf{var}(f_j) = \emptyset$ for all distinct $i, j \in [k]$, such that $f = a + f_1 + \cdots + f_k$, for some $a \in \mathbb{F}$, and each f_i is either uni-variate or decomposes into variable-disjoint factors.*

2. *$k \geq 2$, there exist ROPs $f_1, \dots, f_k$, with $\mathsf{var}(f_i) \cap \mathsf{var}(f_j) = \emptyset$ for all distinct $i, j \in [k]$, such that $f = a \times f_1 \times f_2 \times \cdots \times f_k$ for some $a \in \mathbb{F} \setminus \{0\}$, and none of the f_is can be factorised into variable-disjoint factors.*

Furthermore, ROFs computing such f_is can be constructed from an ROF computing f in polynomial time.

Given an ROF over $\mathbb{Q}$, we can clear all denominators to get an ROF over $\mathbb{Z}$, without changing the status of the $? \equiv 0?$ question. So we now assume that all the numbers a, b appearing in the ROF (recall, leaf labels are of the form $ax + b$) are integers. For a polynomial $p(X)$, let $\mathsf{content}(p(X))$ denote the greatest common divisor (gcd) of the non-zero coefficients of p. The next crucial ingredient in our proof is that for an ROF f, we can efficiently compute its content.

Lemma 5.9. *There is a polynomial-time algorithm that, given an ROF f in $\mathbb{Z}[X]$, computes $\mathsf{content}(f)$ and constructs an ROF f' in $\mathbb{Q}[X]$ such that $f = \mathsf{content}(f) \cdot f'$.*

Proof. It suffices to show how to compute $\mathsf{content}(f)$; then the ROF f' is just $\frac{1}{\mathsf{content}(f)} \times f$. We proceed bottom-up, or alternatively, we prove this by induction on the structure of f.

For a polynomial $p \in \mathbb{Z}[X]$, let $\hat{p} = p - p(0)$, where $p(0) = p(0, \ldots, 0)$, and let $\hat{p}'$ be the polynomial such that $\hat{p} = \mathsf{content}(\hat{p})\hat{p}'$.

If f is a single leaf node, then computing $\mathsf{content}(f)$ and $\mathsf{content}(\hat{f})$ is trivial. Otherwise, say $f = g \circ h$. Since f is an ROF, $\mathsf{var}(g) \cap \mathsf{var}(h) = \emptyset$.

Case $f = g + h$: Then $\hat{f} = \hat{g} + \hat{h}$, and $f(0) = g(0) + h(0)$. So

$$\begin{aligned} \mathsf{content}(f) &= \gcd(\mathsf{content}(\hat{g}), \mathsf{content}(\hat{h}), g(0) + h(0)), \\ \mathsf{content}(\hat{f}) &= \gcd(\mathsf{content}(\hat{g}), \mathsf{content}(\hat{h})). \end{aligned}$$

Case $f = g \times h$: Then $\hat{f} = \hat{g}\hat{h} + h(0)\hat{g} + g(0)\hat{h}$, and $f(0) = g(0)h(0)$. We can show that

Claim 5.10. *For any two variable-disjoint polynomials $p, q \in \mathbb{Z}[X]$, $\mathsf{content}(pq) = \mathsf{content}(p)\mathsf{content}(q)$.*

Proof. Let $p = \mathsf{content}(p)(a_1M_1 + a_2M_2 + \cdots + a_kM_k)$ and $q = \mathsf{content}(q)(b_1N_1 + b_2N_2 + \cdots + b_\ell N_\ell)$, where M_i, N_j are monomials. By definition of content, $\gcd(\ldots, a_i, \ldots) = \gcd(\ldots, b_j, \ldots) = 1$. Since p and q are variable-disjoint, every monomial of the form $\mathsf{content}(p)\mathsf{content}(q)(a_ib_jM_iN_j)$ appears in the polynomial $p \times q$, and there are no other monomials. Hence $\mathsf{content}(p)\mathsf{content}(q)|\mathsf{content}(p \times q)$. For the converse, we need to show that $\gcd(S) = 1$, where $S = \{a_ib_j \mid i \in [k], j \in [\ell]\}$. Suppose not. Let c be the largest prime that divides all numbers in S. Then, $\forall i \in [k]$,

$$\begin{aligned} & c|a_ib_1 \text{ and } c|a_ib_2 \text{ and } \ldots \text{ and } c|a_ib_k. \\ \text{Hence} \quad & c|a_i \text{ or } (c|b_1, c|b_2, \cdots, c|b_\ell)\,. \\ \text{Hence} \quad & c|a_i \text{ or } c = 1, \text{ since } \gcd(b_1, \ldots, b_\ell) = 1. \end{aligned}$$

Thus we conclude that c divides $\gcd(a_1, \ldots, a_k) = 1$, a contradiction. □

Using this claim, we see that

$$\begin{aligned} \mathsf{content}(f) &= \mathsf{content}(g) \times \mathsf{content}(h), \\ \mathsf{content}(\hat{f}) &= \gcd(\mathsf{content}(\hat{g})\mathsf{content}(\hat{h}), h(0)\mathsf{content}(\hat{g}), g(0)\mathsf{content}(\hat{h})). \end{aligned}$$

□

Now we have all the ingredients for proving Theorem 5.7.

Proof of Theorem 5.7. Let $f = f_1 \cdot f_2 \cdots f_r$ and $g = g_1 \cdot g_2 \cdots g_s$ As discussed above, without loss of generality, each f_i, g_i is in $\mathbb{Z}[X]$. Using Lemmas 5.8 and 5.9, we can compute the irreducible variable-disjoint factors of each f_i and each g_i, and also pull out the content for each factor. That is, we express each f_i as $\alpha_i f_{i,1} \cdots f_{i,k_i}$, and each g_i as $\beta_i g_{i,1} \cdots g_{i,\ell_i}$ where the $f_{i,j}$s, $g_{i,j}$s are irreducible and have content 1. We obtain ROFs in $\mathbb{Q}[X]$ for each of the $f_{i,j}$s and $g_{i,j}$s. Note that if $\sum_i k_i \neq \sum_j \ell_j$, then there cannot be a component-wise matching between the factors of f and g, and hence we conclude $f \not\equiv g$. Otherwise, $\sum_i k_i = \sum_j \ell_j$. We now form multisets of the factors of f and of g, and we knock off equivalent factors one by one. (See Algorithm 1.) Detecting equivalent factors (the condition in Step 6) requires an identity test $p \equiv q$?, or $p - q \equiv 0$?, for ROFs in $\mathbb{Q}[X]$. Since we have explicit ROFs computing p and q, this can be done using Proposition 5.2. □

Recently, Amir Shpilka observed that the proof of Theorem 5.7 can be modified to work for polynomials over any field $\mathbb{F}$. We sketch the proof specifically for the part that is different from proof of Theorem 5.7:

Theorem 5.11 (Amir Shpilka (private communication))**.** *Given Read-Once Formulas computing each of the polynomials $f_1, f_2, \cdots, f_r, g_1, g_2, \ldots, g_s \in \mathbb{F}[x_1, \ldots, x_n]$, checking if $f_1 \cdot f_2 \cdots f_r \equiv g_1 \cdot g_2 \cdots g_s$ can be done in deterministic polynomial time.*

Proof. We use Lemma 5.8 to obtain a product of irreducible factors for each f_i and g_i. That is, we express each f_i as $f_{i,1} \cdots f_{i,k_i}$, and each g_i as $g_{i,1} \cdots g_{i,\ell_i}$ where the $f_{i,j}$s and $g_{i,j}$s are irreducible. Since these polynomials are over an arbitrary field $\mathbb{F}$, the notion of content does not exist. The factorization is now unique only upto scalar multiples. We show how to handle this. Similar to proof of Theorem 5.7, we want to find a match on the right side for each irreducible component from the left side. For ease of notation, let $p = f_{i,j}$ and

Algorithm 1 Test if $\prod_{i=1}^{r} \alpha_i \prod_{j=1}^{k_i} f_{i,j} \equiv \prod_{i=1}^{s} \beta_i \prod_{j=1}^{\ell_i} g_{i,j}$

$S \leftarrow \{f_{1,1}, \cdots, f_{1,k_1}, f_{2,1}, \cdots, f_{2,k_2}, \ldots, f_{r,1}, \cdots, f_{r,k_r}\}$
$T \leftarrow \{g_{1,1}, \cdots, g_{1,\ell_1}, g_{2,1}, \cdots, g_{2,\ell_2}, \ldots, g_{s,1}, \cdots, g_{s,\ell_s}\}$
(Both S and T are multisets; repeated factors are retained with multiplicity.)
for $p \in S$ **do**
    for $q \in T$ **do**
        if $p \equiv q$ **then**
            if S and T have unequal number of copies of p and q **then**
                Return No
            else
                $S \leftarrow S \setminus \{p\}$. (Remove all copies).
                $T \leftarrow T \setminus \{q\}$. (Remove all copies).
            end if
        end if
    end for
end for
if $(\alpha_1\alpha_2\cdots\alpha_r = \beta_1\beta_2\cdots\beta_s) \wedge (S = T = \emptyset)$ **then**
    Return Yes
else
    Return No
end if

$q = g_{u,v}$. We want to check if q is a match for p. i.e., is $p = cq$ for some $c \in \mathbb{F}$? p and q are both ROPs and hence the individual degree of each variable is at most 1. We know that $p \not\equiv 0$ and $q \not\equiv 0$. By Combinatorial Nullstellensatz ([Alo99] or see Proposition 5.13 below), it must be the case that there is an $\vec{a} \in \{0,1\}^n$ such that $p(a) \neq 0$. We find a using Algorithm 2. Step 3 of the algorithm can be achieved using Proposition 5.2. Once we have $\vec{a}$, we set $c = p(\vec{a})/q(\vec{a})$. We then check if $p - cq \equiv 0$ using Proposition 5.2. If yes, then we knock off p and q from their respective sides and continue this process of finding a component wise matching while retaining c as a scalar multiple on the right side. If $p - cq \not\equiv 0$, then q is not a match for p and we continue trying to find a match for p exactly like in proof of Theorem 5.7. If no match is found, then the inputs are not identically equal.

