Succinct Numbers, Skew Circuits and Bounded Treewidth Graphs: Algorithms and Complexity

By

Nikhil Balaji

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

to

Chennai Mathematical Institute
July 2016

CHENNAI MATHEMATICAL INSTITUTE

Plot No-H1, SIPCOT IT Park,
Padur Post, Siruseri,
Tamilnadu-603103

Declaration

I declare that this thesis titled, ”Succinct Numbers, Skew Circuits and Bounded Treewidth Graphs: Algorithms and Complexity” submitted by me for the degree of Doctor of Philosophy is the record of work carried out by me during the period from August 2012 to August 2015 under the guidance of Prof. Samir Datta. This work has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other university or other similar institution of higher learning.

Nikhil Balaji

July 2016
Chennai Mathematical Institute
Plot H1, SIPCOT IT Park, Siruseri,
Kelambakkam 603103
India

Certificate

This is to certify that the Ph.D. thesis submitted by Nikhil Balaji to Chennai Mathematical Institute, titled "Succinct Numbers, Skew Circuits and Bounded Treewidth Graphs: Algorithms and Complexity" is a record of bona fide research work done during the period August 2012 to August 2015 under my guidance and supervision. The research work presented in this thesis has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this institute or any other university or institution of higher learning. It is further certified that the thesis represents independent work by the candidate and collaboration when existed was necessitated by the nature and scope of problems dealt with.

Samir Datta
(Thesis superviser)

July 2016
Chennai Mathematical Institute
Plot H1, SIPCOT IT Park, Siruseri,
Kelambakkam 603103
India

Abstract

In this thesis, we concern ourselves with three broad themes that are loosely tied together by the aim to improve our understanding of the complexity of counting problems:

1. *Inferring properties of succinctly represented numbers and matrices:* We study the complexity of computing some simple properties of numbers represented via arithmetic circuits. We give improved algorithms (all of which lie in the Counting Hierarchy) for the problems of obtaining an arbitrary bit of a) A number represented by an arithmetic circuit, b) A constant-sized matrix with n-bit entries raised to n-bit powers c) An n-bit number raised to n-bit powers. We also show a weak reduction (in the Counting Hierarchy) of the classic Sum of Square Roots problem to the problem of powering 2×2 matrices to n-bit powers.

2. *Skew circuits of small width:* We study width 5 branching programs and Barrington's Theorem [Bar89] via a special class of structurally restricted circuits, namely *skew* circuits. We prove that skew circuits of width 7 capture NC^1. We then study the complexity hierarchy within width 7, and show some separations and inclusions between these classes. Our main contribution here is a proof that the parity function requires exponential-size width 3 skew circuits.

3. *Counting problems on bounded treewidth graphs:* We first study the complexity of computing linear algebraic invariants like the Determinant, rank, characteristic polynomial, etc. for matrices whose underying graph is bounded treewidth. We show tight Logspace upper and lower bounds for all these invariants. Next we turn our attention to the problem of counting Euler tours on bounded treewidth graphs. Our Logspace Determinant algorithm gives an efficient way to count Euler tours on directed graphs. In the case of undirected graphs, this problem is known to be $\#\mathsf{P}$ hard on general graphs. We give a $\#\mathsf{SAC}^1$ algorithm for this problem on bounded treewidth graphs.

Acknowledgements

Firstly I thank Samir for being a great advisor and collaborator, for teaching me whatever Complexity Theory I know and for being extraordinarily patient with my procrastination and oddities. Samir treated me like a collaborator from day one and has been extremely generous with his time and ideas. I hope I can display a fraction of his intensity and enthusiasm in the years to come.

I thank my collaborators – Eric Allender, Samir Datta, Venkatesh Ganesan, Andreas Krebs and Nutan Limaye – for letting me include our joint work in this thesis. I would like to thank Stefan Mengel for his email correspondence regarding the work reported in Chapter 5 reporting an error in one of our claimed "proofs". Let me also thank the anonymous referees of various conferences that we submitted to, for very helpful comments which directly helped improve the quality of presentation of my papers (and by extension, this thesis).

I thank Nutan Limaye for hosting me at IIT Bombay, working with me on skew circuits, for being an excellent mentor and for offering me my first job (which was one of the primary motivations to write up this thesis quickly!). Many thanks to Raghav Kulkarni for hosting me at CQT, Singapore, and introducing me to the beautiful area of decision trees and evasiveness. It was a short and memorable visit, and I thank Raghav, Supartha and Samir for the long, enlightening discussions. I thank Heribert Vollmer and Arne Meier for hosting me at Hannover, Andreas Krebs for hosting me at Tübingen – Parts of work reported in Chapter 4 were done during these visits; I learned a lot of cool things and thoroughly enjoyed the discussions and I thank the theory groups at Hannover and Tübingen for their hospitality.

Many thanks to the faculty of CMI and IMSc for some wonderful courses over the last few years. I hope one day I can teach as well as I was taught in some of these courses. I'd especially like to thank Arvind, Meena and Partha for their amazing courses in Complexity Theory that got me interested in the first place.

I thank CMI for funding my research, conference travel, for all the excellent facilities and for being the amazingly cool place that it is. I have spent most of my the last five years going back and forth to CMI and I thank my fellow *shuttlers* for being an entertaining bunch. Let me also thank the administrative staff of CMI (Rajeshwari, Ranjini, Sripathy and Viji) for being super-efficient and making my stay completely hassle-free.

I thank Abhishek, Anish, Gaurav, Karteek, Nitin, Nitesh, Partha, Prajakta, Raja, Raghu, Rameshwar, Sajin, Supartha, Suryajith and Yadu for several interesting discussions I have had over the years, on Complexity Theory. Thanks to Srivathsan for all the LaTeX and TikZ help!

Thanks to Rohith for being a great friend, flatmate and for teaching me most of the mathematics I know. Thanks to Suratno for being the coolest person I have ever known – I hope, one day I can be as absent-minded and condy as Suratno (:-)). Thanks to comrades Abhishek, Alok, Rohith and Suratno for their opinions, advice and help – they have influenced me in more ways than I can count, and I thank them for their company.

Many thanks to my flatmates (both official and honorary) over the years – Abhishek, Anirban, Biswaroop, Chary, Debbie, Jagdheesh, Jose, Naveen, Pabitra, Rohith, Suratno for all the fun times we had at 16/4. Let me also thank my drinking buddies – too long to list – for all the terrace drinking scenes; if not for you guys, I might have finished this thesis earlier (and clearly regretted it!) :-) I thank Flappy and company for the evening's entertainment.

Thanks to the bros from TVS and the Machans of Surathkal for weathering out different time zones, distances and staying in touch, besides the much-needed wackiness over the years.

Finally, I thank Amma, Balaji Mama, Mala Chithi, Nalini Chithi, Sudha Chithi and the rest of my family for *everything*. This thesis is as much theirs as is mine and it is to them that I dedicate this thesis.

Nikhil Balaji
Chennai,
September 2015.

To

Residents (and non-residents[*]) of 3, Doak Nagar

[*]Terms and Conditions apply

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Counting problems and complexity classes defined via counting problems have been an integral part of Complexity Theory in the last four decades. The most well known among them is Valiant's class, #P which is the class of functions that count the number of solutions to an instance of a problem in NP. #P has been at the center of many exciting developments in various aspects of Complexity Theory, including Structural Complexity[FFK94], small depth circuits [BRS95], and approximate counting and sampling problems [JS89, SJ89]. #P and the Counting Hierarchy are an important bridge between hypotheses in Valiant's theory like VP vs VNP[1] and separations in Boolean complexity owing to the fact that the Permanent is a complete function for #P whereas the Permanent polynomial is complete for VNP [Val79a, Val79b].

In this thesis, we are concerned with the Complexity Theoretic aspects of three broad themes which are united by their significance in understanding the complexity of counting problems:

1. *Inferring elementary properties of succinctly represented numbers:* Computational complexity theory works under the premise that we have a specific way to represent an instance of the problem (e.g., representing graphs as adjacency matrices), and there is a specific model(e.g.,Turing machines) of computation for which we want to prove upper and lower bounds; A problem that is trivial in one choice of representation/ model of computation might be very hard in another choice of representation/computational model. A common example of this phenomenon is the Polynomial Identity Testing problem : Given a multivariate polynomial with coefficients over a field, we want to check if the polynomial is uniformly zero. If the polynomial is specified by as list

[1]The overarching goal of this theory is to classify polynomials based on hardness of formally computing them via arithmetic circuits. However, another important reason for interest in the arithmetic circuit model is due to the fact that they are more restricted than Boolean circuits and hence in principle should be an easier model to prove lower bounds for. Valiant argues in [Val92] that VP vs VNP might be a much less formidable goal than the complexity class separations in the Boolean world.

of coefficients, this problem has a simple linear time (linear in the size of the input representation, here it is the vector of coefficients) algorithm. However, when the polynomial is represented as a bunch of instructions (equivalently Straight Line Programs (SLP) or arithmetic circuits), we do not know of any deterministic sub-exponential time (sub-exponential in the size of the SLP) algorithm. Inferring properties of succinctly represented mathematical objects poses many challenges that have theoretical and practical applications, and are far from resolved. We describe our contributions to this area in Section 1.1.

2. *Skew Circuits of small width:* Since lower bounds in the Turing machine model are extremely hard, research in lower bounds turned to stronger combinatorial models of computation. A non-uniform model of computation is one where we give the algorithm designer the option to come up with different algorithms for different input lengths of the problem instance. Circuits and branching programs are two well-studied nonuniform models of computation. Barrington's theorem gives a proof of equivalence between two restricted classes of these non-uniform models of computation. The key import of Barrington's theorem is that it gives a way to count using a constant amount of space. Our work analyzes Barrington's theorem on branching programs via a restricted form of Boolean circuits called *skew* circuits. This forms the content of Section 1.2.

3. *Counting Problems on Bounded Treewidth graphs:* When a graph is represented via its adjacency matrix, it is natural to ask if we can infer anything about the properties of the graph by computing some linear algebraic invariants of the adjacency matrix. Indeed, algebraic graph theory and spectral graph theory are active areas of research which have successfully pursued this agenda. This equivalence bears fruit the other way too: Posing linear algebraic invariants as counting problems on weighted graphs yields a combinatorial way to interpret algebraic quantities like the Determinant and Permanent. We exploit computational aspects of this equivalence to yield memory-efficient algorithms for both computing linear algebraic invariants and for counting problems on graphs, all in the setting of bounded treewidth graphs in Section 1.3

In the next three sections, we delve in to these topics in detail.

1.1 Succinctly represented numbers

Complexity Theory studies how much resources (say time, space, randomness) are necessary and sufficient to solve a problem. Often, how the input is represented leads to dramatic changes in the complexity of a problem. Consider the following example: Given an integer

N, and a number $k \in \mathbb{N}$ (let n denote the length of the string required to specify N and k), compute N^k. If k is specified in unary (i.e., k is at most a polynomial in n), then it is easy to compute N^k (even in parallel). It lies in the complexity class TC^0 which is known to be contained in logarithmic space. However, if k is specified in binary, then N^k is a huge number with exponentially many bits (and could potentially take on a value that is doubly exponential in n) and hence needs exponential time to even write down completely. But it is quite natural to specify N and k in binary and hope for an algorithm to compute partial information about N^k – What is the value of $N^k \pmod{p}$ for a prime p? (known to be in polynomial time) What are the first 10 bits of N^k? (known to be in polynomial time for very special cases), What is the n^{100}-th bit of N^k? (not known to be in polynomial time) It turns out that such questions lie at the heart of many modern numerical analysis algorithms. Whereas the theory of NP-completeness has been quite successful in addressing the hardness of combinatorial problems, problems that arise typically in numerical analysis don't seem to fit it in to this framework. The recent work of Allender et al.[ABKPM09] addressed precisely this issue.

Given an arithmetic circuit succinctly encoding a large number, we want to test if the number is positive or more generally, find any bit in the binary representation of the number. A systematic study of these two problems was initiated by Allender et al.[ABKPM09], where the following two problems were formulated and shown to lie in the Counting Hierarchy[2]. :

$$\begin{aligned} \mathsf{PosSLP} &= \{\langle C\rangle : C \text{ is an arithmetic circuit computing a positive number}\} \\ \mathsf{BitSLP} &= \{\langle C, i, b\rangle : \text{The } i\text{-th bit of number computed by } C \text{ is } b\} \end{aligned}$$

Theorem 1.1 ([ABKPM09]).

1. $\mathsf{PosSLP} \in \mathsf{PH}^{\mathsf{CH}_2}$.
2. $\mathsf{BitSLP} \in \mathsf{PH}^{\mathsf{CH}_4}$.
3. BitSLP *is* $\#\mathsf{P}$*-hard.*

The results in Theorem 1.1 are achieved by analyzing the "majority-depth" (a notion first studied by Maciel and Thérien[MT98]) of threshold circuits for division and then invoking a simple translation lemma that states that a language is decided by uniform threshold circuit families of majority depth d if and only if the padded version of the language lies in the

[2] PP is the class of languages that are decidable by nondeterministic Turning machines that run in polynomial time and in which greater than half fraction of the computation paths accept. The Counting Hierarchy(CH) originally introduced by Wagner [Wag86] to study the complexity of counting problems on succinctly represented graphs, is defined similar to the Polynomial Hierarchy (PH) where the base class in PP instead of NP, i.e., $\mathsf{CH}_0 = \mathsf{P}$, $\mathsf{CH}_1 = \mathsf{PP}$ and $\mathsf{CH}_{k+1} = \mathsf{PP}^{\mathsf{CH}_k}$

Problem	Nonuniform Majority-Depth [MT98]	Uniform Majority-Depth [ABD14]
Iterated multiplication	3	3
Division	2	3
Powering	2	3
CRR-to-binary	1	3
Matrix powering	O(1) [MP00, HAB02]	3

TABLE 1.1: Comparison of Majority-depth bounds

Counting Hierarchy. We make this connection between TC^0 and Counting Hierarchy more precise in Chapter 3.