□

5.4 PIT for sums of powers of low degree polynomials

In this section, we give a blackbox identity testing algorithm for multilinear sums of powers of low-degree polynomials.

Algorithm 2 Find $\vec{a} \in \{0,1\}^n$ such that $p(a) \neq 0$

```
for k = 1 to n do
    a[k] ← 0
    if p|_{x_k=0} ≡ 0 then
        a[k] ← 1
        p ← p|_{x_k=1}
    else
        p ← p|_{x_k=0}
    end if
end for
```

We say that a polynomial f has a sum-powers representation of degree d and size s if there are polynomials f_i each of degree at most d, and a set of positive integers e_i, such that $f = f_1^{e_1} + \ldots + f_s^{e_s}$. In [Kay12], it is shown that computing the full multilinear monomial $\mathcal{M}_n = x_1 x_2 \cdots x_n$ using sums of powers of low-degree polynomials requires exponentially many summands:

Proposition 5.12. *[Kay12] There is a constant c such that for the polynomial $x_1 x_2 \cdots x_n$, any sum-powers representation of degree d requires size $s \geq 2^{\frac{cn}{d}}$.*

Shpilka and Volkovich [SV08] proved that sum of less than $n/3$ 0-justified ROPs cannot equal $\mathcal{M}_n$, and used it to obtain a black-box PIT algorithm for bounded sums of ROPs. Using these ideas along with Proposition 5.12, we note that such a hardness of representation for sums of powers of low-degree polynomials, where the final sum is multilinear, gives sub-exponential time algorithms for black-box PIT for this class.

Let $R = \{0,1\} \subseteq \mathbb{F}$ be a finite set that contains 0. For any $k > 0$, define

$$W_k^n(R) \triangleq \{\vec{a} \in R^n | \ \vec{a} \text{ has at most } k \text{ non-zero coordinates}\}.$$

In Theorem 7.4 of [SV10], it is shown that for a certain kind of formula F (k-sum of degree-d 0-justified preprocessed ROP), and for any $R \subseteq \mathbb{F}$ containing 0 and of size at least $d+1$, $F \equiv 0$ if and only if $F|_{W_{3k}^n(R)} \equiv 0$. The proof uses the Combinatorial Nullstellensatz [Alo99], see also Lemma 2.13 in [SV10]. We re-state it here for convenience:

Proposition 5.13 (Combinatorial Nullstellensatz, [Alo99])**.** *Let $P \in \mathbb{F}[x_1, \ldots, x_n]$ be a polynomial where for every $i \in [n]$, the degree of x_i is bounded by t. Let $R \subseteq \mathbb{F}$ have size at least $t+1$, and $S = R^n$. Then $P \equiv 0 \Leftrightarrow P|_S \equiv 0$.*

Along similar lines, using Propositions 5.12,5.13, we show that

Lemma 5.14. *Let $C(n,s,d)$ be the class of all n-variate multilinear polynomials that have a sum-powers representation of degree d and size s. Let c be the constant from*

Proposition 5.12. For $f \in C(n, s, d)$, $R = \{0, 1\}$, and $k = (d \log s)/c$, $f|_{W_k^n(R)} \equiv 0 \iff f \equiv 0$.

Proof. The $\Leftarrow$ direction in the claim is trivial. To prove the $\Rightarrow$ direction, we proceed by induction on n.

Base case: $n \leq k$. Then $W_k^n(R) = R^n$. Using Proposition 5.13 (since f is multilinear, R is large enough), we conclude that $f \equiv 0$.

Induction Step: $n > k$. Suppose $f \not\equiv 0$. Consider any $i \in [n]$, and let $f' = f|_{x_i=0}$. Then $f' \in C(n-1, s, d)$. Since $f|_{W_k^n(R)} \equiv 0$, we have $f'|_{W_k^{n-1}(R)} \equiv 0$. So by the induction hypothesis, $f' \equiv 0$. Hence $x_i | f$. Since this holds for every $i \in [n]$, the monomial $x_1 \cdots x_n$ must divide f. Since f is multilinear, it must be that $f = x_1 \cdots x_n$. But $n > k = (d \log s)/c$, so $s < 2^{cn/d}$. This contradicts Proposition 5.12. Hence we conclude $f \equiv 0$. □

This gives the required black-box PIT algorithm, since for our choice of k in the above lemma, $|W_k^n(\{0,1\})| \in n^{O(k)} \in 2^{O(d \log s \log n)}$. Thus

Theorem 5.15. *Let $C(n, s, d)$ be the class of all n-variate multilinear polynomials that have a sum-powers representation of degree d and size s. There is a deterministic black-box PIT algorithm for $C(n, s, d)$ running in time $2^{O(d \log n \log s)}$.*

We note that a more general result follows from Proposition 5.12 and Lemma 2.17 from [AvMV14] that was shown recently.

Remark 5.16. *Though f is multilinear in Lemma 5.14 (and hence Theorem 5.15), the polynomials f_i in the sum-powers representation of f need not be multilinear.*

5.5 Hardness of representation for sum of powers of CF-ROPs

The hardness of representation result from [Kay12], stated in Proposition 5.12, and its precursor from [SV08],[SV10], are both for $\mathcal{M}_n$, the former using low-degree polynomials and the latter using a kind of ROPs called 0-justified ROPs. Note that ROPs, even when 0-justified, can have high degree, so these results are incomparable. Here we extend such a hardness result in two ways.

Our first hardness result is for elementary symmetric polynomials $\mathsf{Sym}_{n,d}$, not just for $d = n$ when we get $\mathsf{Sym}_{n,d} = \mathcal{M}_n$. It works against another subclass of ROPs, CF-ROF;

as is the case in [SV08, SV10], this class too can have high-degree polynomials. Recall that this class consists of polynomials computed by read-once formulas that have + and $\times$ gates, and labels ax at leaves ($a \neq 0$). Hence for any f in this class, $f(0) = 0$. We show that powers of such polynomials cannot add up to elementary symmetric polynomials of arbitrary degree $d \leq n$ unless there are many such summands. First, we establish a useful property of this class.

Lemma 5.17. *For every* CF-ROP $f \in \mathbb{F}[x_1, \ldots, x_n]$*, there is a set* $S \subseteq [n]$ *with* $|S| \leq |\mathsf{var}(f)|/2$ *such that* $\deg(f|_{S\to 0}) \leq 1$.

Proof. Consider a CF-ROF F computing f. If F has a single node, then f is already linear, so set $S = \emptyset$. Otherwise, $F = G_1 \circ G_2$, where G_1, G_2 are variable-disjoint CF-ROFs computing CF-ROPs g_1, g_2, respectively.

Case 1: $\circ = \times$**.** Without loss of generality, assume $|\mathsf{var}(g_1)| \leq |\mathsf{var}(f)|/2$. For $S = \{i : x_i \in \mathsf{var}(g_1)\}$, $g_1|_{S\to 0} \equiv f|_{S\to 0} \equiv 0$.

Case 2: $\circ = +$**.** Inductively, we can find sets S_i of at most half the variables of each g_i, such that $g_i|_{S_i\to 0}$ has degree at most 1. Define $S = S_1 \cup S_2$. Since G_1, G_2 are variable-disjoint, $|S| \leq |\mathsf{var}(f)|/2$, and $f|_{S\to 0}$ has degree at most 1. □

We use this to get our hardness-of-representation result for CF-ROPs, irrespective of degree.

Theorem 5.18. *Fix any* $d \in [n]$*. Suppose there are CF-ROPs* $f_1, \ldots, f_k$*, and positive integers* $e_1, \ldots, e_k$ *such that*

$$\sum_{i=1}^{k} f_i^{e_i} = \mathsf{Sym}_{n,d}.$$

Then $k \geq \min\{\log \frac{n}{d}, 2^{\Omega(d)}\}$.

Proof. Let $f = \mathsf{Sym}_{n,d}$.

We repeatedly apply Lemma 5.17 to restrictions of the f_i's to obtain a formula of degree at most 1. Let $S_0 = T_0 = \emptyset$, and let S_{i+1} be the set obtained by applying the Lemma to $f_{i+1}|_{T_i\to 0}$, where each $T_i = S_1 \cup \ldots \cup S_i$. Define $S = T_k$. Since at least half the variables survive at each stage, we see that $r \triangleq |\mathsf{var}(f|_{S\to 0})| \geq |\mathsf{var}(f)|/2^k = n/2^k$.

- If $r \geq d$, then $f|_{S\to 0} = \mathsf{Sym}_{r,d} \not\equiv 0$. Add any $r - d$ surviving variables to the set S to obtain the expression $\mathsf{Sym}_{d,d} = f|_{S\to 0} = \sum_{i=1}^{k} (f_i|_{S\to 0})^{e_i}$ where each f_i is either linear or identically 0. Let k' be the number of non-zero polynomials $f_i|_{S\to 0}$. By Proposition 5.12, $k' \in 2^{\Omega(d)}$, and $k \geq k'$.

- If $r < d$, then $n/2^k \leq r < d$. So $k > \log(\frac{n}{d})$.

Thus if $k \leq \log \frac{n}{d}$, then $k \in 2^{\Omega(d)}$. □

What this tells us is that there is a threshold $r \sim \log \log n$ such that any sum-powers representation of $\mathsf{Sym}_{n,d}$ using CF-ROPs needs size $2^{\Omega(d)}$ for $d \leq r$, and size $\geq \log \frac{n}{d}$ for $d \geq r$.

Our second hardness result is for $\mathcal{M}_n$, but works against a different class of ROFs. These ROFs may not be constant-free, but they have bounded alternation-depth, and are also 0-justified. Again, first we establish a useful property of the class.

Lemma 5.19. *Let $f \in \mathbb{F}[x_1, \ldots, x_n]$ be computed by an ROF with alternation depth 3. For any degree bound $1 \leq d \leq n$, there is an $S \subseteq [n]$ of size at most $|\mathsf{var}(f)|/d$, and an assignment of values A_S to the variables x_i for $i \in S$, such that $\deg(f|_{S \to A}) \leq d$. Moreover, if f is 0-justified, then we can find an A_S with all non-zero values.*

Proof. Let f be computed by the ROF F with alternation depth 3, where no gate computes the 0 polynomial.

If the top gate in F is a +, then $F = \sum_{i=1}^{r} f_i$, where each summand f_i is of the form $\prod_{j=1}^{t_i} \ell_{i,j}$ and the factors $\ell_{i,j}$'s are linear forms on disjoint variable sets. We find a partial assignment that kills all summand of degree more than d. For each such summand f_i, identify the factor with fewest variables, and assign values to the variables in it to make it 0. We assign values to at most $|\mathsf{var}(f_i)|/d$ variables, so overall no more than $|\mathsf{var}(f)|/d$ variables are set.

Further, if f is 0-justified and read-once, then each f_i is also a 0-justified ROF. Hence no factor of f_i vanishes at 0; each factor $\ell_{i,j}$ is of the form $\sum_{k=1}^{p} a_{i,j,k} x_{i,j,k} - c_{i,j}$ where $c_{i,j} \neq 0$. We can kill such a factor with an assignment avoiding 0s (eg set $x_{i,j,k} = c_{i,j}/pa_{i,j,k}$.)

If the top gate in F is a $\times$, then $F = \prod_{i=1}^{r} F_i$, where the F_i have alternation depth 2 and are on disjoint variables. If f has degree more than d, it suffices to kill any one factor F_i to make the polynomial 0. Choosing the factor with fewest variables, and proceeding as above, we set no more than $|\mathsf{var}(f)|/d$ variables. Again, since F is an ROF, if F is 0-justified, then so are the F_i. So A_S can be chosen avoiding 0s. □

Using this, we get a hardness of representation result for 0-justified alternation-depth 3 ROPs.

Theorem 5.20. *Let $\epsilon \in (0, \frac{1}{2})$. If there are 0-justified, alternation-depth-3 ROPs $f_1, \ldots, f_s$, and non-negative integers $e_1, \ldots, e_s$ such that*

$$\sum_{i=1}^{s} f_i^{e_i} = x_1 \cdots x_n$$

then $s \geq n^{\frac{1}{2}-\epsilon}$.

Proof. Let d be a parameter to be chosen later. We identify a subset of variables S and an assignment A avoiding zeroes to variables of S, such that under this partial assignment, all the f_i's are reduced to degree at most d. We show that for any $d \in [n]$, this is possible with $|S| = t \leq \frac{s^2 n}{d}$. This gives a sum-powers representation of degree d and size s for $\prod_{x_i \notin S} x_i = M_{n-t}$. Invoking Kayal's result from Proposition 5.12, we see that $s \geq 2^{c(n-t)/d}$, and hence $\log s + \frac{cns^2}{d^2} \geq \frac{cn}{d}$. Choosing $d = 4n^{1-2\epsilon}$, we conclude that $s \geq n^{\frac{1}{2}-\epsilon}$.

The construction of S proceeds in stages. At the kth stage, polynomials $f_1, \ldots, f_{i-1}$ have already been reduced to low-degree polynomials, and we consider f_i. We want to use Lemma 5.19 at each stage. This requires that each polynomial f_i, **after all the substitutions from the previous stages**, is still a 0-justified ROF with alternation-depth 3. The alternation-depth-3 ROF is obvious; it is only maintaining 0-justified that is a bit tricky. We describe the construction for stage 1; the other stages are similar.

Applying Lemma 5.19 to f_1 with d as the parameter, we obtain a set R_1 of variables with $|R_1| \leq n/d$ and an assignment A_{R_1} avoiding 0, such that $\deg(f_1|_{R_1 \to A_{R_1}}) \leq d$. It may be the case that for some $i > 1$, the polynomial $f_i|_{R_1 \to A_{R_1}}$ is no longer 0-justified. We fix this by augmenting R_1 as follows.

Assume first that the ROFs for all the f_i's have top-gate +; we will discuss top-gate $\times$ later. So, as discussed in the proof of Lemma 5.19, each f_i has the form $\sum \prod \ell_{j,k}$ where each $\ell_{j,k}$ is a linear form. If $f_i|_{R_1 \to A_{R_1}}$ is not 0-justified, then some of the linear forms in it are homogeneous linear (no constant term). We identify such linear forms in each f_i, $i \geq 2$. Call this set L_1. That is,

$$L_1 = \left\{ \ell \;\middle|\; \begin{array}{l} \ell \text{ is a linear form at level-2 of some } f_i;\ \ell|_{R_1 \to A_{R_1}} \text{ is homogeneous linear but not} \\ \text{identically 0.} \end{array} \right\}$$

Since each f_i is a ROF, it contributes at most $|R_1|$ linear forms to L_1. Hence $|L_1| \leq (s-1)|R_1|$. Now pick a minimal set T_1 of variables from $X \setminus R_1$ that intersects each of the linear forms in L_1. By minimality, $|T_1| \leq |L_1| \leq (s-1)|R_1|$. We want to assign non-zero values A_{T_1} to variables in T_1 in such a way that for all $i \geq 2$, the $f_i|_{R_1 \to A_{R_1}; T_1 \to A_{T_1}}$ are 0-justified. We must ensure that the linear forms in L_1 become homogeneous (or

vanish altogether), and we must also ensure that previously non-homogeneous forms do not become homogeneous. To achieve this, consider

$$L_2 = \left\{ \ell \mid \begin{array}{l} \ell \text{ is a linear form at level-2 of some } f_i;\ \ell|_{R_1 \to A_{R_1}} \not\equiv 0; \\ \ell|_{R_1 \to A_{R_1}} \text{ contains a variable from } T_1. \end{array} \right\}$$

Clearly, $L_1 \subseteq L_2$. It suffices to find an assignment A_{T_1} to variables in T_1, avoiding zeroes, such that for each $\ell \in L_2$, either $\ell|_{R_1 \to A_{R_1}; T_1 \to A_{T_1}} \equiv 0$ or $\ell|_{R_1 \to A_{R_1}; T_1 \to A_{T_1}}(0) \neq 0$. For sufficiently large fields, such an assignment can always be found.

If some of the f_i's have top-gate $\times$, we need only a minor modification. We use this fact:

Observation 5.21. *If $F = \prod F_r$ is a read-once formula, then F is* 0-justified *if and only if for each r, F_r is* 0-justified *and satisfies $F_r(0) \neq 0$.*

Treat each factor of the polynomials with top-gate $\times$ exactly as we dealt with the other polynomials. Add their level-2 linear factors to L_1. Note that each such f_i can have many factors, but since it is read-once, any one variable can occur in at most one of these factors. So f_i still contributes no more than R_1 linear forms to L_1. Also modify the definition of L_2 to include also all linear forms at level 3 of such f_i's, containing a variable of T_1. Finally, look for an assignment also satisfying the additional condition that the factors do not vanish at 0. Again, over sufficiently large fields, it is possible to find such an assignment.

Now we set $S_1 = R_1 \cup T_1$, and $A_1 = A_{R_1} \cup A_{T_1}$. We have ensured the following:

1. $\deg(f_1|_{S_1 \to A_1}) \leq d$; and
2. for $i \geq 2$, $f_i|_{S_1 \to A_1}$ is 0-justified.

Furthermore, $|S_1| = |R_1| + |T_1| \leq |R_1|(1 + (s-1)) \leq sn/d$.