1.1.1 Contributions of this thesis

Here we report work that first appeared in [ABD14]. We study the complexity of various numerical problems like Division, Integer Powering, Matrix Powering and present improved construction of threshold circuits where the improvement is in terms of majority-depth in the sense of Maciel and Thérien. However, in contrast to the work of [MT98], our circuit families are *uniform*. This allows us to exploit the connection between threshold circuits and Counting Hierarchy to yield upper bounds in the Counting Hierarchy for the succinct versions of these problems.

Table 1.1 compares the complexity bounds that Maciel and Thérien obtained in the *nonuniform* setting with the bounds that we are able to obtain in the uniform setting. (Maciel and Thérien also considered several problems for which they gave uniform circuit bounds; the problems listed in Table 1.1 were not known to lie in dlogtime-uniform TC^0 until the subsequent work of [HAB02].) All previously-known dlogtime-uniform TC^0 algorithms for these problems rely on the CRR-to-binary algorithm of [HAB02], and thus have at *least* majority-depth 4 (as analyzed by [AS05]); no other depth analysis beyond $O(1)$ was attempted.

All parallel algorithms for division reduce finding the i-th bit of X/Y to computing an iterated sum of powers of rational numbers and truncating this iterated sum after a specific number of summands which depends on i and the size of representation of X and Y. In effect, they compute approximations to X/Y. Our main technical innovation in this context is a new approximation to X/Y, which can be computed by threshold circuits of majority-depth 3.

For the Matrix Powering problem, we first reduce the problem to division of two univariate polynomials (first observed in [MP00]) and then adapt the interpolation method of [HV06] in majority depth three to yield the coefficients of the remainder polynomial.

The Sum-of-Square-Roots problem (SSR) is a classic question that arises in Computational Geometry: Given m distinct positive integers $a_1, a_2 \ldots, a_m$ and signs $\sigma_1, \sigma_2, \ldots, \sigma_m$, is

$$\sum_{i=1}^{m} \sigma_i \sqrt{a_i} > 0$$

SSR is a key sub-routine in many computational geometry problems. Yet it is not known whether SSR admits a Polynomial-time algorithm[3]. The Sum-of-Square-Roots problem reduces to PosSLP [ABKPM09], which in turn reduces to BitSLP. We show that SSR reduces (via $\mathsf{PH}^{\mathsf{PP}}$-Turing reductions) to the problem of obtaining a bit of (an entry of) a constant size matrix raised to n-bit powers.

We further study stronger variants of PosSLP and obtain the same $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$ upper bound for all of them. This provides more evidence that the current bounds for PosSLP are probably not optimal. Our stronger variants makes sense for BitSLP too and in this case, we obtain a $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$ upper bound. We also show that BitSLP is in $\mathsf{PH}^{\mathsf{PP}}$ conditioned on $\binom{2n}{n}$ having small arithmetic circuits. Such conditional results are similar in spirit to those explored by Burgisser [Bür09].

1.2 Skew Circuits of Small Width

A Boolean circuit is a directed acyclic graph where the leaf nodes are labeled by input variables (and their negations) and/or constants $\{0, 1\}$ and the intermediate nodes are labeled by $\{\wedge, \vee\}$ and there is a designated output node. Every node computes a Boolean function of its children in a natural way. The Boolean function computed by the circuit is the function computed with the output node.

The Boolean circuit complexity class NC^1 consists of Boolean functions computable by polynomial sized logarithmic depth circuits. Basic arithmetic operations like addition, multiplication and division are known to be in NC^1. All regular languages have uniform NC^1 families deciding them and there is a regular language which is NC^1-hard. Over the years, several useful characterizations of NC^1 have emerged: NC^1 contains exactly those regular languages that are characterized by having a monoid containing a non-solvable group. They are also equally expressive as Branching Programs of constant width.

A Deterministic Branching Program (BP) is a layered directed acyclic graph G with the following properties:

[3]An affirmative answer to this question (or just even an NP algorithm), among other things, would imply that the Euclidean Traveling Salesman Problem is NP-complete. Currently it is known to be NP-hard.

- There is a designated source vertex s in the first layer (of in-degree 0) and a sink vertex t (of out-degree 0) in the last layer.
- For every node in the DAG, either the node has out-degree 1, in which case the out edge is labeled 1, or it has out-degree 2, in which case the two out-going edges are labeled exactly by literals $x_i, \neg x_i \in X$ (where X is the set of input literals) respectively.

The branching program naturally computes a boolean function $f(X)$, where $f(X) = 1$ if and only if there is path from s to t in which each edge is labeled by a true literal or a constant 1 on input X. The *length* (*width*) of the BP is the length (respectively, width) of the underlying layered DAG.

Our interest in NC^1 is motivated by the celebrated result of Barrington [Bar89], that Branching Programs of width 5 are sufficient to capture NC^1 in its entirety.

Branching programs have been pivotal to our understanding of computation with limited resources. They were first defined in [Lee59] and formally studied by Masek in his thesis [Mas76]. Borodin et al.[BDFP86] proved that AC^0 is contained in the class of functions computed by bounded width branching programs and conjectured that Majority cannot be computed by them. In a surprising result, Barrington showed that in fact, width 5 branching programs can compute all of NC^1 and hence the Majority function.

After the strong lower bound results of [Raz87][Smo87] for AC^0, the question of proving lower bounds for NC^1 gained a lot of attention. However this has turned out to be a notorious open problem. The branching program characterization of NC^1 has provided an avenue to understand the power of classes that reside inside NC^1. Though proving lower bounds for width 5 branching programs is equivalent to proving lower bounds for NC^1, it is conceivable that proving lower bounds for width 4 branching programs is easier. In this regard, it is known [Bar85] that width 3 branching programs of a restricted type (permutation branching programs) require exponential size to compute the AND function. It is worthwhile to contrast this against the situation at width 5, where permutation branching programs are known to be as powerful as general branching programs of width 5 and hence NC^1 itself.

1.2.1 Contributions of this thesis

Here we report work that first appeared in [BKL15]. An AND gate in a Boolean circuit is called *skew* if all but one of its children are input variables. A Boolean circuit in which all the AND gates are skew is called a *skew circuit*.

It is known that bounded width branching programs can be equivalently thought of as bounded width skew circuits (see for example [Rag10]). Here, we take a closer look at this relationship. The folklore construction[4] converts a polynomial size branching program of width w into a polynomial size skew circuit of width w^2. We improve this construction and show that any bounded width branching program of width greater than or equal to 5 can be converted into an equivalent skew circuit of width 7. We also study the conversion of skew circuits into branching programs. Here, the known construction converts a skew circuit of width w into a branching program of width $w + 1$ [Rag10]. We improve this construction and prove that a polynomial size skew circuit of width w can be converted into a polynomial size branching program of width w. These results prove that width 7 skew circuits of polynomial size characterize NC^1.

These structural results allow us to examine the set of languages in NC^1 by varying the width of skew circuits between 1 and 7. We start by examining the power of width 2 skew circuits. We observe that they are not universal as they cannot compute parity of two bits.

We then study the power of width 3 skew circuits. Recall that a CNF (DNF) is an AND (OR) of ORs (ANDs) of variables, i.e. in a CNF the AND gate is (possibly) non-skew. We implement a CNF by a width 3 skew circuit. Formally, we prove that any k-CNF or any k-DNF of size s has width 3 skew circuits of length $O(sk)$. Given that any Boolean function on n variables has a CNF of exponential (in n) size, this also proves that width 3 skew circuits are universal.

We consider the problem of proving lower bound for width 3 skew circuits. A natural candidate is a function which has no polynomial sized CNF or DNF. It is known that Parity is one such function. We prove that Parity requires width 3 skew circuits of exponential size. We observe that Parity and Approximate Majority have respectively, linear and polynomial size width 4 skew circuits. This separates width 3 skew circuits from width 4 skew circuits.

1.3 Counting Problems on Bounded Treewidth Graphs

Treewidth of a graph is a measure of how close a graph is to a tree. Since many hard problems turn out to be easy on trees, intuitively they should be easy on tree-like graphs too. This turns out to be the case and many NP-complete graph problems become tractable when restricted to graphs of bounded tree-width. In an influential paper, Courcelle [Cou90] proved that any property of graphs expressible in Monadic Second Order MSO logic can be decided in linear time on bounded tree-width graphs. For example, Hamiltonicity is an MSO property and hence deciding if a bounded tree-width graph has a Hamiltonian cycle can be done in linear time. More recently Elberfeld, Jakoby, Tantau [EJT10] showed that in fact, MSO properties on bounded tree-width graphs can be decided in L.

[4]Replace each wire by an AND gate and each node by an OR gate.

1.3.1 Linear Algebra in Bounded Treewidth

The determinant is a fundamental algebraic invariant of a matrix. For an $n \times n$ matrix A the determinant is given by the expression $\mathrm{Det}(A) = \sum_{\sigma \in S_n} \mathrm{sign}(\sigma) \prod_{i \in [n]} a_{i,\sigma(i)}$ where S_n is the symmetric group on n elements, σ is a permutation from S_n and $\mathrm{sign}(\sigma)$ is the parity of the number of inversions in σ ($\mathrm{sign}(\sigma) = 1$ if the number of inversions in σ is even and 0 if it is odd). Even though the summation in the definition runs over $n!$ many terms, there are many efficient sequential [vzGG13] and parallel [Ber84] algorithms for computing the determinant.

Apart from the inherently algebraic methods to compute the determinant there are also combinatorial algorithms (see, for instance, Mahajan and Vinay [MV97]) which extend the definition of determinant as a signed sum of cycle covers in the weighted adjacency matrix of a graph. [MV97] are thus able to give another proof of the GapL-completeness of the determinant, a result first proved by Toda [Tod91].

Armed with this combinatorial interpretation of the determinant and faced with its GapL-hardness, one can ask if the determinant is any easier when the underlying matrix represents simpler classes of graphs. Datta, Kulkarni, Limaye, Mahajan [DKLM10] study the complexity of the determinant and permanent, when the underlying directed graph is planar and show that they are as hard as the general case - GapL and #P-hard, respectively. We revisit these questions in the context of bounded treewidth graphs.

1.3.1.1 Contributions of this thesis

Here we present work first reported in [BD15]. We work with $(n \times n)$ matrices with entries from $\mathbb{Q}$, unless stated otherwise. We show that the following can be computed/tested in L:

1. The Determinant of a bounded treewidth matrix. As a corollary we can also compute the coefficients of the characteristic polynomial of a matrix.
2. The inverse of a bounded treewidth matrix. As a corollary we get a Logspace algorithm to compute powers of a matrix (with rational entries) whose support is a bounded treewidth digraph.
3. Testing if a system of rational linear equations $Ax = b$ is feasible where A is (a not necessarily square) matrix whose support is the biadjacency matrix of an undirected bipartite graph of bounded treewidth.
4. The number of Spanning Trees in graphs of bounded treewidth.
5. The number of Euler tours in a bounded treewidth directed graph

We also show hardness results to complement the Logspace algorithms above:

1. Computing the determinant of a bounded treewidth matrix is L-hard which precludes further improvement in the Logspace upper bound.
2. Computing the iterated matrix multiplication of bounded treewidth matrices is GapL-hard which precludes attempts to extend the L-bound on powering matrices of bounded treewidth to iterated matrix multiplication.
3. Powering matrices are however L-hard which prevents attempts to further improve the L-bound on matrix powering.

At the core of the upper bound results is our algorithm to compute the determinant by writing down an MSO formula that evaluates to true on every valid cycle cover of the bounded treewidth graph underlying A. The crucial point being that the cycle covers are parameterized by the number of cycles in the cycle cover, a quantity closely related to the sign of the cycle covers. This makes it possible to invoke the cardinality version of Courcelle's theorem (for Logspace) due to [EJT10] to compute the determinant.

1.3.2 Counting Euler Tours

An Euler tour of a graph is a walk on the graph that traverses every edge in the graph exactly once. Given a graph, deciding if there is an Euler tour of the graph is quite simple. Indeed, the famous Königsberg bridge problem that founded graph theory is just a question of existence of Euler tours on these bridges. Euler settled in the negative and in the process gave a necessary and sufficient condition for a graph to be *Eulerian*(A connected graph is Eulerian if and only if all the vertices are of even degree). This gives a simple algorithm to check if a graph is Eulerian.

An equally natural question is to ask for the number of distinct Euler tours in a graph. For the case of directed graphs, the BEST theorem due to De Bruijn, Ehrenfest, Smith and Tutte gives an exact formula that gives the number of Euler tours in a directed graph [AEB87, TS41] which yields a polynomial time algorithm via a determinant computation.

For undirected graphs, no such closed form expression is known and the computational problem is #P-complete[BW05]. In fact, the problem is #P-complete even when restricted to 4-regular planar graphs [GŠ12].So exactly computing the number of Euler tours is not in polynomial time unless $\#P = P$. This is an interesting phenomenon in the complexity of counting problems where the decision problem is easy while the counting problem is #P-hard. The flagship example of this phenomenon is the Perfect Matching problem for which there are several efficient algorithms to decide if a graph has a perfect matching, whereas counting the number of perfect matchings in a graph is #P-hard. When faced with such hard problems, there are two methods that are traditionally pursued:

1. Settle for an *approximate* solution that can be computable in polynomial time [5]. This problem is wide open for counting Euler Tours.
2. Find restricted classes of graphs for which one can count exactly the number of Euler tours in polynomial time[6]. Chebolu, Cryan, Martin have given a polynomial time algorithm for counting Euler tours in undirected series-parallel graphs [CCM12].

1.3.2.1 Contributions of this thesis

Here, we report work that appeared in [BDG15]. We give a Polynomial time algorithm to count the number of Euler tours in bounded treewidth graphs.