Other stages are identical, working on the polynomials restricted by the already-chosen assignments. Finally, $S = S_1 \cup \ldots \cup S_s$, and so $|S| \leq s^2 n/d$, as required. □

5.6 Conclusion and open problems

Our results show that for the restricted case of read-2 and read-3 formulas, PIT and MLIN can indeed be decided efficiently in the non-black box setting. We feel the techniques we use for read-2 should be helpful is attacking PIT for formulas that read variables more often. We conclude with some natural problems that are open:

- Can the results of [SV08] be extended to the case $\sum_{i=1}^{k} f_i^{r_i}$, where f_i's are ROFs?
- Can a hardness of representation for $\mathsf{Sym}_{n,d}$ be transformed into a polynomial identity test for a related model?
- Can the bound given by Theorem 5.18 be improved? We conjecture:

  Conjecture 2. *There is a constant $\epsilon > 0$ such that if there are CF-ROPs $f_1, \ldots, f_k$, and integers $e_1, \ldots e_k \geq 0$ satisfying*

  $$\sum_{i=1}^{k} f_i^{e_i} = \mathsf{Sym}_{n,n/2},$$

  then $k = \Omega(n^{\epsilon})$.
- Do the results of [AvMV11] extend to read-k-multilinear branching programs?
- Can we obtain a blackbox version of PIT for Read-2 and Read-3 formulas?

Chapter 6

Problems on Monomials

In this chapter we study the complexity of some natural computational problems that arise in the context of arithmetic circuits. These problems are defined over an input arithmetic circuit C and sometimes also an input monomial m. The problems we study are: (1) Computing the number of monomials in the polynomial computed by C (Section 6.2), (2) Checking if the coefficient of m in the polynomial computed by C is zero (Section 6.3), and (3) Checking if there is a monomial M in the polynomial computed by C such that $m \subseteq M$. We start with some definitions:

6.1 Preliminaries

An algebraic branching program (ABP for short) over a ring R is an undirected acyclic graph B with edges labeled from $\{x_1, \ldots, x_n\} \cup R$, and with two designated nodes, s with zero in-degree, and t with zero out-degree. For any directed path ρ in B, define

$$\mathsf{weight}(\rho) = \prod_{e\text{: an edge in } \rho} \mathsf{label}(e).$$

Any pair of nodes u, v in B computes a polynomial in $R[x_1 \ldots x_n]$ defined as follows:

$$p_B(u, v) = \sum_{\rho\text{: } \rho \text{ is a } u \rightsquigarrow v \text{ path in } B} \mathsf{weight}(\rho).$$

The ABP B computes the polynomial $p_B \triangleq p_B(s, t)$. We drop the subscript B from the above when clear from context.

Unless otherwise stated, we consider R to be the ring of integers $\mathbb{Z}$, and we allow only the constants $\{-1, 0, 1\}$ in the circuits and branching programs.

We consider the following restrictions of ABPs in increasing order of generality: (1) occur-once ABPs (OBP for short), where each variable appears at most once anywhere in the ABP (such BPs generalize ROFs), (2) read-once ABPs, where no path has two occurrences of the same variable, and (3) multilinear BPs, or MBPs, where the polynomial computed at every node is multilinear.

For any arithmetic circuit C, we let p_C denote the polynomial computed by C.

A formula of size s computing a polynomial on n variables can be converted to an ABP of size $O(s)$ computing the same polynomial. Furthermore, if a variable is read k times in the formula, then it appears k times as a label in the ABP. This is done by constructing the ABP from the formula inductively starting with the leaves and proceeding upwards. An ABP of size s can be converted to an equivalent arithmetic circuit of size $O(s)$. For more on conversion between formulas, ABPs and circuits, see [Nis91].

We use the following short-forms (some of these were introduced in 5) for some computational problems used in this chapter.

MonCount: Given an arithmetic circuit (or branching program) C, compute the number of monomials in the polynomial computed by C. For a polynomial p, $\#p$ denotes the number of monomials in p. For any gate g in C, let $\#g$ denote $\#p_g$. (The constant term, even if non-zero, does not count as a monomial.)

ZMC: Given an arithmetic circuit (or branching program) C, and a monomial m, test if $\mathsf{coeff}(p_C, m) = 0$ or not. We will use ZMCas a Boolean predicate that takes the value 1 if and only if $\mathsf{coeff}(p_C, m) = 0$.

ExistExtMon: Given an arithmetic circuit (or branching program) C, and a monomial m, test if there is a monomial M with non-zero coefficient in p_C such that M extends m; that is, $m|M$.

6.2 Counting Monomials

We now consider the MonCount problem. First we show that it is very hard even for read-twice formulas. Then we consider ROFs and OBPs. In both ROFs and OBPs, a monomial, once generated in a sub-formula/program, can be cancelled only by multiplication with a

zero polynomial. We exploit this fact to obtain efficient algorithms for counting monomials in ROFs and OBPs. We use the fact that for ROFs, almost everything (like PIT, MonCount, ZMC etc.) is easy and in P (See [SV08]).

6.2.1 Hardness of MonCount

We show that even for formulas that are monotone (no negative constants) and are read-twice, and furthermore, are decomposable as the sum of two read-once formulas, MonCount is as hard as #P.

Theorem 6.1. MonCount *is* #P *complete for the sum of two monotone read-once formulas.*

Proof. First we show hardness. Valiant showed [Val79] that the problem of computing the number of perfect matchings in a bipartite graph is #P hard. We reduce this to the problem of computing the number of monomials common to two monotone read-once alternation-depth two formulas. Then we reduce the latter problem to computing the number of monomials in the sum of two monotone alternation-depth two read once formulas.

Let $G = (U, V, E)$ be a bipartite graph with $|U| = |V| = n$. Let $X = \{x_{uv} \mid (u, v) \in E\}$. Define two polynomials

$$f = \prod_{u \in U} \Big(\sum_{v:(u,v)\in E} x_{uv} \Big); \qquad g = \prod_{v \in V} \Big(\sum_{u:(u,v)\in E} x_{uv} \Big)$$

Clearly, both f and g are computable by alternation-depth two read-once formulas. Consider a monomial m common to both f and g. Since m is in f, it contains, for each $u \in U$, exactly one variable of the form x_{uv}. Similarly, since m is in g, it contains, for each $v \in V$, exactly one variable of the form x_{uv}. Thus the set $\{(u, v) \mid x_{uv} \in m\}$ is a perfect matching in G. Conversely, any perfect matching M in G corresponds to a unique monomial $\prod_{(u,v)\in M} x_{uv}$ that is common to both f and g. Therefore, the number of perfect matchings in G is equal to the number of common monomials of f and g. Let $\#(f \cap g)$ denote the latter number.

Since f and g are monotone formulas, adding them cannot result in any monomial cancellations. Thus

$$\#(f + g) = \#f + \#g - \#(f \cap g)$$

Since $\#f$ and $\#g$ are ROFs, $\#f$ and $\#g$ can be computed easily in P. (Theorem 6.2 below shows that in fact it can be computed in DLOG.) Hence, using the above relation, comput-

ing $\#(f \cap g)$ reduces to computing $\#(f+g)$, and the reduction is computable in polynomial time (even logspace).

To see the #P upper bound, consider $f + g$, where f and g are monotone ROFs. A monomial m appears in $f + g$ if and only if it appears in at least one of f, g. Now define a nondeterministic machine M as follows: M guesses a monomial m, computes $a = \mathsf{ZMC}(f, m)$ and $b = \mathsf{ZMC}(g, m)$, and accepts if $a \wedge b = 0$. The number of accepting paths of M is exactly $\#(f + g)$. Since f and g are ROFs, a and b can be computed in polynomial time (Theorem 6.13 below shows that in fact it can be computed in DLOG). All potential monomials are multilinear and so can be guessed in polynomial time. Hence M is an NP machine, as required. □

6.2.2 Counting Monomials in Read-Once Formulas

Theorem 6.2. *Given a read-once formula F, $\mathsf{MonCount}(p_F)$ can be computed by an AC^0 circuit with oracle gates for GapNC^1 functions. Hence it can be computed in DLOG.*

Define a 0-1 valued bit $\mathsf{NZ}(g)$, to indicate whether or not the constant term of p_g is non-zero, as follows:

$$\mathsf{NZ}(g) = \begin{cases} 1 & \text{if } p_g(0) \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 6.3. *The language L defined below is in $\mathsf{C}_=\mathsf{NC}^1$:*

$$L = \{\langle F, g\rangle \mid F \text{ is an arithmetic formula, } g \text{ is a gate in } F, \text{ and } \ p_g(0) = 0\}$$

Proof. Convert F to formula F' where all variables are set to 0, and g is the output gate. Then F' evaluates to $p_g(0)$, so we need to check if F' evaluates to 0. By Proposition 5.1, this check can be performed in $\mathsf{C}_=\mathsf{NC}^1$. □

Proof. (of Theorem 6.2) Since F is a read-once formula, we can compute the value of $\#f$ for each gate f inductively, based on the structure of F beneath f. When f is a leaf node, it is labelled 0 or ± 1 or x_i for some i.