Theorem 1.2. *Exactly counting the number of Hamiltonian Cycles (or Paths) on bounded clique-width graphs that are line graphs of bounded treewidth graphs is in Logspace-uniform* $\#\mathsf{SAC}^1$*. Consequently, #Euler Tours on bounded tree-width graphs is in Logspace-uniform* $\#\mathsf{SAC}^1$*.*

The same techniques also yield the following :

Theorem 1.3. *The following counts can be obtained in* $\#\mathsf{SAC}^1$ *for bounded clique-width graphs:*

1. *#Longest Paths/Cycles*
2. *#Cycle Covers*
3. *#Perfect Matchings (on bipartite graphs)*

Every Euler tour in a graph yields a Hamiltonian cycle in its line graph. Though this map is not bijective in general, we show that we can make it so by altering the input graph while keeping the treewidth constant. It is well known [GW07] that the line graphs of bounded tree-width graphs have bounded clique-width. We show how to obtain a bounded clique width decomposition for the line graph of a bounded tree-width graph in Logspace – We first obtain a bounded tree-width decomposition (via a Logspace version of Bodlaender's theorem

[5]For the Perfect Matching problem, a long line of work culminating in the beautiful algorithm due to Jerrum and Sinclair [JS89, SJ89] gives a Fully Polynomial Randomized Approximation Scheme (FPRAS). But these techniques have not yielded an FPRAS for counting Euler tours so far.

[6]Exactly counting Perfect Matchings in bounded treewidth graphs is in Logspace [EJT10] since Perfect Matching is MSO- expressible.

[EJT10]) and implement the polynomial time procedure to obtain a bounded clique-width decomposition given a bounded treewidth decomposition due to [GW07] in Logspace.

Finally, we adapt the sequential algorithm from [EGW01] to decide if a bounded clique-width graph has a Hamiltonian path to yield a parallel algorithm that counts the number of Hamiltonian cycles.

1.4 Organization of this thesis

The rest of this thesis is organized as follows: We introduce some preliminaries from circuit complexity, graph theory and logic to aid reading the rest of the thesis in Chapter 2. In Chapter 3, we give an improved upper bound for BitSLP and investigate some related problems around succinctly represented mathematical objects. In Chapter 4, we study the bounded width skew circuits and present some upper and lower bounds and also tighten the connection between branching programs and skew circuits. In Chapter 5, we study the complexity of some linear algebraic invariants when restricted to bounded treewidth graphs. Chapter 6 deals with the problem of counting Euler tours on undirected bounded treewidth graphs. We conclude with Chapter 7 with some interesting problems and possible directions for further research around the topics investigated in this thesis.

Chapter 2

Preliminaries

In this chapter, we recall some basic notions from circuit complexity, graph theory and logic that will aid in reading the rest of the thesis. We start with some background on circuit complexity.

2.1 Background in Circuit Complexity

In this section, we describe some basics of circuit complexity. For a detailed treatment we refer the reader to [Vol99]. For definitions of standard complexity classes like Logspace, P, NP, etc., we refer the reader to Arora and Barak's book [AB09].

To fix some notation, let Σ^* denote the set of all strings over the alphabet Σ. We will mostly concern ourselves with $\Sigma = \{0, 1\}$. A language L is a subset of Σ^*. By L_n, we denote $\{x \in L : |x| = n\}$.

We say an algorithm A decides a language $L \subseteq \Sigma^*$ if for all $x \in L$, A on input x accepts, and for all $x \notin L$, A on input x rejects. A uniform model of computation is one where a single algorithm decides the language L for all input lengths.

In contrast, a non-uniform model of computation is one where we allow the algorithm designer the option of implementing (possibly) different algorithms to solve instances of different lengths. More formally, $A = \{A_n\}_{n\geqslant 1}$ is a family of algorithms that decides the language $L = \cup_{n\geqslant 1} L_n$, if A_n decides L_n. Now we introduce two common models of non-uniform computation: Boolean circuits and branching programs.

2.1.1 Boolean Circuits

A Boolean circuit is a directed acyclic graph (DAG) whose vertices (called gates) are labeled by constants $\{0, 1\}$ or variables from the set $X = \{x_n\}_{n \geqslant 1}$ at the leaves and Boolean functions $\{\mathrm{OR}, \mathrm{AND}, \mathrm{NOT}\}$[1] at the internal vertices and there is a designated output gate. The edges connecting these vertices are called wires. The number of wires entering (respectively exiting) a gate is called the fan-in (respectively fan-out) of the gate. Every gate computes an appropriate function (as given by its label) of its children naturally and the circuit is said to compute the Boolean function computed at the output gate. We have two measures of the efficiency of a Boolean circuit construction: the size of the circuit which is the number of wires in the circuit, and the depth of the circuit which is the length of the longest path from the input gates to the output gate.

We say a language L is decided by a Boolean circuit family $\{C_n\}_{n \geqslant 1}$ if for every n and for every $x \in L_n$, $C_n(x) = 1$. We denote by $\mathsf{SizeDepth}_n(s(n), d(n), \mathcal{B})$ the set of all languages decidable by circuit families of size $s(n)$ and depth $d(n)$ on inputs of length n, where the gates are functions from a fixed finite set $\mathcal{B}$ (often called the *basis*[2]). We assume that basis is just $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ unless specified otherwise[3]. We assume both OR and AND gates have unbounded fan-in unless specified otherwise. Some well studied circuit complexity class hierarchies:

$$\begin{aligned}
\mathsf{NC}^i &= \mathsf{SizeDepth}_n(n^{O(1)}, \log^i n) \text{ (where each gate has fan-in 2)} \\
\mathsf{SAC}^i &= \mathsf{SizeDepth}_n(n^{O(1)}, \log^i n) \text{ (where only OR gates can have unbounded fan-in)} \\
\mathsf{AC}^i &= \mathsf{SizeDepth}_n(n^{O(1)}, \log^i n) \\
\mathsf{ACC}^i &= \mathsf{SizeDepth}_n(n^{O(1)}, \log^i n, \{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}, \mathrm{MOD}_m\}) \\
\mathsf{TC}^i &= \mathsf{SizeDepth}_n(n^{O(1)}, \log^i n, \{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}, \mathrm{MAJ}\})
\end{aligned}$$

It is known that

$$\mathsf{NC}^i \subseteq \mathsf{SAC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{ACC}^i \subseteq \mathsf{TC}^i$$

In particular it is known that

$$\mathsf{NC}^0 \subsetneq \mathsf{SAC}^0 \subsetneq \mathsf{AC}^0 \subsetneq \mathsf{ACC}^0 \subsetneq \mathsf{TC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{SAC}^1 \subseteq \mathsf{AC}^1 \ldots$$

[1]Again, $\mathrm{OR}, \mathrm{AND}$ here are families of Boolean functions $f : \{0, 1\}^* \to \{0, 1\}$. We mildly abuse notation and drop the subscript denoting the domain of the function, whenever it is clear from the context which specific Boolean function $\mathrm{OR}_n, \mathrm{AND}_n : \{0, 1\}^n \to \{0, 1\}$ we are referring to.

[2]Two non-standard functions that we augment to the $\{\mathrm{OR}, \mathrm{AND}, \mathrm{NOT}\}$ basis are $\mathrm{MOD}_m, \mathrm{MAJ}$. They are defined as follows: $\mathrm{MOD}_m(x) = 1$ if and only if the number of 1's in x is a multiple of m; $\mathrm{MAJ}(x) = 1$ if and only if the number of 1's in x is greater than or equal to the number of 0's in x

[3]For SAC^i we also have the requirement that the NOT gates occur only at the leaves

In Chapter 3, since our primary aim is to understand the complexity of problems in the counting hierarchy, it is much more relevant to consider the notion of *majority depth* that was considered by Maciel and Thérien [MT98]. In this model, circuits have unbounded-fan-in AND, OR, and MAJORITY gates (as well as NOT gates). The class $\widehat{\mathsf{TC}}^0_d$ consists of functions computable by families of threshold circuits of polynomial size and constant depth such that no path from an input to an output gate encounters more than d MAJORITY gates. Thus the class of functions with majority depth zero, $\widehat{\mathsf{TC}}^0_0$, is precisely AC^0. Henceforth, we will use $\widehat{\mathsf{TC}}^0_d$ to refer to threshold circuits of majority-depth d and reserve TC^0 for threshold circuits in general without reference to the majority-depth

2.1.1.1 Uniform Circuit Classes

One way to compare the power of combinatorial non-uniform models like Boolean circuits and uniform models like Turing machines is to impose the condition that there is an algorithm that generates a description of the circuit family given n as an input – this way an infinite circuit family has a finite description and this allows us to study how the hierarchy of circuit classes interleave with uniform complexity classes defined via Turing machines like Logspace and allow us to find circuit characterizations of these uniform complexity classes. As a result, the general rule of thumb for circuit classes generated uniformly is that the uniformity machine should not be more powerful than than the circuit class it generates.

For the purposes of this thesis, all our upper bounds defined via circuit classes refer to $\mathsf{DLOGTIME}$-uniform or L-uniform versions of these classes. A language is said to be decidable in $\mathsf{DLOGTIME}$-uniform $\mathcal{C}$ if there is a circuit family belonging to the class $\mathcal{C}$ that decides it and furthermore, there exists a labeling of the circuit (each label of length at most $O(\log s(n))$, where $s(n)$ is the size of the circuit) that is computable in *linear* (in the size of the labels) time. Similarly, L-uniform circuit families have descriptions that can be computed in Logspace.

For the results in Chapter 3, we require that the TC^0 families are $\mathsf{DLOGTIME}$-uniform as elucidated in Proposition 3.14 which gives a crcucial connection between the Counting Hierarchy and Threshold circuits.

2.1.2 Branching Programs

We now turn our attention to another well-studied non-uniform model of computation, namely branching programs.

Firstly, to fix some terminology, call a directed acyclic graph $G = (V, E)$ layered if the vertex set of the graph can be partitioned, $V = V_1 \uplus \ldots \uplus V_\ell$ in such a way that for each edge

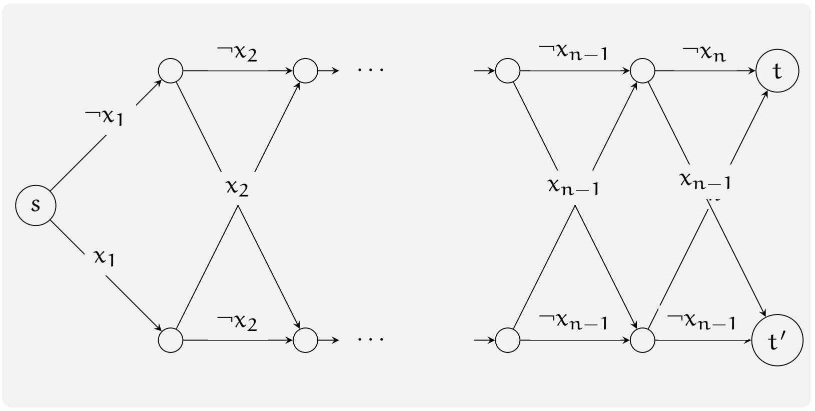

FIGURE 2.1: A width 2 BP computing Parity of n bits

$e = (u, v)$ there exists $1 \leqslant i < \ell$ such that $u \in V_i$ and $v \in V_{i+1}$. Given a layered graph G the *length* of the graph is the number of layers in it and the *width* of the graph is the maximum over $i \in [\ell]$, $|V_i|$.

Definition 2.1. (Branching Programs) A Deterministic Branching Program (BP) is a layered directed acyclic graph G with the following properties:

- There is a designated source vertex s in the first layer (of in-degree 0) and a sink vertex t (of out-degree 0) in the last layer.
- For every node in the DAG, either the node has out-degree 1, in which case the out edge is labeled 1, or it has out-degree 2, in which case the two out-going edges are labeled exactly by literals $x_i, \neg x_i \in X$ (where X is the set of input literals) respectively.

The branching program naturally computes a boolean function $f(X)$, where $f(X) = 1$ if and only if there is path from s to t in which each edge is labelled by a true literal or a constant 1 on input X. The *length* (*width*) of the BP is the length (respectively, width) of the underlying layered DAG.

We will denote the class of languages accepted by width-w BP by BP^w. Barrington [Bar85][Bar89] defined a restricted notion of branching programs called the Permutation Branching Program (PBP):

Definition 2.2. (Permutation Branching Programs as a graph) A width-w PBP is a layered width w BP in which the following conditions hold:

- There are designated source vertices $s_1, s_2, \ldots, s_w$ in the first layer, say layer 1 (of in-degree 0) and sink vertices $t_1, t_2, \ldots, t_w$ (of out-degree 0) in the last layer, layer ℓ.
- Each layer has exactly w vertices.

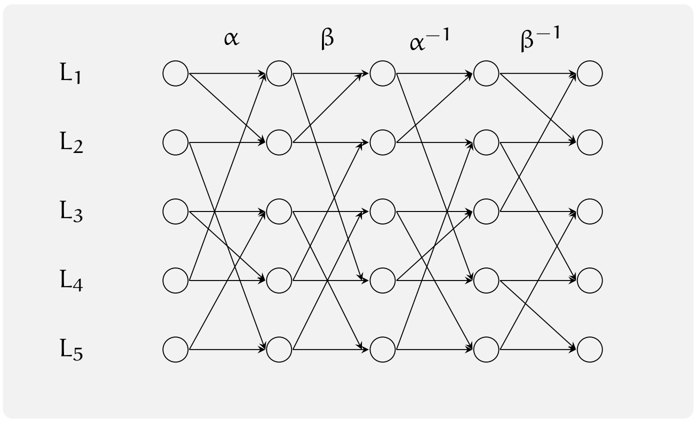

FIGURE 2.2: The commutator construction for AND in Barrington's Theorem

- In each layer $1 \leqslant i < \ell$, all the edges are labelled by a unique variable, say x_{j_i}.
- In each layer $1 \leqslant i \leqslant \ell$ and $b \in \{0, 1\}$, the edges activated when $x_{j_i} = b$ forms a permutation/matching, say $\theta_{i,b}$.