$$\#(0) = \#(\pm 1) = 0; \qquad \#(x_i) = 1.$$

Now assume f is not a leaf. Suppose $f = g+h$, then g and h are variable-disjoint read-once

formulas. Since the monomials of g and h are distinct,

$$\#f \quad = \quad \#g + \#h; \tag{6.1}$$

Finally, suppose $f = g \times h$, then again g and h are variable-disjoint. Each pair of monomials m in p_f and m' in p_g gives rise to a monomial mm' in p_f. In addition, if $p_g(0) \neq 0$, then each m' also appears as a monomial in p_f; similarly for $p_h(0)$ and m. Thus

$$\#f = [\#g \times \#h] + [\#g \times \mathsf{NZ}(h)] + [\mathsf{NZ}(g) \times \#h]. \tag{6.2}$$

Using Equation 6.1 and Equation 6.2, we can transform the given read-once formula F to a new formula F' over $\mathbb{Z}$ that computes $\mathsf{MonCount}(F)$. The transformation is local, and can be done in AC^0 with oracle access to $\mathsf{C}_=\mathsf{NC}^1$. For each gate f in F the local transformation can be described as follows: If f is a leaf gate, then relabel f by $\#f$. If $f = g + h$ gate, then it remains a + gate with children $\#g$ and $\#h$. If $f = g \times h$, using Equation 6.2 involves using $\#g$ and $\#h$ more than once, and so we do not get a formula. However, we can modify Equation 6.2 so that $\#f$ gets the structure of a formula, with oracle access to NZ. We use the identity

$$\#f = \#(g \times h) = (\#g + \mathsf{NZ}(g)) \times (\#h + \mathsf{NZ}(h)) - (\mathsf{NZ}(g) \times \mathsf{NZ}(h)).$$

The values $\mathsf{NZ}(g)$ and $\mathsf{NZ}(h)$ can be obtained with oracle access to the language L defined in Lemma 6.3. Now $\#g$ and $\#h$ are used only once.

Thus, from F we construct a formula F'' where the leaves of F'' are labeled by constants $0, \pm 1$ or by the outputs of $\mathsf{C}_=\mathsf{NC}^1$ oracle gates. Equivalently, in $\mathsf{AC}^0(\mathsf{C}_=\mathsf{NC}^1)$, we can transform F to formula F' whose leaves are labeled by $0, \pm 1$. By construction, F' is variable-free, and $\#p_F = \mathsf{val}(F')$. By Proposition 5.1, $\mathsf{val}(F')$ can be computed in GapNC^1, completing the proof. □

Remark 6.4. *The* AC^0 *circuit constructed above needs oracle access mainly to* $\mathsf{C}_=\mathsf{NC}^1$ *gates, which check whether a* GapNC^1 *function is zero or not. Only the topmost oracle query requires the entire value of the* GapNC^1 *function.*

For any polynomial p, $p \equiv 0$ if and only if the constant term of p is 0 and $\mathsf{MonCount}(\mathrm{p})$ is 0. Hence, from Theorem 6.2 and Lemma 6.3, we have the following:

Corollary 6.5. *In the non-blackbox setting,* PIT *on ROFs is in the* GapNC *hierarchy and hence in* DLOG.

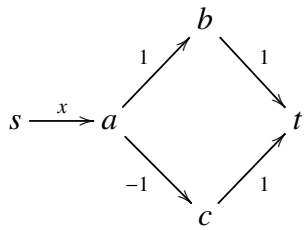
6.2.3 Counting Monomials in Occur-Once Branching Programs

We now show how to count monomials in OBPs. The approach used in Theorem 6.2 does not directly generalize to OBPs, *i.e.* knowing MonCount at immediately preceding nodes is not enough to compute MonCount at a given node in an OBP. However, since every variable occurs at most once in an OBP, every path generating a monomial involving a variable x should pass through the only edge labeled x. This allows us to keep track of the monomials at any given node of the OBP, given the monomial count of all of its predecessors.

We begin with some notations. Let B be an occur-once BP on the set of variables X, and u, v be any nodes in B. Let $c(u, v)$ be the constant term in $p(u, v)$. We define the 0-1 valued indicator function that describes whether this term is non-zero:

$$\mathsf{NZ}(u, v) = \begin{cases} 1 & \text{if } c(u, v) \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

We cannot directly use the strategy we used for ROFs, since even in an OBP, there can be cancellations due to the constant terms. For instance, in the figure below, $\#p(s, b) = \#p(s, c) = 1$, but $\#p(s, t) = 0$.



We therefore identify edges critical for a polynomial. We say that edge $e = (w, u)$ of B *is critical to* v if

1. $\mathsf{label}((w, u)) \in X$; and
2. B has a directed path ρ from u to v consisting only of edges labeled by $\{-1, 1\}$.

We have the following structural property for the monomials in $p(s, v)$:

Lemma 6.6. *In an occur-once OBP B with start node s, for any node v in B,*

$$p(s, v) = c(s, v) + \sum_{(w,u) \text{ critical to } v} p(s, w) \cdot \mathsf{label}(w, u) \cdot c(u, v) \ .$$

Proof. First, note that if edges $(w, u) \neq (w', u')$ are both critical to v, then the monomials in $p(s, w) \cdot \mathsf{label}(w, u)$ and $p(s, w') \cdot \mathsf{label}(w', u')$ will be disjoint, because P is occur-once.

(The variables labeling (w, u) and (w', u') make the monomials distinct.) Moreover, for any monomial m in $p(s, v)$, there is exactly one critical edge (w, u) such that the monomial m has non-zero coefficient in the polynomial $p(s, w) \times \mathsf{label}(w, u)$. The critical edge corresponds to the last variable of the monomial to be "collected" en route to v from s. This completes the proof. □

For nodes w, u, v in B where (w, u) is an edge, define a 0-1 valued indicator function that specifies whether or not (w, u) is critical to v. That is,

$$\mathsf{critical}(\langle w, u\rangle, v) = \begin{cases} 1 & \text{if } (w, u) \text{ is critical for } v \\ 0 & \text{otherwise} \end{cases}$$

Using this and Lemma 6.6, we can show:

Lemma 6.7. *In an occur-once OBP B with start node s, for any node v in B,*

$$\#p(s, v) = \sum_{e=(w,u)} \mathsf{critical}(\langle w, u\rangle, v) \cdot (\#p(s, w) + \mathsf{NZ}(s, w)) \cdot \mathsf{NZ}(u, v).$$

Proof. Consider the expression $p(s, w) \times \mathsf{label}(w, u)$, where (w, u) is an edge critical to v. Then $\mathsf{label}(w, u)$ is in X, and multiplies every monomial in $p(s, w)$. Hence every monomial of $p(s, w)$ contributes a monomial to $p(s, w) \times \mathsf{label}(w, u)$. Additionally, if $c(s, w) \neq 0$, then $c(s, w) \times \mathsf{label}(w, u)$ too contributes a monomial. Hence

$$\#[p(s, w) \times \mathsf{label}(w, u)] = \#p(s, w) + \mathsf{NZ}(s, w)\ .$$

Using this observation along with Lemma 6.6 completes the proof. □

If w is not in a layer to the left of v, then (w, u) cannot be critical to v, and so $\#p(s, w)$ is not required while computing $\#p(s, v)$. Hence we can sequentially evaluate $\#p(s, v)$ for all nodes v in layers going left to right, provided we have all the values $\mathsf{NZ}(s, w)$,$\mathsf{NZ}(u, v)$ and $\mathsf{critical}(\langle w, u\rangle, v)$.

Lemma 6.8. *Define languages L_1, L_2 as follows:*

$$L_1 = \{\langle B, u, v\rangle \mid B \text{ is an OBP, } u, v \text{ are nodes in } B, \text{ and } \mathsf{NZ}(u, v) = 0. \}$$

$$L_2 = \{\langle B, u, v, w\rangle \mid B \text{ is an OBP, } u, v, w \text{ are nodes in } B, \text{ and } \mathsf{critical}(\langle w, u\rangle, v) = 1. \}$$

Then L_1 and L_2 are both in $\mathsf{C_=L}$.

Proof. Delete from B all edges with labels from X to get a variable-free BP B'. Then $p_{B'}(u,v) = c_B(u,v)$. Checking whether $p_{B'}(u,v) = 0$ is the canonical complete problem for $\mathsf{C_=L}$. Hence L_1 is in $\mathsf{C_=L}$. To check membership in L_2, we need to check that $\mathsf{label}(w,u) \in X$ and that v is reachable from u in B'. This can be done in NLOG, which is contained in $\mathsf{C_=L}$. □

From Lemma 6.7, the comment following it, and Lemma 6.8, we obtain a polynomial time algorithm to count the monomials in p_B.

Theorem 6.9. *Given an occur-once branching program B, the number of monomials in p_B can be computed in* P.

With a little bit of care, we can obtain the following stronger result:

Theorem 6.10. *Given an occur-once branching program B, the number of monomials in p_B can be computed in the GapL hierarchy and hence in* NC^2.

Proof. Starting from B, we construct another BP B' as follows: B' has a node v' for each node v of B. For each triple w, u, v where (w,u) is an edge in B, we check via oracles for L_1 and L_2 whether (w,u) is critical to v and whether $\mathsf{NZ}(u,v) = 1$. If both checks pass, we add an edge from w' to v'. We also check whether $\mathsf{NZ}(s,w) = 1$, and if so, we add an edge from s' to v'. (We do this for every w, u, so we may end up with multiple parallel edges from s' to v'. To avoid this, we can subdivide each such edge added.) B' thus implements the right-hand-side expression in Lemma 6.7. It follows that $p_{B'}(s',v')$ equals $\#p_B(s,v)$. Note that B' can be constructed in logspace with oracle access to $\mathsf{C_=L}$. Also, since B' is variable-free, it can be evaluated in GapL. Hence $\#p_B$ can be computed in the GapL hierarchy. □

As in Corollary 6.5, using Theorem 6.10 and Lemma 6.8, we have:

Corollary 6.11. *In the non-blackbox setting,* PIT *on OBPs is in the* GapL *hierarchy and hence in* NC^2.