The permutation branching program naturally computes a boolean function $f(X)$, where $f(X) = 1$ if and only if there is path from s_1 to t_1, s_2 to t_2, and so on till s_w to t_w, where in each path each edge is labelled by a true literal or a constant 1 on input X. We will refer to the class of languages accepted by polynomial sized width-w PBP by PBP^w.

The above definition of PBP can be rephrased as follows:

Definition 2.3. (Permutation Branching Programs as a set of instructions) A width-w length-ℓ PBP is a program given by a set of ℓ instructions in which for any $1 \leqslant i \leqslant \ell$, the i-th instruction is a three tuple $\langle j_i, \theta^i, \sigma^i \rangle$, where j_i is an index from $\{1, 2, \ldots, |X|\}$, θ^i, σ^i are permutations of $\{1, 2, \ldots, w\}$. The output of the instruction is θ^i if $x_{j_i} = 1$ and it is σ^i if $x_{j_i} = 0$. The output of the program on input x is the product of the output of each instruction of the program on x.

We say that a permutation branching program computes a function f if there exists a fixed permutation $\pi \neq \mathrm{id}$ such that for every x such that $f(x) = 1$ the program outputs π and for every x such that $f(x) = 0$ the program outputs id[4]. We then say that the PBP *π-recognizes* f.

2.1.2.1 Skew Circuits

Closely related to branching programs are *skew* circuits, which form the main object of interest in Chapter 4.

[4]This is often called the strong acceptance condiction. Other notions of acceptance have been studied in the literature. See for example [Bro05]

Definition 2.4. (Skew Circuits) An AND gate is called *skew* if all but one of its children are input variables. A Boolean circuit in which all the AND gates are skew is called a *skew circuit*.

We assume that the skew circuits are layered. The width of the circuit is the maximum number of gates in any layer. The layer may have AND, OR or input gates. Each type of gate contributes towards the width. We assume that the fan-in of the AND gates is bounded by 2 and there are no NOT gates (negations appear only for the input variables)[5]. We denote the class of languages decided by width-w skew circuits by SK^w. The following lemma summarises some well known connections between BP^w, PBP^w, SK^w.

Lemma 2.5. *Let $w \in \mathbb{N}$. Then*

1. *For any w, $\mathsf{PBP}^w \subseteq \mathsf{BP}^w$.*
2. *For any $w \geqslant 5$, $\mathsf{BP}^w \subseteq \mathsf{PBP}^w$ [Bar89]*
3. *For any w, BP^w is contained in SK^{w^2} (see e.g. [Rag10])*

2.1.2.2 Barrington's Theorem: A brief overview

Here we briefly sketch the proof of item 2 in Lemma 2.5 due to Barrington. Firstly, notice that it is sufficient to simulate NC^1 circuits by PBP^5 in order to get the desired result. This is because given any bounded width branching program, we can construct a NC^1 circuit simulating it by a simple divide and conquer approach: To check there is a $s-t$ path in the branching program on input x, check if there is a node v in the middle layer of the branching program such that there is path from s to v and v to t. Since there are constantly many choices for v, this gives a bounded fan-in circuit, whose depth is at most $O(\log s(n))$ (where $s(n)$ is the length of the branching program) since the recursion bottoms out in that many steps.

To simulate an NC^1 circuit by a PBP^5 program, first convert the NC^1 circuit to a form where there are only AND and NOT gates (the OR gates are eliminated using De Morgan's laws and this increases the size of the circuit only by a constant factor).

The idea behind the proof is to show that there is a 5-cycle $\sigma \in S_5$ such that if the NC^1 circuit C evaluates to 1, then the PBP^5 evaluates to σ, otherwise it evaluates to $id \in S_5$ (We then say the PBP^5 P, σ-recognizes the circuit C). Firstly, note that for 5-cycles $\sigma, \tau \in S_5$, if a PBP^5 of

[5]This assumption is not without loss of generality. However, we will see that when a branching program is converted into a skew circuits, exactly this type of skew circuits arise.

length ℓ σ-recognizes a circuit C, then there is a PBP^5 of same length that τ-recognizes C[6]. One crucial reason why width 5 PBPs suffice is because S_5 is a non-solvable group where there are two elements α, β which are conjugates to each other and to their commutator, i.e. $\alpha\beta\alpha^{-1}\beta^{-1}$.

Theorem 2.6. *Given a Boolean circuit C on n inputs, bounded fan-in and depth d, there is a PBP^5 P of length 4^d, and a permutation σ such that for all x of length n, if $C(x) = 1$, then $P(x) = \sigma$, and otherwise, $P(x) = id$.*

Sketch. We proceed by induction on the depth d of the circuit. When $d = 1$, we have just input variables, and they can we recognized by a PBP with instruction, $\langle i, \sigma, id\rangle$. For the induction hypothesis, assume that we can always get a PBP of length 4^d for a depth d circuit. From this to get a PBP for depth $d + 1$ circuits, there are two cases: either the top gate is a NOT gate to which a circuit computing a function f is fed. Let P_f be the corresponding PBP of length 4^d for the circuit computing f. The PBP $P_{\neg f}$ is obtained by shifting the last instruction of P_f by σ^{-1}.

For the case when the top gate is an AND gate, let f_1 and f_2 be the two subcircuits feeding in to the AND gate, and hence have PBP^5s P_{f_1} (resp. P_{f_2}) σ_1 (resp. σ_2) recognizing them by the induction hypothesis. It is easily verified that PBP $P_{f_1 \wedge f_2}$ is just the commutator $[\sigma_1 \sigma_2]$, i.e., $\sigma_1 \sigma_2 \sigma_1{}^{-1} \sigma_2{}^{-1}$. $\square$

2.1.3 Arithmetic circuits

Also relevant to us in this thesis will be arithmetic circuit complexity classes, which are defined exactly like Boolean circuit complexity classes except the basis is now $\{\times, +\}$ and the circuits now compute functions $f : \{0, 1\}^n \to \mathbb{Z}$.

Any Boolean circuit can be *arithmetized* to yield an arithmetic circuit, i.e., replace $\wedge$ by $\times$, $\vee$ by $+$ and $\neg x_i$ by $(1 - x_i)$. It turns out that such an arithmetization, gives an interesting counting complexity characterization. For a Boolean circuit class $\mathcal{C}$, define $\#\mathcal{C}$ to be the set of all functions f from $\{0, 1\}^n$ to nonnegative integers, such that $f(x)$ gives the number of accepting *proof trees* of the given Boolean circuit on the input x. *Proof trees* of a Boolean circuit on an input x are certificates of acceptance of x, i.e., they form the minimal substructure of the tree (obtained by unwinding the DAG underlying the circuit in to a tree) needed to verify that the Boolean circuit on input x outputs 1. They are defined inductively: For an OR gate, exactly one of its children is in the proof tree and for an AND gate all its children are

[6]This is because since σ and τ are 5-cycles, there is a δ such that $\tau = \delta\sigma\delta^{-1}$ and hence to convert from a σ-recognizing PBP^5 to a τ-recognizing PBP^5, one simply shifts every the permutation in every instruction $\langle i, \sigma id\rangle$ by $\langle i, \delta\sigma\delta^{-1}\rangle$

in the proof tree. On a given input x, a Boolean circuit can have exponentially many proof trees.

Hence for any Boolean circuit class of interest $\mathcal{C}$, one can define its "counting" version $\#\mathcal{C}$. Relationship between Boolean circuit classes and their arithmetized versions have been investigated extensively in the last two decades; we refer the reader to the excellent survey of Allender [All04]. Two such counting classes will be of interest to us in this thesis:

Definition 2.7 ($\#\mathsf{SAC}^1$, GapL)**.** $\#\mathsf{SAC}^1$ is the class of functions from $\{0,1\}^n$ to nonnegative integers computed by polynomial-size logarithmic-depth, semi unbounded arithmetic circuits[7], using $+$ (unbounded fan-in) and $\times$ gates (fan-in 2) and the constants 0 and 1.
GapL is the class of functions from $\{0,1\}^n \to \mathbb{Z}$ computed by polynomial-size skew arithmetic circuits[8] using bounded fan-in $+$ and $\times$ gates and the constants 0 and 1. Alternately, they are exactly the class of functions that can be expressed as a difference of accepting and rejecting computations of a Nondeterministic Logspace Turing machine.

We recall an important and useful (in our context, in Chapter 6) result from circuit complexity, without proof.

Proposition 2.8 ([AJMV98, Vin91])**.** *Any function $f : \{0,1\}^n \to R$, where R is a semi-ring, computed by arithmetic circuits of size s and degree d can be computed by semi-unbounded arithmetic circuits of size $\mathsf{poly}(s,d)$ and depth $O(\log d)$. In particular, all functions computed by polynomial sized circuits of polynomial degree are exactly those in $\#\mathsf{SAC}^1$.*

2.2 Background on Logic

We define some basic notions in logic, in particular Monadic Second Order Logic and recall (without proof) some specific results at the intersection of Complexity theory and Finite Model Theory that have been obtained in the last few years. We refer the interested reader to standard texts in Finite Model Theory [Lib04] and Parameterized Complexity [FG06] for more background in logic tailored to the type of applications that we often need in this thesis.

Definition 2.9 (Monadic Second Order Logic)**.** Let the variables $V = \{v_1, v_2, \ldots, v_n\}$ denote the vertices of a graph $G = (V, E)$. Let X, Y denote subsets[9] of V or E. Let $E(x, y)$ be the

[7] Note that such circuits have degree that is at most a polynomial in the number of input variables.

[8] A $\times$ gate in an arithmetic circuit is said to be skew if all but one of its children are input variables or constants. A circuit in which all $\times$ gates are skew is called a skew arithmetic circuit.

[9] The case when quantification over subset of edges is not allowed is referred to as MSO_1 which is known to be strictly less powerful than MSO_2, the case when edge set quantification is allowed. Throughtout our thesis, we will work with MSO_2 and hence we will just refer to it as MSO.

predicate that evaluates to 1 when there is an edge between x and y in G. A logical formula ϕ is called an MSO-formula if it can be constructed using the following:

- $v \in X$
- $v_1 = v_2$
- $E(v_1, v_2)$
- $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\neg\phi$
- $\exists x\phi, \forall x\phi$
- $\exists X\phi, \forall X\phi$

In addition, if the Gaifman graph[10] of a relation $R(x_1, \dots, x_n)$ is bounded treewidth, then we can use R in item (3) above. A property Π of graphs is MSO-definable, if it can be expressed as a MSO formula ϕ such that ϕ evaluates to TRUE on a graph G if and only if G has property Π. (See [FG06] for more background on MSO)

Definition 2.10 (Solution Histogram)**.** Given a graph $G = (V, E)$ and an MSO formula $\phi(X_1, \dots, X_d)$ in free variables $X_1, \dots, X_d$, where $X_i \subseteq V$(or E), the $(i_1, \dots, i_d)$-th entry of histogram(G, ϕ) gives the number of subsets $S_1, \dots, S_d$ such that $|S_j| = i_j$ for which $\phi(S_1, \dots, S_d)$ is true.

We need the following results from [EJT10]:

Theorem 2.11 (Logspace version of Bodlaender's theorem)**.** *For every $k \geqslant 1$, there is a Logspace machine that on input of any graph G of treewidth at most k outputs a width-k tree decomposition of G.*

Theorem 2.12 (Logspace version of Courcelle's theorem)**.** *For every $k \geqslant 1$ and every MSO-formula ϕ, there is a Logspace machine that on input of any logical structure $\mathcal{A}$ of treewidth at most k decides whether $A \vDash \phi$ holds.*

Theorem 2.13 (Cardinality version of Courcelle's theorem)**.** *Let $k \geqslant 1$ and let $\phi(X_1, \dots, X_d)$ be an MSO-formula on free variables $X_1, \dots, X_d$. Then there is a Logspace machine that on input of the tree decomposition of a graph G of treewidth at most k, MSO-formula ϕ and $(i_1, \dots, i_d)$, outputs the value of histogram(G, ϕ) at $|X_1| = i_1, \dots, |X_d| = i_d$.*

[10]The Gaifman graph (also called the *Primal Graph*) of a binary relation $R \subseteq A \times A$ is the graph whose nodes are elements of A and an edge joins a pair of variables x, y if $(x, y) \in R$.

2.3 Background on Graph Theory

Definition 2.14 (Line Graph)**.** For an undirected graph $G = (V, E)$, the line graph of G denoted $L(G) = (L_V, L_E)$ is the graph where $L_V = E$ and $(e_i, e_j) \in L_E$ if and only if there exists a vertex $v \in V$ such that both e_i and e_j are incident on v.

The notion of treewidth was independently discovered by Halin [Hal76], Robertson and Seymour [RS84] in the late 70's/early 80's. It is a measure of how close a graph is to a tree, for an appropriate notion of closeness. Many NP-hard problems are solvable efficiently on trees, and the intuitively should be easy on "tree-like" graphs too. To make this more precise, we have

Definition 2.15 (Treewidth)**.** Given an undirected graph $G = (V_G, E_G)$ a tree decomposition of G is a tree $T = (V_T, E_T)$(the vertices in $V_T \subseteq 2^{V_G}$ are called *bags*), such that

1. Every vertex $v \in V_G$ is present in at least one bag, i.e., $\cup_{X \in V_T} X = V_G$.
2. If $v \in V_G$ is present in bags $X_i, X_j \in V_T$, then v is present in every bag X_k in the unique path between X_i and X_j in the tree T.
3. For every edge $(u, v) \in E_G$, there is a bag $X_r \in V_T$ such that $u, v \in X_r$.

The width of a tree decomposition is the $\max_{X \in V_T}(|X| - 1)$. The treewidth of a graph is the minimum width over all possible tree decomposition of the graph.

It is worth noting that a tree has treewidth 1, series-parallel graphs have treewidth 2 and a clique on n-vertices has treewidth $n - 1$.

Notice that graphs of bounded treewidth are always sparse graphs, i.e., a n-vertex graph of treewidth k has at most $O(kn)$ edges. But treewidth somehow doesn't capture the fact that there could also be dense graphs which have a "nice" structure, and hence nice properties. For example, the complete graph on n-vertices is also an "easy" graph in the sense that many properties which are hard to check in general graphs (say, for example Hamiltonicity) are easily checked on K_n. In this regard, it will be nice to have a measure which is low for K_n and the measure being bounded gives a measure of how close the graph is to a clique or a union of cliques, the same way treewidth measures how close a graph is to a tree. The relevant measure in this case is called *clique-width* and was introduced by Courcelle and Olariu [CO00]:

Definition 2.16 (Clique Width)**.** Let k be a positive integer. The class CW_k of labeled graphs is recursively defined as follows:

1. The single vertex graph $\bullet_a$ for $a \in [k]$ is in CW_k.

2. Let $G = (V_G, E_G, lab) \in CW_k$ and $H = (V_H, E_H, lab) \in CW_k$ be two vertex-disjoint labeled graphs. Then $G \oplus H = (V', E', lab') \in CW_k$, where $V' = V_G \cup V_H$ and $E' = E_G \cup E_H$ and for all $u \in V'$

$$lab'(u) = \begin{cases} lab_G(u), & \text{if } u \in V_G \\ lab_H(u), & \text{if } u \in V_H \end{cases}$$

3. Let a, b be distinct positive integers and $G = (V_G, E_G, lab) \in CW_k$ be a labeled graph. Then,

   (a) $\rho_{a \to b}(G) := (V_G, E_G, lab') \in CW_k$ where for all $u \in V'$

$$lab'(u) = \begin{cases} lab_G(u), & \text{if } lab_G(u) \neq a \\ b, & \text{if } lab_G(u) = a \end{cases}$$

   (b) $\eta_{a,b}(G) := (V_G, E', lab_G) \in CW_k$ where,

$$E' = E_G \cup \{(u, v) | u, v \in V_G, lab(u) = a, lab(v) = b\}$$

The clique-width of a labeled graph G is the least integer k such that $G \in CW_k$. The clique-width of an unlabeled graph G is the least integer k such that for some labeling function $L : V(G) \to [k]$, the labeled graph (G, L) is in CW_k. An expression X built with $\bullet_a, \oplus, \rho_{a \to b}, \eta_{a,b}$ for integers $a, b \in [k]$ is called a clique-width k expression. By $val(X)$, we denote the graph defined by expression X.

Note that any clique has clique width 2. It is easy to verify that the following clique width expression gives K_4 (See also Figure 2.3):

$$\eta_{1,2}(\rho_{2 \to 1}(\eta_{1,2}(\rho_{2 \to 1}(\eta_{1,2}(\bullet_1 \oplus \bullet_2)) \oplus \bullet_2)) \oplus \bullet_2)$$

Iterating the block of operations (in Figure 2.3) $n/4$ times gives K_n, while using only 2 labels.

A nice theorem due to Gurski and Wanke [GW07] relates the three defintions above: a set of graphs has bounded treewidth if and only if the corresponding set of line graphs has bounded clique width. We use this link crucially to develop an algorithm for counting Euler tours in bounded treewidth graphs.

Another graph parameter which is closely related to clique width is *NLC-width*. NLC stands for *Node Label Controlled* and has its origins in graph grammars; It was originally introduced

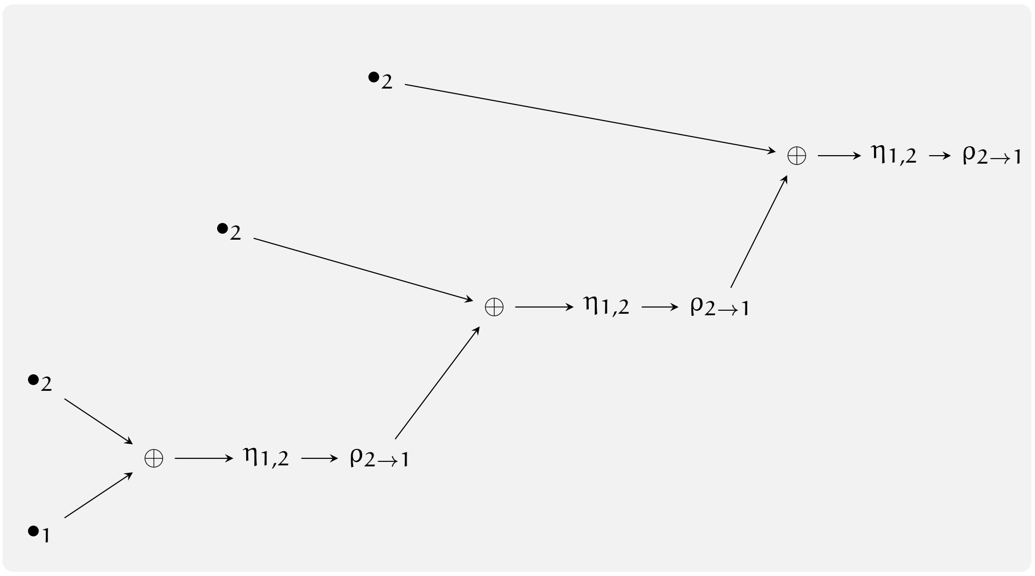

Figure 2.3: Parse tree for K_4

by Wanke [Wan94] who gave polynomial time algorithms for some NP-complete problems on graphs of bounded NLC-width. The NLC-width of a graph is defined via a grammar just like clique width, the only difference being that in the NLC grammar, the addition of edges (the $\eta_{i,j}$ operation in clique width grammar) and the disjoint union ($\oplus$ in the clique width grammar) are now combined in to one operation (namely $\times_S$). Notice that if S is empty, then we get the usual disjoint union ($\oplus$) operation. More formally, we have

Definition 2.17 (NLC-width). Let k be a positive integer. The class NLC_k of labeled graphs is recursively defined as follows:

1. The single vertex graph $\bullet_a$ for $a \in [k]$ is in NLC_k.
2. Let $G = (V_G, E_G, lab) \in \mathrm{NLC}_k$ and $H = (V_H, E_H, lab) \in \mathrm{NLC}_k$ be two vertex-disjoint labeled graphs and $S \subseteq [k]^2$, then $G \times_S H = (V', E', lab') \in \mathrm{NLC}_k$, where $V' = V_G \cup V_H$ and
$$E' = E_G \cup E_H \cup \{(u,v) | u \in V_G, v \in V_H, (lab_G(u), lab_H(v)) \in S\}$$
and for all $u \in V'$,
$$lab'(u) = \begin{cases} lab_G(u), & \text{if } u \in V_G \\ lab_H(u), & \text{if } u \in V_H \end{cases}$$
3. Let $G = (V_G, E_G, lab) \in \mathrm{NLC}_k$ and $R : [k] \to [k]$ be a function, then $\circ_R(G) := (V_G, E_G, lab')$ defined by $lab'(u) = R(lab(u))$ for all $u \in V_G$ is in NLC_k.

The NLC-width of a labeled graph G is the least integer k such that $G \in \mathrm{NLC}_k$. The NLC-width of an unlabeled graph G is the least integer k such that for some labeling function

$L : V(G) \to [k]$, the labeled graph (G, L) is in NLC_k. An expression Y built with $\bullet_a, \times_S, \circ_R$, for integers $a \in [k]$, $S \in [k]^2$ and $R : [k] \to [k]$ is called a NLC-width k expression. The graph defined by expression Y is denoted by $val(Y)$.

It is known that every graph of clique width at most k has NLC-width at most k and every graph of NLC-width at most k has clique width at most $2k$ [Joh98]. For a detailed treatment of the relationship between clique width and NLC-width and examples where the conversion between them is tight, we refer the reader to Johansson's Thesis [Joh01]. For our purposes, we can work with either of the two notions. We will choose one over the other in various occasions since it is sometimes much more comfortable to use NLC-width expressions instead of clique-width expressions and vice versa.

Definition 2.18 (Chordal graph, Chordal completion)**.** A graph is said to be chordal if every cycle with at least 4 vertices always contains a chord. A chordal completion of a graph G is a chordal graph with the same vertex set as G which contains all edges of G.

Definition 2.19 (Perfect Elimination Ordering, Elimination Tree)**.** Let $G = (V, E)$ be a graph and $o = (v_1, v_2, \ldots, v_n)$ be an ordering of the vertices of G. Let $N^-(G, o, i)$ and $N^+(G, o, i)$ for $i = 1, \ldots, n$ be the set of neighbors v_j of vertex v_i with $j < i$ and $j > i$ respectively.

$$\begin{aligned} N^-(G,o,i) &= \{v_j | (v_i, v_j) \in E \text{ and } j < i\} \\ N^+(G,o,i) &= \{v_j | (v_i, v_j) \in E \text{ and } j > i\} \end{aligned}$$

The vertex order o is said to be a Perfect Elimination Ordering (PEO) if for all $i \in [n]$, $N^+(G, o, i)$ induces a complete subgraph of G. The structure of G can then be characterized by a tree $T(G, o) = (V_T, E_T)$ defined as follows:

$$\begin{aligned} V_T &= V \\ E_T &= \{(v_i, v_j) \in E | i < j \text{ and } \forall j', i < j' < j, (v_i, v_{j'}) \notin E\} \end{aligned}$$

Such a $T(G, o)$ is called the Elimination Tree associated with the graph G.

It is known that a graph is chordal if and only if it has a Perfect Elimination Ordering [FG65].

Definition 2.20 (Cycle Cover)**.** A *cycle cover* $\mathcal{C}$ of $G = (V, E)$ is a set of vertex-disjoint cycles that cover the vertices of G. I.e., $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$, where $V(C_i) = \{c_{i_1}, \ldots, c_{i_r}\} \subseteq V$ such that (c_{i_1}, c_{i_2}), (c_{i_2}, c_{i_3}), ..., $(c_{i_{r-1}}, c_{i_r})$, $(c_{i_r}, c_{i_1}) \in E(C_i) \subseteq E$ and $\dot\cup_{i=1}^{k} V(C_i) = V$. The least numbered vertex $h_i \in V(C_i)$, is called the head of the cycle.

Fact 1. The weight of the cycle $C_i = \prod_{j\in[r]} wt(a_{ij})$ and the weight of the cycle cover $wt(\mathcal{C}) = \prod_{i\in[k]} wt(C_i)$. The sign of the cycle cover $\mathcal{C}$, $sign(\mathcal{C})$ is $(-1)^{n+k}$.

Every permutation $\sigma \in S_n$ can be written as a union of vertex disjoint cycles. Hence a permutation corresponds to a cycle cover of a graph on n vertices. In this light, the determinant of an $(n \times n)$ matrix A can be seen as a signed sum of cycle covers: $\det(A) = \sum_{\mathcal{C}} \text{sign}(\mathcal{C})\text{wt}(\mathcal{C})$

Fact 2 ([Ete95]). $\mathsf{ORD}(G, s, t)$ is the following problem: Given a directed path G and two distinguished vertices s and t, does s occur before t? ORD is L-complete.

2.4 Mathematical Preliminaries

Fact 3 (Kronecker substitution [CLO92]). Let $P(x_1, x_2, \dots, x_n)$ be a multivariate polynomial of degree d. We replace every occurence of variable x_i by x^{d^i}. This yields an unique univariate polynomial $Q(x)$ of degree at most $O(d^n)$ such that P can be efficiently recovered from the knowledge of coefficients of Q. When the number of variables is a constant, the degree of the multivariate polynomial and the univariate polynomial are polynomially related.

Fact 4 (Lagrange interpolation [vzGG13]). Let $p(x)$ be an univariate polynomial of degree at most d. Then

$$p(x) = \sum_{i=0}^{d} p(i) \prod_{j \neq i} \frac{x - j}{i - j}$$

where $0 \leqslant j \leqslant d$.

Given a list of primes $\Pi = (p_1, \dots, p_m)$ and a number X, the CRR_Π representation of X is the list $(X \bmod p_1, \dots, X \bmod p_m)$. We omit the subscript Π if context makes it clear.

We now state the Chinese Remainder Theorem, which is the main tool used in Chapter 3 to derive efficient TC^0 circuits for division:

Fact 5 (Chinese Remainder Theorem). The unique number $0 < X < M$ such that for every q ($q = p^e$ where $e \geqslant 1$ are relatively prime) $X \bmod q = X_q$ is given by the expression:

$$X = M \left(\sum_q \frac{X_q h_q}{q} - \left\lfloor \sum_q \frac{X_q h_q}{q} \right\rfloor \right)$$

where $M = \prod_q q$ and $h_q = (M/q)^{-1} \bmod q$.

Chapter 3

Succinctly represented numbers

How hard is it to compute the 10^{100}-th bit of the binary expansion of $\sqrt{2}$? Datta and Pratap [DP12], and Jeřábek [Jeř12] considered the question of computing the m-th bit of an algebraic number. Jeřábek [Jeř12] showed that this problem has uniform TC^0 circuits[1] of size polynomial in m (which is not so useful when $m = 10^{100}$). Earlier, Datta and Pratap showed a related result: when m is expressed in *binary*, this problem lies in the counting hierarchy. More precisely, Datta and Pratap showed that this problem is reducible to the problem of computing certain bits of the quotient of two numbers represented by arithmetic circuits of polynomial size.[2] Thus, we are led to the problem of evaluating arithmetic circuits. In this chapter, we focus on arithmetic circuits *without input variables*. Thus an arithmetic circuit is a (possibly very compact) representation of a number.

Arithmetic circuits of polynomial size can produce numbers that require exponentially-many bits to represent in binary (See for example Figure 3.1). The problem[3]known as BitSLP ($= \{(C, i, b) :$ the i-th bit of the number represented by arithmetic circuit C is $b\}$) is known to be hard for $\#\mathsf{P}$ [ABKPM09]. It was known that BitSLP lies in the counting hierarchy [ABKPM09], but the best previously-known bound for this problem is the bound mentioned in [ABKPM09] and credited there to [AS05]: $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}}$. That bound follows via a straightforward translation of a uniform TC^0 algorithm presented in [HAB02].