6.3 Zero-test on a Monomial Coefficient (ZMC)

From [FMM12a], ZMC is known to be in the second level of CH and hard for the class $\mathsf{C_=P}$. For the very restricted case of depth-3 read-3 formulas, ZMC is known to be NP-hard. (In Proposition 13 of [Str13], hardness is shown for depth-3 degree-3 formulas. It

can be verified that the hard formulas there are also read-thrice.) For the case of multilinear BPs (i.e. MBPs), we show that ZMC exactly characterizes the complexity class $\mathsf{C}_=\mathsf{L}$.

Theorem 6.12. *Given a BP B computing a multilinear polynomial p_B, and given a multilinear monomial m, the coefficient of m in p_B can be computed in* GapL. *Hence* ZMC *for multilinear BPs is complete for* $\mathsf{C}_=\mathsf{L}$.

Proof. We first show that ZMC, even for OBPs, is hard for $\mathsf{C}_=\mathsf{L}$. A complete problem for $\mathsf{C}_=\mathsf{L}$ is: does a BP B with labels from $\{-1, 0, 1\}$ evaluate to 0? Add a node t' as the new target node, and add edge $t \to t'$ labeled x to get B'. Then B' is an OBP, and $(B', x) \in \mathsf{ZMC}$ if and only if B evaluates to 0.

Now we show the upper bound. We show that given a branching program B computing a multilinear polynomial p_B, and given a multilinear monomial m, the coefficient of m in p_B can be computed in GapL. This will imply that the zero-test is in $\mathsf{C}_=\mathsf{L}$.

Let $S \subseteq [n]$ be such that $m = m_S$. Let $p_B = \sum_{T\subseteq[n]} \mathsf{coeff}(p_B, m_T)m_T$. We are interested in $\mathsf{coeff}(p_B, m_S)$. The idea is to construct a branching program B' computing a univariate polynomial, and a monomial m', such that $m \in p_B$ if and only if $m' \in p_{B'}$. We obtain B' by relabelling the edges of B as follows:

label in B	constant c	x_i for $i \in S$	x_i for $i \notin S$
label in B'	constant c	y	0

B' now computes a univariate polynomial $p_{B'}$ in y.

Observe that the coefficient c_S of m in p_B is equal to the coefficient of $y^{|S|}$ in $p_{B'}$. To see this, note that

$$p_B = \sum_{T\subseteq[n]} \mathsf{coeff}(p_B, m_T)m_T = \sum_{T\subseteq S} \mathsf{coeff}(p_B, m_T)m_T + \sum_{T\not\subseteq S} \mathsf{coeff}(p_B, m_T)m_T$$

The substitution described above sends the second sum to zero in B'. Hence,

$$p_{B'}(y) = \sum_{T\subseteq S} \mathsf{coeff}(p_B, m_T)y^{|T|} = \sum_{j=0}^{|S|} \left(\sum_{\substack{T\subseteq S \\ |T|=j}} \mathsf{coeff}(p_B, m_T) \right) y^j$$

The only monomial in p_B that generates $y^{|S|}$ in $p_{B'}$ is $\prod_{i\in S} x_i = m_S$; hence $\mathsf{coeff}(p_{B'}, y^{|S|}) = \mathsf{coeff}(p_B, m_S)$.

(This argument only requires that p_B be multilinear; we do not need B to be occur-once

or even read-once.)

Thus the problem now reduces to computing the coefficient of $y^{|S|}$ in B', which is a branching program over just one input variable. A standard construction allows us to explicitly construct all coefficients of $p_{B'}(y)$ in another branching program B''. For completeness, we describe the construction of B''. For each node v in B', B'' has $|S|+1$ nodes $v_0, \ldots, v_{|S|}$, with the intention that v_i should compute the coefficient of y^i in the polynomial $p_{B'}(s, v)$. The start node of B'' is the node s_0, and the final node is $t_{|S|}$. If edge (u, v) has label y in B', we include the edges (u_i, v_{i+1}) with label 1, for $0 \leq i < |S|$, in B''. If edge (u, v) has label $\ell \neq y$ in B', we include the edges (u_i, v_i) with label ℓ, for $0 \leq i \leq |S|$, in B''. By induction on the structure of B', we see that the value computed by B'' at $t_{|S|}$ is the coefficient of $y^{|S|}$ in $p_{B'}(s, v)$.

The above transformation from B' to B'' can be done in DLOG. Since B'' is variable-free, it can be evaluated in GapL. Composing these procedures, we obtain a GapL procedure for computing the coefficient of m in p_B. □

The upper bound above, for ZMC on MBPs, also applies to ROFs, since ROFs can be converted to OBPs by a standard construction. However, with a careful top-down algorithm, we can give a stronger upper bound of DLOG for ZMC on ROFs.

Theorem 6.13. *Given a read-once formula F computing a polynomial p_F, and given a multilinear monomial m, the coefficient of m in p_F can be computed in* DLOG. *Hence* ZMC *for ROFs is in* DLOG.

Proof. For any $T \subseteq [n]$, and any gate g in F, let $\alpha(g, T)$ denote the coefficient of monomial m_T in p_g. (That is, $\alpha(g, T) = \mathsf{coeff}(p_g, m_T)$.) Let r be the output gate. Let $S \subseteq [n]$ be such that $m = m_S$. The goal is to compute $\alpha(r, S)$. First, we observe some properties of α:

1. For any gate g and any $T \subseteq [n]$, if $T \not\subseteq \mathsf{subvar}_g$, then $\alpha(g, T) = 0$.
2. For a leaf g labelled x_i, $\alpha(g, T) = 1$ if $T = \{i\}$, 0 otherwise.
3. For a leaf g labelled by a constant c, $\alpha(g, T)$=c if $T = \emptyset$, 0 otherwise.
4. For an addition gate that computes $g + h$, $\alpha(g + h, T) = \alpha(g, T) + \alpha(h, T)$. And since F is an ROF, at least one of $\alpha(g, T), \alpha(h, T)$ is zero.
5. For a product gate that computes $g \times h$,

$$\alpha(g \times h, T) = \alpha(g, T \cap \mathsf{subvar}_g) \cdot \alpha(h, T \cap \mathsf{subvar}_h) \cdot [T \subseteq \mathsf{subvar}_g \cup \mathsf{subvar}_h].$$

This is because $\alpha(g \times h, T) = \sum_{Z \subseteq T} \alpha(g, Z)\alpha(h, T \setminus Z)$. But if either $Z \not\subseteq \mathsf{subvar}_g$ or $T \setminus Z \not\subseteq \mathsf{subvar}_h$, then $\alpha(g, Z) = 0$ or $\alpha(h, T \setminus Z) = 0$. Further, F is a read once formula, so $\mathsf{subvar}_g \cap \mathsf{subvar}_h = \emptyset$, and subvar_g and subvar_h partition $\mathsf{subvar}_{(g \times h)}$. Hence T must also be similarly partitioned.

Now we construct a formula F' whose evaluation gives us $\alpha(r, S)$. F' will recursively compute $\alpha(g, S \cap \mathsf{subvar}_g)$ for each gate g. If g is a leaf, we just use properties (2,3) to compute $\alpha(g, S \cap \mathsf{subvar}_g)$. We show how to compute $\alpha(f, S \cap \mathsf{subvar}_f)$ for an internal gate f with children g and h knowing the values for $\alpha(g, S \cap \mathsf{subvar}_g)$ and $\alpha(h, S \cap \mathsf{subvar}_h)$:

- Case $f = g + h$:

$$\begin{aligned} \alpha(f, S \cap \mathsf{subvar}_f) &= \alpha(g, S \cap \mathsf{subvar}_f) + \alpha(h, S \cap \mathsf{subvar}_f) && \text{from property (4)} \\ &= \alpha(g, S \cap \mathsf{subvar}_g)[S \cap \mathsf{subvar}_g = S \cap \mathsf{subvar}_f] \\ &+ \alpha(h, S \cap \mathsf{subvar}_h)[S \cap \mathsf{subvar}_h = S \cap \mathsf{subvar}_f] && \text{from property (1)} \end{aligned}$$

- Case $f = g \times h$:

$$\alpha(f, S \cap \mathsf{subvar}_f) = \alpha(g, S \cap \mathsf{subvar}_g) \cdot \alpha(h, S \cap \mathsf{subvar}_h) \quad \text{from properties (1,5)}$$

This gives us the formula F' that computes $\alpha(r, S)$ at the topmost gate. By Proposition 5.1, F' can be evaluated in GapNC^1. Constructing F' from F requires a local transformation at + gates and computation of the predicates $[S \cap \mathsf{subvar}_f = S \cap \mathsf{subvar}_g]$. By Proposition 5.4, these predicates can be computed in DLOG. □

For ROFs, the lower bound proof in Theorem 6.12 can be modified to show that ZMC on ROFs is hard for $\mathsf{C_=NC}^1$. It is natural to ask whether there is a matching upper bound. In our construction above, we need to compute predicates of the form $[x \in \mathsf{subvar}_g]$. If these can be computed in NC^1 for ROFs, then the monomial coefficients can be computed in GapNC^1 and hence the upper bound of ZMC can be improved to $\mathsf{C_=NC}^1$. However, this depends on the specific encoding in which the formula is presented. In the standard pointer representation, the problem models reachability in out-degree-1 directed acyclic graphs, and hence is as hard as DLOG. Perhaps, under some other encoding, an upper bound of NC^1 is possible. To see why this may be plausible, consider the problem of testing whether two trees are isomorphic. (And note that the undirected graph underlying

a formula is a tree.) For trees encoded as pointer lists, isomorphism testing is DLOG-complete, whereas for trees encoded as strings, the same problem of isomorphism testing is complete for NC^1 ([JMT98]).