In this chapter, we improve this bound on the complexity of BitSLP to $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$. In order to do this, we present improved uniform TC^0 algorithms for a number of problems that were already known to reside in uniform TC^0.

[1]For somewhat-related TC^0 algorithms on sums of radicals, see [HBM+10].

[2]It is mistakenly claimed in [DP12] that this problem lies in $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$. In this thesis, we prove the weaker bound that it lies in $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$.

[3]"SLP" stands for "straight-line program"; which is a model equivalent to arithmetic circuits. Throughout the rest of the chapter, we will stick with the arithmetic circuit formalism.

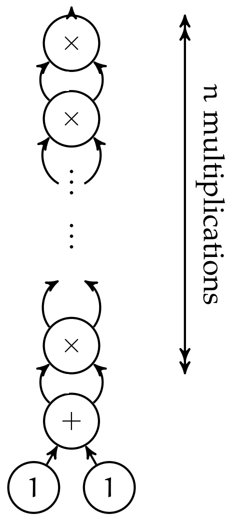

FIGURE 3.1: Arithmetic circuit with $n+3$ gates computing 2^{2^n}

As explained in Section 1.1 of Chapter 1, this is an interesting class of problems whose complexity is far from settled. We first give a quick overview of some basic definitions about representing a real number in binary, and review the circuit complexity of some basic arithmetic operations that are implementable in TC^0, with a specific emphasis on majority depth.

We need to refer (repeatedly) to the binary expansion of a rational number. Furthermore, we want to avoid possible confusion caused by the fact that some numbers have more than one binary expansion (e.g. $1 = \sum_{i=1}^{\infty} 2^{-i}$). Thus the following definition fixes a *unique* binary representation for every rational number.

Definition 3.1.

The *Binary expansion* of the rational number X/Y is the *unique* expression $X/Y = \sum_{i=-\infty}^{\infty} a_i 2^i$, where each $a_i \in \{0,1\}$, and where the binary expansion of any integer multiple of 2^j has $a_i = 0$ for all $i < j$.

The *binary expansion of* X/Y *correct to* m *places* is the sequence of bits representing $\sum_{i=-m}^{\lfloor \log(X/Y) \rfloor} a_i 2^i$.

The following lemma is a list of useful subroutines of problems that are computable in AC^0 and $\widehat{\mathsf{TC}}^0_1$.

Lemma 3.2. *Let* $x, y, i, j, k, x_j \in (0, n^c)$ *(*$c \geqslant 3$ *is a constant). Let* $X, X_j \in [0, 2^n)$ *and let* $p < n^c$ *be prime. Then the following operations have the indicated complexities:*

1. $p \mapsto$ *first* n^c *bits of* $1/p$ *in* $\widehat{\mathsf{TC}}^0_0 = \mathsf{AC}^0$.

2. $k, X_1 \ldots, X_k \mapsto \sum_{j=1}^{k} X_j \bmod p$ *in* $\widehat{\mathsf{TC}}_1^0$.
3. $x \mapsto x^i \bmod p$ *in* $\widehat{\mathsf{TC}}_0^0 = \mathsf{AC}^0$.
4. $p \mapsto g_p$ *in* $\widehat{\mathsf{TC}}_0^0 = \mathsf{AC}^0$ *where* g_p *is a generator of the multiplicative group modulo* p.
5. $X \mapsto X \bmod p$ *in* $\widehat{\mathsf{TC}}_1^0$.
6. $x, y \mapsto xy \bmod p$ *in* $\widehat{\mathsf{TC}}_0^0 = \mathsf{AC}^0$.
7. $(x_1, \ldots, x_k) \mapsto \prod_{j=1}^{k} x_j \bmod p$ *in* $\widehat{\mathsf{TC}}_1^0$.

Proof. We list the proofs of items in the Lemma above:

1. Follows from Lemma 4.2 and Corollary 6.2 in [HAB02].
2. Follows from Corollary 3.4.2 in [MT98].
3. Follows from Corollary 6.2 in [HAB02].
4. Follows from testing each integer $x \in [1, n-1]$ for being a generator by checking if $x^{(p-1)/2} \not\equiv 1 \bmod p$ and reporting the first successful x (implicit in [HAB02, ABKPM09]).
5. Follows from (the proof of) Lemma 4.1 in [HAB02].
6. Follows from Proposition 3.7 in [MT98] and the fact that two $\log n$-bit integers can be multiplied in AC^0.
7. Follows from the reduction of multiplication to addition of discrete logs and the previous parts.

□

3.1 Uniform Circuits for Division

In this section, we provide constructions of majority-depth-efficient TC^0 circuits for division. A number of efficient sequential algorithms were known for dividing two integers since antiquity. It is however entirely plausible that fast parallel algorithms for division had their genesis in computational complexity. We discuss some of them briefly in order to put in context our contribution to this line of work.

3.1.1 Parallel algorithms for division

In a landmark paper that studied the parallel complexity of division and related problems, Beame, Cook and Hoover [BCH86] show that integer division, integer powering and iterated iterated integer product are equivalent and have log-depth circuits deciding them, i.e, P-uniform NC^1circuits for division[4]. This was followed by constuctions [DMS94, Mac98, CDL01] each improving on the uniformity of the circuit families, until finally Hesse, Allender and Barrington [HAB02] gave DLOGTIME-uniform TC^0 circuits for division.

The basic structure of all TC^0 algorithms for division (reducing the problem to iterated product, and computing iterated product via a reduction to iterated addition, via conversion to and from Chinese Remainder Representation) has remained unchanged since the pioneering work of [BCH86]. Subsequent improvements have focused on finding more efficient implementations of these various tasks – the notion of efficiency has mostly focussed on the uniformity of the circuits constructed.

Broadly, the central idea of Beame et al.'s construction is as follows: If we can obtain X/Y correct to r places for any r (r specified in unary), then we can obtain the m-th bit of X/Y by computing $\lfloor \frac{X}{2^m Y} \rfloor - \lfloor \frac{X}{2^{m+1} Y} \rfloor$. So the focus is on computing X/Y correctly to some r places that will enable us to compute the m-th bit of X/Y. Let t be a number such that $2^{t-1} < Y \leqslant 2^t$. They take $Y' = 2^{-t}Y$ and use the identity $\frac{1}{1-Y'} = \sum_{j=0}^{\infty}(Y')^j$. Now it just remains to compute $2^{-t}X\frac{1}{1-(1-Y')} = 2^{-t}X\sum_{j=0}^{\infty}(1-Y')^j$ to appropriate accuracy. Now each term $2^{-t}X(1-Y')^j = \frac{X(2^t-Y)^j}{2^{(j+1)t}}$ can be computed[5] in parallel and summed up. So $\sum_{j=0}^{O(m)} X(2^t-Y)^j$ is computed in the CRR basis – the product $X(Y')^j$ can be done in the CRR basis as follows: For every prime p in the CRR basis, find $X(Y')^j \pmod p$. To this end, they first find a generator–say a–of the cyclic group $\mathbb{Z}_p^*$ and find i, k such that $X \equiv a^i \pmod p$ and $Y \equiv a^k \pmod p$. This reduces the iterated product $X(Y')^j \pmod p$ to iterated addition in the exponent of a. Now, to invoke the Chinese Remainder Theorem (See Fact 5), they just have to find h_p, M and $1/p$ to sufficient accuracy to reconstruct $\lfloor \frac{X}{2^m Y} \rfloor$ via yet another iterated addition.

The NC^1circuit implementation of Beame, Cook and Hoover however is P-uniform and this was further refined by Chiu, Davida and Litow [CDL01] to yield logspace-uniform NC^1 circuits for division. Notice however that actually both these implementations are actually TC^0 implementations and at the time of their discovery the emphasis was more on the uniformity of the circuit class. For our purposes, i.e., to upper bound the complexity of

[4]Actually their circuits are P-uniform TC^0 circuits, but their circuits predate TC^0 itself, which was first defined by Hajnal et al. [HMP+87]

[5]Note that since division by a power of 2 is easy, the primary concern here is the computation of the other part namely, $\sum_{j=0}^{O(m)} X(2^t-Y)^j$

BitSLP[6], it is absolutely necessary that the TC^0 circuits constructed are DLOGTIME-uniform, because only then can we invoke Proposition 3.14. (This is because the translation from threshold circuits of majority depth d to the d-th level of counting hierarchy works via padding, and a straight forward translation of the TC^0 circuits of [BCH86, CDL01] will yield a trivial upper bound of PSPACE.) The main breaktrough in this regard is the work on Hesse, Allender and Barrington who gave DLOGTIME-uniform TC^0 circuits for this problem and this is the work that is directly relevant to us. Allender et al. [ABKPM09] credit Allender and Schnorr [AS05] for the analysis of the majority depth of [HAB02], which yields the previous best bound on BitSLP, namely the 4th level of the counting hierarchy. We now give an overview of the work of Hesse et al. [HAB02]: We first give an outline of their construction and give and indicate why it has majority depth 4; we then discuss what we do differently in this thesis in order to improve the majority depth to 3.

Hesse et al. use the same basic idea as that of Beame, Cook and Hoover. However, they give DLOGTIME-uniform-TC^0 implementations of the various steps involved. The idea in [HAB02] is the following: Given X, Y we can obtain $X \pmod p$ and $Y^{-1} \pmod p$ for sufficiently many $O(\log n)$-bit p in $\widehat{\mathsf{TC}}_1^0$ (item 5 in Lemma 3.2). Now, to get the m-th bit of a number Z that is given to us in CRR, they compute $u = Z/2^m$ and $v = Z/2^{m+1}$, and note that the desired bit is $u - 2v$. They get this bit as a CRR number, but it is easy to recognize the CRR forms of the numbers 0 and 1. A nice idea due to [HAB02] here is to note that dividing Z by a power of 2, like 2^{m+1} is quite close to multiplying Z by $\prod_{i=1}^{m} \frac{(1+M_i)/2}{M_i}$. Here M_i's are obtained by partitioning the CRR basis suitably and taking product of all the primes in the i-th partition to yield M_i. Intuitively, the number of partitions created depends on the power of 2 that we would want to divide by. In our case, $Z = \sum_{j=0}^{m} X(Y')^j$ where $Y' = 1 - 2^{-t}Y$ for a sufficently scaled Y, and we compute the CRR of

$$Q = \left\lfloor \sum_{j=0}^{m} X(Y')^j \prod_{i=1}^{m} \frac{(1+M_i)/2}{M_i} \right\rfloor$$

which can be done in $\widehat{\mathsf{TC}}_2^0$ and this is either $\{\lfloor \frac{\sum_{j=0}^{m} X(Y')^j}{2^m} \rfloor, \lfloor \frac{\sum_{j=0}^{m} X(Y')^j}{2^m} \rfloor + 1\}$. They determine which of these is the case by checking $Q2^m > Z$ (in the CRR). This positivity check (essentially the verbose version of PosSLP) adds another $\widehat{\mathsf{TC}}_2^0$ and this overall yields a $\widehat{\mathsf{TC}}_4^0$ upper bound[7].

[6]Notice also that strictly speaking, the problem that is of direct interest to us is the complexity of converting a number given by its CRR representation to its binary representation (and not division). This is because, given an SLP X encoding a huge number, $X \pmod p$ for a prime p is computable in polynomial time. So to get a bit of the number represented by X it is sufficient to get a specific bit of X, given $X \pmod p$ for sufficiently many p. However, in this thesis we give TC^0 circuits for division which involves CRR to binary as a subroutine, and this yields the desired bound.

[7]Notice that they had a $\widehat{\mathsf{TC}}_1^0$ circuit to convert Z to $Z \pmod p$, and a $\widehat{\mathsf{TC}}_2^0$ circuitry for computing Q and one more $\widehat{\mathsf{TC}}_2^0$ circuitry on top of this to check positivity, and still got a circuit with majority depth 4 rather than 5 because one layer of $\widehat{\mathsf{TC}}_1^0$ in the positivity check can be done parallely with the $\widehat{\mathsf{TC}}_1^0$ to compute $Z \pmod p$

The main cost incurred (and which looks like it could be avoided) in [HAB02] is the last step (checking $Q2^m > Z$).

In contrast, our approach[8] is to simply use the reconstruction formula (See Fact 5) of the Chinese Remainder Theorem to reconstruct the bit we need. We arrange the different modules used (the basic circuitry for which is given by Lemma 3.2) in an efficient way. For example, we know that iterated addition and product of n^c-many $O(\log n)$-bit numbers can be done in majority-depth one via item 2 in Lemma 3.2, so whenever we can collapse two iterated sums/product in to one iterated sum/product, this saves on the majority depth provided the combined iterated sum/product has polynomially many short numbers.

The main idea in our approach to computing X/Y will be to compute $\widetilde{V}(X,Y)$, a strict underestimate of X/Y, such that $X/Y - \widetilde{V}(X,Y) < 1/Y$. Since $Y > 1$, we have that $\lfloor X/Y \rfloor \neq \lfloor (X+1)/Y \rfloor$ if and only if $(X+1)/Y = \lfloor X/Y \rfloor + 1$. It follows that in *all* cases $\lfloor X/Y \rfloor = \lfloor \widetilde{V}(X+1,Y) \rfloor$, since

$$\left\lfloor \frac{X}{Y} \right\rfloor \leqslant \frac{X}{Y} = \frac{X+1}{Y} - \frac{1}{Y} < \widetilde{V}(X+1,Y) < \frac{X+1}{Y}.$$

The approximation guarantee provided by $\widetilde{V}(X,Y)$ helps us to avoid the positivity check altogether. It is interesting to note that, in order to compute $\lfloor \frac{X}{Y} \rfloor$, we actually compute an approximation to $(X+1)/Y$.

3.1.2 Division in majority-depth three

We now give a detailed construction of our $\widehat{TC}^0_3$ circuits for division. More formally, we prove:

Theorem 3.3. *The function taking as input $X \in [0, 2^n), Y \in [1, 2^n)$, and 0^m and producing as output the binary expansion of X/Y correct to m places is in $\widehat{TC}^0_3$.*

Proof. This task is trivial if $Y = 1$; thus in the rest of this argument assume that $Y \geqslant 2$.

Computing the binary expansion of Z/Y correct to m places is equivalent to computing $\lfloor 2^m Z/Y \rfloor$. Thus we will focus on the task of computing $\lfloor X/Y \rfloor$, given integers X and Y.

Using AC^0 circuitry, we can compute a value t such that $2^{t-1} \leqslant Y < 2^t$.

[8]Notice that from the description above it looks like [HAB02] have completely avoided computing the quantities $M = \prod_q q$ and $h_q = (M/q)^{-1} \bmod q$ for primes q in the CRR basis – which is a majority-depth-intesive computation in the parallel algorithms for division – and somehow estimate the requisite bit from just the CRR representations of X and $(Y')^j$ and comparing them against the CRR of 0 and 1. This is not the case since a positivity check $Q2^m > Z$ involves computation of these quantities (at least with our current techniques we do not know how to avoid these computations).

Let $u = 1 - 2^{-t}Y$. Then $u \in (0, \frac{1}{2}]$. Thus, $Y^{-1} = 2^{-t}(1-u)^{-1} = 2^{-t}(1+u+u^2+\ldots)$. Set $Y' = 2^{-t}(1+u+u^2+\ldots+u^{2n+1})$, then

$$0 < Y^{-1} - Y' \leqslant 2^{-t} \sum_{j>2n+1} 2^{-j} < 2^{-(2n+1)}$$

Define $W(X,Y)$ to be XY'. Hence, $0 < \frac{X}{Y} - W(X,Y) < 2^{-(n+1)}$.

As explained previously, our approach to computing X/Y will be to compute $\widetilde{V}(X,Y)$, a strict underestimate of X/Y, such that $X/Y - \widetilde{V}(X,Y) < 1/Y$. Since $Y > 1$, we have that $\lfloor X/Y \rfloor \neq \lfloor (X+1)/Y \rfloor$ if and only if $(X+1)/Y = \lfloor X/Y \rfloor + 1$. It follows that in *all* cases $\lfloor X/Y \rfloor = \lfloor \widetilde{V}(X+1,Y) \rfloor$, since

$$\left\lfloor \frac{X}{Y} \right\rfloor \leqslant \frac{X}{Y} = \frac{X+1}{Y} - \frac{1}{Y} < \widetilde{V}(X+1,Y) < \frac{X+1}{Y}.$$

Note that, in order to compute $\lfloor \frac{X}{Y} \rfloor$, we actually compute an approximation to $(X+1)/Y$.

The approximation $\widetilde{V}(X,Y)$ is actually defined in terms of another rational approximation $W(X,Y)$, which will have the property that $\widetilde{V}(X,Y) \leqslant W(X,Y) < X/Y$. $W(X,Y)$ will be an under approximation of $\frac{X}{Y}$ with error at most $2^{-(n+1)}$. We find it useful to use this equivalent expression for $W(X,Y)$:

$$W(X,Y) = \frac{X}{2^t} \sum_{j=0}^{2n+1} (1 - \frac{Y}{2^t})^j = \frac{1}{2^{2(n+1)t}} \sum_{j=0}^{2n+1} X(2^t - Y)^j 2^{(2n+1-j)t}.$$

Define $W_j(X,Y)$ to be $X(2^t - Y)^j(2^{(2n+1-j)t})$. Thus $W(X,Y) = \frac{1}{2^{2(n+1)t}} \sum_{j=0}^{2n+1} W_j(X,Y)$.

Lemma 3.4. *(Adapted from [DP12]) Let* Π *be any set of primes such that the product* M *of these primes lies in* $(2^{n^c}, 2^{n^d})$ *for some* $d > c \geqslant 3$*. Then, given* X, Y, Π *we can compute the* CRR$_\Pi$ *representations of the* $2(n+1)$ *numbers* $W_j(X,Y)$ *(for* $j \in \{0, \ldots, 2n+1\}$*) in* $\widehat{\mathsf{TC}}^0_1$*.*

Proof. With the aid of Lemma 3.2, we see that using AC^0 circuitry, we can compute $2^t - Y$, $2^j \bmod p$ for each prime $p \in \Pi$ and various powers j, as well as finding generators mod p. In $\widehat{\mathsf{TC}}^0_1$ we can compute $X \bmod p$ and $(2^t - Y) \bmod p$ (each of which has $O(\log n)$ bits). Using those results, with AC^0 circuitry we can compute the powers $(2^t - Y)^j \bmod p$ and then do additional arithmetic on numbers of $O(\log n)$ bits to obtain the product $X(2^t - Y)^j(2^{(2n+1-j)t}) \bmod p$ for each $p \in \Pi$. (The condition that $c \geqslant 3$ ensures that the numbers that we are representing are all less than M.) □

Having the CRR$_\Pi$ representation of the number $W_j(X,Y)$, our goal might be to convert the $W_j(X,Y)$ to binary, and take their sum. In order to do this efficiently, we first show how to obtain an approximation (in binary) to $W(X,Y)/M$ where $M = \prod_{p \in \Pi} p$, and then in Lemma 3.9 we build on this to compute our approximation $\widetilde{V}(X,Y)$ to $W(X,Y)$.

Recall that $W(X,Y) = \frac{1}{2^{2(n+1)t}} \sum_{j=0}^{2n+1} W_j(X,Y)$. Thus $2^{2(n+1)t} W(X,Y)$ is an integer with the same significant bits as $W(X,Y)$.

Lemma 3.5. *Let Π be any set of primes such that the product M of these primes lies in $(2^{n^c}, 2^{n^d})$ for a fixed constant $d > c \geqslant 3$, and let b be any natural number. Then, given X, Y, Π we can compute the binary representation of a* good *approximation to $\frac{2^{2(n+1)t} W(X,Y)}{M}$ in $\widehat{\mathsf{TC}}_2^0$ (where by good we mean that it under-estimates the correct value by at most an additive term of $1/2^{n^b}$).*

Proof. Let $h_p^\Pi = (M/p)^{-1} \bmod p$ for each prime $p \in \Pi$.

If we were to first compute a good approximation $\tilde{A}_\Pi$ to the fractional part of:

$$A_\Pi = \sum_{p \in \Pi} \frac{(2^{2(n+1)t} W(X,Y) \bmod p) h_p^\Pi}{p}$$

i.e. if $\tilde{A}_\Pi$ were a good approximation to $A_\Pi - \lfloor A_\Pi \rfloor$, then $\tilde{A_\Pi} M$ would be a good approximation to $2^{2(n+1)t} W(X,Y)$. This follows from observing that the fractional part of A_Π is exactly $\frac{2^{2(n+1)t} W(X,Y)}{M}$ (as in [HAB02, ABKPM09]).

Instead, we will compute a good approximation $\widetilde{A'}_\Pi$ to the fractional part of

$$A'_\Pi = \sum_{p \in \Pi} \sum_{j=0}^{2n+1} \frac{(W_j(X,Y) \bmod p) h_p^\Pi}{p}.$$

Note that the exact magnitudes of the two quantities A_Π, A'_Π are not the same but their *fractional parts* will be the same. Since we are adding up $2(n+1)|\Pi|$ approximate quantities it suffices to compute each of them to $b_m = 2n^b + 2(n+1)|\Pi|$ bits of accuracy to ensure:

$$0 \leqslant \frac{W(X,Y)}{M} - \widetilde{A'}_\Pi < \frac{1}{2^{n^b}}.$$

Now we analyze the complexity. By Lemma 3.4, we obtain in $\widehat{\mathsf{TC}}_1^0$ the CRR_Π representation of $W_j(X,Y) \in [0, 2^n)$ for $j \in \{0, \ldots, O(n)\}$. Also, by Lemma 3.2, each h_p^Π can be computed in $\widehat{\mathsf{TC}}_1^0$, and polynomially-many bits of the binary expansion of $1/p$ can be obtained in AC^0.

Using AC^0 circuitry we can multiply together the $O(\log n)$-bit numbers $W_j(X,Y) \bmod p$ and h_p^Π, and then obtain the binary expansion of $((W_j(X,Y) \bmod p) h_p^\Pi) \cdot (1/p)$ (since multiplying an n-bit number by a $\log n$ bit number can be done in AC^0).

Thus, with one more layer of majority gates, we can compute

$$A'_\Pi = \sum_{p \in \Pi} \sum_{j=0}^{2n+1} \frac{(W_j(X,Y) \bmod p) h_p^\Pi}{p}$$

and strip off the integer part, to obtain the desired approximation. □

Corollary 3.6. *Let Π be any set of primes such that the product M of these primes lies in $(2^{n^c}, 2^{n^d})$ for a fixed constant $d > c \geqslant 3$. Then, given Z in CRR$_\Pi$ representation and the numbers h_p^Π for each $p \in \Pi$, we can compute the binary representation of a* good *approximation to $\frac{Z}{M}$ in $\widehat{\mathsf{TC}}_1^0$*

Before presenting our approximation $\widetilde{V}(X, Y)$, first we present a claim, which helps motivate the definition.

Claim 3.7. Let Π_i for $i \in \{1, \dots, n^c\}$ be n^c pairwise disjoint sets of primes such that $M_i = \prod_{p \in \Pi_i} p \in (2^{n^c}, 2^{n^d})$ (for some constants $c, d : 3 \leqslant c < d$). Let $\Pi = \cup_{i=1}^{n^c} \Pi_i$. Then, for any value A, it holds that

$$A(1 - \frac{n^c}{2^{n^c}}) < \frac{A \prod_{i=1}^{n^c} (M_i - 1)}{\prod_{i=1}^{n^c} M_i} < A$$

The claim follows immediately from Proposition 3.8 below:

Proposition 3.8. *Let $x > 1$ be given. Then for all $n > 1$,*

$$1 - \frac{n}{x} < \left(1 - \frac{1}{x}\right)^n.$$

Proof. By induction. Assume that $1 - \frac{n}{x} \leqslant (1 - \frac{1}{x})^n$. Then

$$\begin{aligned} 1 - \left(\frac{n+1}{x}\right) &= 1 - \frac{n}{x} + \frac{n}{x} - \left(\frac{n+1}{x}\right) \\ &< \left(1 - \frac{1}{x}\right)^n - \frac{1}{x} \\ &< \left(1 - \frac{1}{x}\right)^n - \frac{(1 - \frac{1}{x})^n}{x} \\ &= \left(1 - \frac{1}{x}\right)^n \left(1 - \frac{1}{x}\right) \end{aligned}$$

□

Now, finally, we present our desired approximation. $\widetilde{V}(X, Y)$ is $2^{n^c} \cdot V'(X, Y)$, where $V'(X, Y)$ is an approximation (within $1/2^{n^{2c}}$) of

$$V(X, Y) = \frac{W(X, Y) \prod_{i=1}^{n^c} (M_i - 1)/2}{\prod_{i=1}^{n^c} M_i}.$$

Note that

$$\begin{aligned} W(X,Y) - 2^{n^c}V(X,Y) &= W(X,Y) - 2^{n^c}\frac{W(X,Y)\prod_{i=1}^{n^c}(M_i-1)/2}{\prod_{i=1}^{n^c}M_i} \\ &= W(X,Y) - \frac{W(X,Y)\prod_{i=1}^{n^c}(M_i-1)}{\prod_{i=1}^{n^c}M_i} \\ &< W(X,Y)\frac{n^c}{2^{n^c}} < \frac{2^{2n}n^c}{2^{n^c}} \end{aligned}$$

and

$$\begin{aligned} 2^{n^c}V(X,Y) - \widetilde{V}(X,Y) &= 2^{n^c}V(X,Y) - 2^{n^c}V'(X,Y) \\ &= 2^{n^c}(V(X,Y) - V'(X,Y)) \\ &\leqslant 2^{n^c}(\frac{1}{2^{n^{2c}}}) \\ &= \frac{2^{n^c}}{2^{n^{2c}}}. \end{aligned}$$

Thus $X/Y - \widetilde{V}(X,Y) = (X/Y - W(X,Y) + (W(X,Y) - 2^{n^c}V(X,Y)) + (2^{n^c}V(X,Y) - \widetilde{V}(X,Y)) < 2^{-(n+1)} + n^c 2^{2n}/2^{n^c} + 2^{n^c}/2^{n^{2c}} < 1/Y$.

Lemma 3.9. *Let Π_i for $i \in \{1,\dots,n^c\}$ be n^c pairwise disjoint sets of primes such that $M_i = \prod_{p\in\Pi_i} p \in (2^{n^c}, 2^{n^d})$ (for some constants $c, d : 3 \leqslant c < d$). Let $\Pi = \cup_{i=1}^{n^c}\Pi_i$. Then, given X, Y and the Π_i, we can compute $\widetilde{V}(X,Y)$ in $\widehat{\mathsf{TC}}_3^0$.*

Proof. Via Lemma 3.2, in $\widehat{\mathsf{TC}}_1^0$ we can compute the CRR_Π representation of each M_i, as well as the numbers $W_j \bmod p$ (using Lemma 3.4). Also, as in Lemma 3.5, we can compute the values h_p^Π for each prime p.

Then, via Lemma 3.2, with one more layer of majority gates we can compute the CRR representation of $\prod_i (M_i - 1)/2$, as well as the CRR representation of $2^{2(n+1)t}W(X,Y) = \sum_{j=0}^{2n+1} W_j(X,Y)$. The CRR representation of the product $2^{2(n+1)t}W(X,Y)\cdot\prod_i(M_i-1)/2$ can then be computed with AC^0 circuitry to obtain the CRR representation of the numerator of the expression for $V(X,Y)$. (It is important to note that $2^{2(n+1)t}W(X,Y)\cdot\prod_i(M_i-1)/2 < \prod_i M_i$, so that it is appropriate to talk about this CRR representation. Indeed, that is the reason why we divide each factor $M_i - 1$ by two.)