6.4 Checking existence of monomial extensions

We now address the problem ExistExtMon. Given a monomial m, one wants to check if the polynomial computed by the input arithmetic circuit has a monomial M that extends m (that is, with $m|M$). This problem is seemingly harder than ZMC, and hence the bound of Theorem 6.12 does not directly apply to ExistExtMon. We show that ExistExtMon for OBPs is in the GapL hierarchy.

Theorem 6.14. *The following problem lies in the* GapL *hierarchy: Given an occur-once branching program B and a multilinear monomial m, check whether p_B contains any monomial M such that $m|M$.*

Proof. Let $S \subseteq [n]$ be such that $m = m_S$. If $S = \emptyset$, then this amounts to checking if $p_B \not\equiv 0$. By Corollary 6.11, this is in the GapL hierarchy. So now assume that $S \neq \emptyset$. We call an edge that is labelled by a variable from S a "bridge".

The algorithm is as follows:

if $\exists i \in S$ such that x_i does not appear in B at all **then**
  Output NO and halt.
else if $\exists$ layer l with more than one bridge to layer $l+1$ **then**
  Output NO and halt.
else
  For each layer l that has a bridge e to layer $l+1$ in B, remove all edges except e. Call the branching program thus obtained B'.
end if
Output $\overline{\mathsf{PIT}(p_{B'})}$ and halt.

We now show that m_S has an extended monomial in p_B if and only if the above algorithm outputs YES. If any of the variables of m_S do not appear at all in B, then clearly an extension to m_S cannot exist. So the algorithm rejects correctly. If there is a layer with more than one bridge to the next layer, then any path can go through at most one of these bridges. Since B is occur-once, every path would compute a monomial with at least one variable from m_S missing. So the algorithm correctly rejects. We are only interested in monomial extensions of m_S. So paths that do not go through all the bridges can be

ignored. Hence we can safely delete all non-bridge edges in layers which have a bridge to the next layer. Thus $p_{B'}$ is a polynomial where each monomial is an extension of m_S.

By Corollary 6.11, the above algorithm is in the GapL hierarchy. □

With a little bit of care, the above bound can be brought down to DLOG for the case of ROFs.

Theorem 6.15. *The following problem is in* DLOG*: Given a read-once formula F computing a polynomial p_F, and given a multilinear monomial m, check whether p_F contains any monomial M such that $m|M$.*

Proof. Let $S \subseteq [n]$ be such that $m = m_S$. If $S = \emptyset$, then this amounts to checking if $p_F \not\equiv 0$. By Corollary 6.5, this is in DLOG. So now assume that $S \neq \emptyset$. Similar to the case of branching programs, we transform F to a new formula F' as follows:

if $\exists x_i \in S$ such that x_i does not appear in F at all **then**
  Output NO and halt.
else if $\exists x_i, x_j \in S$, $i \neq j$, with LCA(x_i, x_j) in F labeled + **then**
  Output NO and halt.
else
  For every $x_i \in S$, and every + gate g on the unique leaf-to-root path γ from x_i, replace the input of g not on the path γ by 0.
  Let F' be the resulting formula.
end if
Output $\overline{\mathsf{PIT}(F')}$.

We show correctness of the above algorithm. Since F is read-once, if any of the two variables in S have a + gate as their least common ancestor, then m cannot appear as a monomial in F. If the algorithm reaches the **else** statement, then all sub-formulas that are additively related to some variable x_i in S are removed. This implies that every monomial produced by F' has m as a factor. Also, any monomial m' of p_F with $m|m'$ has the same coefficient in $p_{F'}$ as in p_F. Thus, the resulting formula F' computes a polynomial that contains exactly all monomials m' of p_F such that $m|m'$. This proves the correctness.

For the complexity bound, we note that the transformation from F to F' can be done in DLOG (using Proposition 5.4). Then by Corollary 6.5, the overall algorithm can be implemented in DLOG. □

6.5 Conclusion

Although one would expect the complexity of MonCount to reduce drastically for the case of severely restricted circuits, it remains #P hard for even read-twice formulas. However, the complexity of ZMC and ExistExtMon, does reduces drastically for the case of restricted circuits as expected. Ideally, we would like these problems to characterise complexity classes within P; we have partially succeeded in this. We leave open the question of extending these bounds for formulas and branching programs that are constant-read. It appears that this will require non-trivial modifications of our techniques.

Bibliography

[AAI+01] Manindra Agrawal, Eric Allender, Russell Impagliazzo, Toniann Pitassi, and Steven Rudich. Reducing the complexity of reductions. *Computational Complexity*, 10(2):117–138, 2001.

[AAR98] Manindra Agrawal, Eric Allender, and Steven Rudich. Reductions in circuit complexity: An isomorphism theorem and a gap theorem. *Journal of Computer and System Sciences*, 57(2):127–143, 1998.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[ABC+09] E. Allender, D. A. Mix Barrington, T. Chakraborty, S. Datta, and S. Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45(4):675–723, 2009.

[ABO99] Eric Allender, Robert Beals, and Mitsunori Ogihara. The complexity of matrix rank and feasible systems of linear equations. *Computational Complexity*, 8(2):99–126, 1999.

[Agr10] Manindra Agrawal. The isomorphism conjecture for constant depth reductions. *Journal of Computer and System Sciences*, 77(1):3–13, 2010.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. *SIAM Journal on Computing*, 36(4):845–888, 2006.

[AIK08] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. On pseudorandom generators with linear stretch in NC^0. *Computational Complexity*, 17(1):38–69, 2008.

[AIK09] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography with constant input locality. *Journal of Cryptology*, 22(4):429–469, 2009.

[Ajt83] Miklós Ajtai. $\sum_1^1$-formulae on finite structures. *Annals of Pure and Applied Logic*, 24(1):1 – 48, 1983.

[Ajt94] Miklós Ajtai. The complexity of the pigeonhole principle. *Combinatorica*, 14(4):417–433, 1994.

[All04] E. Allender. Arithmetic circuits and counting complexity classes. In Jan Krajicek, editor, *Complexity of Computations and Proofs*, Quaderni di Matematica Vol. 13, pages 33–72. Seconda Universita di Napoli, 2004.

[Alo99] Noga Alon. Combinatorial nullstellensatz. *Combinatorics, Problem and Computing*, 8, 1999.

[AO94] E. Allender and M. Ogihara. Relationships among PL, #L, and the determinant. In *Structure in Complexity Theory Conference, 1994., Proceedings of the Ninth Annual*, pages 267–278, Jun 1994.

[AV08] Manindra Agrawal and V. Vinay. Arithmetic circuits: A chasm at depth four. In *Foundations Of Computer Science*, pages 67–75, 2008.

[AV11] Vikraman Arvind and T. C. Vijayaraghavan. The orbit problem is in the GapL hierarchy. *Journal of Combinatorial Optimization*, 21(1):124–137, 2011.

[AvMV11] Matthew Anderson, Dieter van Melkebeek, and Ilya Volkovich. Derandomizing polynomial identity testing for multilinear constant-read formulae. In *IEEE Conference on Computational Complexity*, pages 273–282, 2011.

[AvMV14] Matthew Anderson, Dieter van Melkebeek, and Ilya Volkovich. Derandomizing polynomial identity testing for multilinear constant-read formulae. *To Appear in Computational Complexity*, 2014.

[Bar89] D.A. Barrington. Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[BCD+89] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal on Computing*, 18(3):559–578, June 1989.

[BCGR92] S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992.

[BDK+13] Olaf Beyersdorff, Samir Datta, Andreas Krebs, Meena Mahajan, Gido Scharfenberger-Fabian, Karteek Sreenivasaiah, Michael Thomas, and Heribert Vollmer. Verifying proofs in constant depth. *ACM Transactions on Computation Theory*, 5(1):2:1–2:23, May 2013. A preliminary version appeared in [BDM+11].

[BDM+11] Olaf Beyersdorff, Samir Datta, Meena Mahajan, Gido Scharfenberger-Fabian, Karteek Sreenivasaiah, Michael Thomas, and Heribert Vollmer. Verifying proofs in constant depth. In *Proceedings of 36th Mathematical Foundations of Computer Science Symposium(MFCS)*, pages 84–95, 2011.

[BE11] Markus Bläser and Christian Engels. Randomness efficient testing of sparse black box identities of unbounded degree over the reals. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 555–566, 2011.

[BHS08] Markus Bläser, Moritz Hardt, and David Steurer. Asymptotically optimal hitting sets against polynomials. In *International Colloquium on Automata, Languages, and Programming(ICALP)*, pages 345–356, 2008.

[BLM93] F. Bedard, F. Lemieux, and P. McKenzie. Extensions to Barrington's M-program model. *Theoretical Computer Science*, 107:31–61, 1993.

[BP01] Paul Beame and Toniann Pitassi. Propositional proof complexity: Past, present, and future. In G. Paun, G. Rozenberg, and A. Salomaa, editors, *Current Trends in Theoretical Computer Science: Entering the 21st Century*, pages 42–70. World Scientific Publishing, 2001.

[BT88] D.A. Barrington and D. Thérien. Finite monoids and the fine structure of NC^1. *Journal of the Association of Computing Machinery*, 35:941–952, 1988.

[Bus87] S. Buss. The Boolean formula value problem is in ALOGTIME. In *Symposium on Theory of Computation(STOC)*, pages 123–131, 1987.

[CD06] Tanmoy Chakraborty and Samir Datta. One-input-face MPCVP is hard for L, but in LogDCFL. In S. Arun-Kumar and Naveen Garg, editors, *Foundations of Software Technology and Theoretical Computer Science(FSTTCS)*, volume 4337 of *Lecture Notes in Computer Science*, pages 57–68, Kolkata, India, 2006. Springer.

[CK07] Stephen A. Cook and Jan Krajíček. Consequences of the provability of NP $\subseteq$ P/poly. *Journal of Symbolic Logic(JSL)*, 72(4):1353–1371, 2007.

[CM87] Stephen A. Cook and Pierre McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8(3):385–394, 1987.

[CM01] Mary Cryan and Peter Bro Miltersen. On pseudorandom generators in NC^0. In *Proceedings of 26th Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 272–284, 2001.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the annual symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

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

[CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic(JSL)*, 44(1):36–50, 1979.

[DS07] Zeev Dvir and Amir Shpilka. Locally decodable codes with two queries and polynomial identity testing for depth 3 circuits. *SIAM Journal on Computing*, 36(5):1404–1434, 2007.

[FMM12a] Hervé Fournier, Guillaume Malod, and Stefan Mengel. Monomials in arithmetic circuits: Complete problems in the counting hierarchy. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 362–373, 2012.

[FMM12b] Hervé Fournier, Guillaume Malod, and Stefan Mengel. Monomials in arithmetic circuits: Complete problems in the counting hierarchy. In *STACS*, pages 362–373, 2012.

[FSS84] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[GKKS13] Ankit Gupta, Pritish Kamath, Neeraj Kayal, and Ramprasad Saptharishi. Arithmetic circuits: A chasm at depth three. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2013. To appear.

[GKP06] Igor Grunsky, Oleksiy Kurganskyy, and Igor Potapov. On a maximal nfa without mergible states. In *Computer Science Symposium in Russia (CSR)*, pages 202–210, 2006.

[Gol77] L. M. Goldschlager. The monotone and planar circuit value problems are logspace complete for P. *SIGACT News*, 9(2):25–29, 1977.

[HAB02] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65(4):695–716, 2002.

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

[Hås86] Johan Håstad. Almost optimal lower bounds for small depth circuits. In *Procedings of 18th Symposium on Theory of Computing (STOC)*, pages 6–20. IEEE Computer Society, 1986.

[Hås87] Johan Håstad. One-way permutations in NC^0. *Information Processing Letters*, 26(3):153–155, 1987.

[HI10] Edward A. Hirsch and Dmitry Itsykson. On optimal heuristic randomized semidecision procedures, with application to proof complexity. In *of 27th annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 453–464, 2010.

[Hir10] EdwardA. Hirsch. Optimal acceptors and optimal proof systems. In Jan KratochvÃŋl, Angsheng Li, JiÅŹÃŋ Fiala, and Petr Kolman, editors, *Theory and Applications of Models of Computation (TAMC)*, volume 6108 of *Lecture Notes in Computer Science*, pages 28–39. Springer Berlin Heidelberg, 2010.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[JMT98] Birgit Jenner, Pierre McKenzie, and Jacobo Torán. A note on the hardness of tree isomorphism. In *IEEE Conference on Computational Complexity*, pages 101–105, 1998.

[JQS10] Maurice J. Jansen, Youming Qiao, and Jayalal M. N. Sarma. Deterministic black-box identity testing π-ordered algebraic branching programs. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 296–307, 2010.

[Jus72] J. Justesen. Class of constructive asymptotically good algebraic codes. *IEEE Transactions on Information Theory*, 18(5):652–656, 1972.

[Kay12] Neeraj Kayal. An exponential lower bound for the sum of powers of bounded degree polynomials. *ECCC*, 19(TR12-081):81, 2012.

[KI04] Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity*, 13(1-2):1–46, 2004.

[KLMS13] Andreas Krebs, Nutan Limaye, Meena Mahajan, and Karteek Sreenivasaiah. Small depth proof systems. In *Mathematical Foundations of Computer Science (MFCS) 2013*, volume 8087 of *LNCS*, pages 583–594. Springer Berlin Heidelberg, 2013.

[KP07] Pascal Koiran and Sylvain Perifel. The complexity of two problems on arithmetic circuits. *Theoretical Computer Science*, 389(1-2):172–181, 2007.

[KPW95] Jan Krajícek, Pavel Pudlák, and Alan R. Woods. An exponenetioal lower bound to the size of bounded depth Frege proofs of the pigeonhole principle. *Random Structures and Algorithms*, 7(1):15–40, 1995.

[KS01] Adam Klivans and Daniel A. Spielman. Randomness efficient identity testing of multivariate polynomials. In *Symposium on the Theory of Computing (STOC)*, pages 216–223, 2001.

[KS07] Neeraj Kayal and Nitin Saxena. Polynomial identity testing for depth 3 circuits. *Computational Complexity*, 16(2):115–138, 2007.

[MRS14a] Meena Mahajan, B. V. Raghavendra Rao, and Karteek Sreenivasaiah. Monomials, multilinearity and identity testing in simple read-restricted circuits. *Theoretical Computer Science*, 524:90–102, 2014. preliminary version in MFCS 2012.

[MRS14b] Meena Mahajan, Raghavendra Rao, and Karteek Sreenivasaiah. Building above read-once polynomials: identity testing and hardness of representation. In *International Computing and Combinatorics Conference (COCOON)*, 2014. To appear.

[MST06] Elchanan Mossel, Amir Shpilka, and Luca Trevisan. On ε-biased generators in NC^0. *Random Structures and Algorithms*, 29(1):56–81, 2006.

[Nis91] Noam Nisan. Lower bounds for non-commutative computation (extended abstract). In *Proceedings of the 23rd ACM Symposium on Theory of Computing (STOC)*, pages 410–418, 1991.

[Pud09] Pavel Pudlák. Quantum deduction rules. *Annals of Pure and Applied Logic*, 157(1):16–29, 2009. See also ECCC TR07-032.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *Journal of ACM*, 55(4), 2008. (Originally appeared in STOC '05).

[RS11] B. V. Raghavendra Rao and Jayalal M. N. Sarma. Isomorphism testing of read-once functions and polynomials. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 115–126, 2011.

[RS13] B. V. Raghavendra Rao and Jayalal M. N. Sarma. Isomorphism testing of read-once functions and polynomials. manuscript, 2013.

[Sax14] Nitin Saxena. Progress on polynomial identity testing - ii. *CoRR*, abs/1401.0976, 2014.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of ACM*, 27(4):701–717, 1980.

[Seg07] Nathan Segerlind. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 13(4):417–481, 2007.

[Str13] Yann Strozecki. On enumerating monomials and other combinatorial structures by polynomial interpolation. *Theory of Computing Systems*, 53(4):532–568, 2013.

[SV08] Amir Shpilka and Ilya Volkovich. Read-once polynomial identity testing. In *Symposium on Theory of Computation(STOC)*, pages 507–516, 2008.

[SV10] Amir Shpilka and Ilya Volkovich. Read-once polynomial identity testing. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:11, 2010.

[SY10] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010.

[Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica.*, 26(3):279–284, November 1988.

[Val79] Leslie G. Valiant. The complexity of computing the permanent. *Theorerical Computer Science*, 8:189–201, 1979.

[Veb12] Oswald Veblen. An application of modular equations in analysis situs. *Annals of Mathematics*, 14(1/4):pp. 86–94, 1912.

[Ven91] H. Venkateswaran. Properties that characterize LOGCFL. *Journal of Computer and System Sciences*, 43(2):380 – 404, 1991.

[Vio11] Emanuele Viola. Extractors for circuit sources. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 220–229, 2011.

[Vio12] Emanuele Viola. The complexity of distributions. *SIAM Journal on Computing*, 41(1):191–218, 2012.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity – A Uniform Approach*. Texts in Theoretical Computer Science. Springer Verlag, Berlin Heidelberg, 1999.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*. Wiley-Teubner series in computer science. B. G. Teubner & John Wiley, Stuttgart, 1987.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International Symposium on Symbolic and Algebraic Computation (EUROSAM)*, pages 216–226, 1979.