This value can then be converted to binary with one additional layer of majority gates, via Corollary 3.6, to obtain $\widetilde{V}(X,Y)$. □

This completes the proof of Theorem 3.3. □

Corollary 3.10. *Let Π be any set of primes such that the product M of these primes lies in $(2^{n^c}, 2^{n^d})$ for a fixed constant $d > c \geqslant 3$. Then, given Z in CRR$_\Pi$ representation, the binary representation of Z can be computed in $\widehat{\mathsf{TC}}^0_3$*

Proof. Recall from the proof of Theorem 3.3 that, in order to compute the bits of $Z/2$, our circuit actually computes an approximation to $(Z+1)/2$. Although, of course, it is trivial to compute $Z/2$ if Z is given to us in binary, let us consider how to modify the circuit described in the proof of Lemma 3.9, if we were computing $\widetilde{V}(Z+1,2)$, where we are given Z in CRR representation.

With one layer of majority gates, we can compute the CRR$_\Pi$ representation of each M_i and the values h_p^Π for each prime p. (We will not need the numbers W_j mod p.)

Then, with one more layer of majority gates we can compute the CRR representation of $\prod_i (M_i - 1)/2$. In place of the gates that store the value of the CRR representation of $2^{2(n+1)t}W(X,Y)$, we insert the CRR representation of Z (which is given to us as input) and using AC^0 circuitry store the value of $Z+1$. The CRR representation of the product $Z+1 \cdot \prod_i (M_i - 1)/2$ can then be computed with AC^0 circuitry to obtain the CRR representation of the numerator of the expression for $V(Z+1,2)$.

Then this value can be converted to binary with one additional layer of majority gates, from which the bits of Z can be read off. ☐

It is rather frustrating to observe that the input values Z are not used until quite late in the $\widehat{\mathsf{TC}}^0_3$ computation (when just one layer of majority gates remains). However, we see no simpler uniform algorithm to convert CRR to binary.

For our application regarding problems in the counting hierarchy, it is useful to consider the analog to Theorem 3.3 where the values X and Y are presented in CRR notation.

Theorem 3.11. *The function taking as input $X \in [0, 2^n), Y \in [1, 2^n)$ (in CRR) as well as 0^m, and producing as output the binary expansion of X/Y correct to m places is in $\widehat{\mathsf{TC}}^0_3$.*

Proof. We assume that the CRR basis consists of pairwise disjoint sets of primes M_i, as in Lemma 3.9.

The algorithm is much the same as in Theorem 3.3, but there are some important differences that require comment. The first step is to determine if $Y = 1$, which can be done using AC^0 circuitry (since the CRR of 1 is easy to recognize). The next step is to determine a value t such that $2^{t-1} \leqslant Y < 2^t$. Although this is trivial when the input is presented in binary, when the input is given in CRR it requires the following lemma:

Lemma 3.12. *(Adapted from [AAD00, DMS94, ABKPM09]) Let* X *be an integer from* $(-2^n, 2^n)$ *specified by its residues modulo each* $p \in \Pi_n$*. Then, the predicate* $X > 0$ *is in* $\widehat{\mathsf{TC}}^0_2$

Since we are able to determine inequalities in majority-depth two, we will carry out the initial part of the algorithm from Theorem 3.3 using *all* possible values of t, and then select the correct value between the second and third levels of MAJORITY gates.

Thus, for each t, and for each j, we compute the values $W_{j,t}(X+1, Y) = (X+1)(2^t - Y)^j(2^{(2n+1-j)t})$ in CRR, along with the desired number of bits of accuracy of $1/p$ for each p in our CRR basis.

With this information available, as in Lemma 3.9, in majority-depth one we can compute h_p^{Π}, as well as the CRR representation of each M_i, and thus with AC^0 circuitry we obtain $(W_{j,t}(X+1, Y)$ and the CRR for each $(M_i - 1)/2$.

Next, with our second layer of majority gates we sum the values $W_{j,t}(X+1, Y)$ (over all j), and at this point we also will have been able to determine which is the correct value of t, so that we can take the correct sum, to obtain $2^{2(n+1)t}W(X, Y)$.

Thus, after majority-depth two, we have obtained the same partial results as in the proof of Lemma 3.9, and the rest of the algorithm is thus identical. □

Proposition 3.13. *Iterated product is in uniform* $\widehat{\mathsf{TC}}^0_3$.

Proof. The overall algorithm is identical to the algorithm outlined in [MT98], although the implementation of the basic building blocks is different. In majority-depth one, we convert the input from binary to CRR. With one more level of majority gates, we compute the CRR of the product.

Simultaneously, in majority-depth two we compute the bottom two levels of our circuit that computes from CRR to binary, as in Corollary 3.10.

Thus, with one final level of majority gates, we are able to convert the answer from CRR to binary. □

3.1.3 Consequences for the Counting Hierarchy

The following proposition gives a way to translate results obtained in the context of TC^0 to the counting hierarchy.

Proposition 3.14. *(Implicit in [ABKPM09, Theorem 4.1].) Let* A *be a set such that, for some* k*, some polynomial-time computable function* f *and for some dlogtime-uniform*[9] $\widehat{\mathsf{TC}}^0_d$ *circuit family* C_n*, it holds that* $x \in A$ *if and only if* $C_{|x|+2^{|x|^k}}(x, f(x,1)f(x,2)\dots f(x,2^{|x|^k}))$ *accepts. Then* $A \in \mathsf{PH}^{\mathsf{CH}_d}$*.*

Corollary 3.15. $\mathsf{BitSLP} \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$.

Proof. This is immediate from Proposition 3.14 and Corollary 3.10.

Let f be the function that takes as input a tuple $(C,(p,j))$ and if p is a prime, evaluates the arithmetic circuit C mod p and outputs the j-th bit of the result. This function f, taken together with the $\widehat{\mathsf{TC}}^0_3$ circuit family promised by Corollary 3.10, satisfies the hypothesis of Proposition 3.14. (There is a minor subtlety, regarding how to partition the set of primes into the groupings M_i, but this is easily handled by merely using all of the primes of a given length, at most polynomially-larger than $|C|$.) □

Via essentially identical methods, using Theorem 3.11, we obtain:

Corollary 3.16. $\{(C_X, C_Y, i)$ *: the* i*-th bit of the quotient* $X/Y\}$*, where* X *and* Y *are represented by arithmetic circuits* C_X *and* C_Y*, respectively, is in* $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$.

3.2 Powering

We investigate the complexity of integer powering and powering constant size matrices from the perspective of optimizing the majority depth. We present TC^0 circuits with majority depth three for both these problems. Note that for the succinct versions of the two problems in this section, there is a simple algorithm that is derived from Corollary 3.15, namely construct an SLP via repeated squaring and run the BitSLP subroutine on this SLP which immediately gives a CH_3 bound. However, in the case of the verbose versions which lie in $\widehat{\mathsf{TC}}^0_3$ via methods developed in this section, such a reduction doesn't seem immediate. Here we present alternate algorithms in the next two subsections for these powering problems which achieve $\widehat{\mathsf{TC}}^0_3$ upper bounds in the verbose case, and CH_3 bound in the succinct version.

3.2.1 Integer Powering

Since iterated integer product is in uniform $\widehat{\mathsf{TC}}^0_3$, by Proposition 3.13, it is immediate that integer powering is also in this class. Here, we present an alternative algorithm for integer powering, which serves to illustrate the approach that we take to matrix powering.

[9]Note that the dlogtime-uniformity condition is crucial here. All references to $\widehat{\mathsf{TC}}^0_d$ will refer to the dlogtime-uniform version of this class, unless we specifically refer to nonuniform circuits.

Theorem 3.17. *The function taking as input $X \in [0, 2^n)$, 1^m and 1^i(where $i \in [1, nm]$) and producing as output the i-th bit of X^m is in $\widehat{\mathsf{TC}}^0_3$.*

Proof. Our algorithm is as follows:

1. Convert X to CRR. Let $X \equiv X_j \mod p_j$ for $j \in [k]$. This is implementable in $\widehat{\mathsf{TC}}^0_1$ by item 5 in Lemma 3.2.
2. Compute X^m by reducing via Fermat's little theorem. Since $X^{p-1} \equiv 1 \mod p$ for any prime p, we can compute $X_j^{m_j} \mod p_j$ where $m = q_j(p_j - 1) + m_j$ for $j \in [k]$. This step is in AC^0 via item 3 in Lemma 3.2. In parallel, compute the first two phases of our uniform algorithm to convert CRR to binary (Corollary 3.10).
3. At this stage, we have the answer encoded in CRR, and we invoke the final layer of the circuit from Corollary 3.10, convert the answer to binary.

Putting these three together, we have integer powering in $\widehat{\mathsf{TC}}^0_3$. □

3.2.2 Integer Matrix Powering

Theorem 3.18. *The function MPOW(A, m, p, q, i) taking as input a $(d \times d)$ integer matrix $A \in \{0,1\}^{d^2 n}$, p, q, 1^i, where $p, q \in [d]$, $i \in [O(n)]$ and producing as output the i-th bit of the (p, q)-th entry of A^m is in $\widehat{\mathsf{TC}}^0_3$.*

For a $(d \times d)$ matrix, the characteristic polynomial $\chi_A(x) : \mathbb{Z} \to \mathbb{Z}$ is a univariate polynomial of degree at most d. Let $q, r : \mathbb{Z} \to \mathbb{Z}$ be univariate polynomials of degree at most $(m - d)$ and $(d - 1)$ such that $x^m = q(x)\chi_A(x) + r(x)$. By the Cayley-Hamilton theorem, we have that $\chi_A(A) = 0$. So, in order to compute A^m, we just have to compute $r(A)$.

Lemma 3.19. *Given a $(d \times d)$ matrix A with entries that are n-bit integers, the coefficients of the characteristic polynomial of A in CRR can be computed in $\widehat{\mathsf{TC}}^0_1$.*

Proof. We convert the entries of A to CRR and compute the determinant of $(xI - A)$. This involves an iterated sum of $O(2^d d!)$ integers each of which is an iterated product of d n-bit integers. The conversion to CRR is in $\widehat{\mathsf{TC}}^0_1$ by item 5 in Lemma 3.2. Since addition, multiplication, and powering of $O(1)$ numbers of $O(\log n)$ bits is computable in AC^0 (by Lemma 3.2, items 3, 4 and 6), it follows that the coefficients of the characteristic polynomial can be computed in $\widehat{\mathsf{TC}}^0_1$. □

Lemma 3.20. *Given the coefficients of the polynomial r, in CRR, and given A in CRR, we can compute A^m in CRR using AC^0 circuitry.*

Proof. Recall that $A^m = r(A)$. Let $r(x) = r_0 + r_1 x + \ldots + r_{d-1}x^{d-1}$. Computing any entry of $r(A)$ in CRR involves an iterated sum of $O(1)$ many numbers which are themselves an iterated product of $O(1)$ many $O(\log n)$-bit integers. The claim follows by appeal to Lemma 3.2. □

Lemma 3.21. *(Adapted from [HV06]) Let p be a prime of magnitude $\mathsf{poly}(m)$. Let $g(x)$ of degree m and $f(x)$ of degree d be monic univariate polynomials over GF_p, such that $g(x) = q(x)f(x) + r(x)$ for some polynomials $q(x)$ of degree $(m-d)$ and $r(x)$ of degree $(d-1)$. Then, given the coefficients of g and f, the coefficients of r can be computed in $\widehat{\mathsf{TC}}^0_1$.*

Proof. Following [HV06], let $f(x) = \sum_{i=0}^{d} a_i x^i$, $g(x) = \sum_{i=0}^{m} b_i x^i$, $r(x) = \sum_{i=0}^{d-1} r_i x^i$ and $q(x) = \sum_{i=0}^{m-d} q_i x^i$. Since f, g are monic, we have $a_d = b_m = 1$. Denote by $f_R(x), g_R(x), r_R(x)$ and $q_R(x)$ respectively the polynomial with the i-th coefficient $a_{d-i}, b_{m-i}, r_{d-i-1}$ and q_{m-d-i} respectively. Then note that $x^d f(1/x) = f_R(x)$, $x^m g(1/x) = g_R(x)$, $x^{m-d}q(1/x) = q_R(x)$ and $x^{d-1}r(1/x) = r_R(x)$.

We use the Kung-Sieveking algorithm (as implemented in [HV06]). The algorithm is as follows:

1. Compute $\tilde{f}_R(x) = \sum_{i=0}^{m-d}(1 - f_R(x))^i$ via interpolation modulo p.
2. Compute $h(x) = \tilde{f}_R(x) g_R(x) = c_0 + c_1 x + \ldots + c_{d(m-d)+m} x^{d(m-d)+m}$. from which the coefficients of $q(x)$ can be obtained as $q_i = c_{d(m-d)+m-i}$.
3. Compute $r(x) = g(x) - q(x)f(x)$.

To prove the correctness of our algorithm, note that we have $g(1/x) = q(1/x)f(1/x) + r(1/x)$. Scaling the whole equation by x^m, we get $g_R(x) = q_R(x)f_R(x) + x^{m-d+1}r_R(x)$. Hence when we compute $h(x) = \tilde{f}_R(x)g_R(x)$ in step 2 of our algorithm, we get

$$h(x) = \tilde{f}_R(x)g_R(x) = \tilde{f}_R(x)q_R(x)f_R(x) + x^{m-d+1}\tilde{f}_R(x)r_R(x).$$

Note that $\tilde{f}_R(x)f_R(x) = \tilde{f}_R(x)(1-(1-f_R(x))) = \sum_{i=0}^{m-d}(1-f_R(x))^i - \sum_{i=0}^{m-d}(1-f_R(x))^{i+1} = 1-(1-f_R(x))^{m-d+1}$ (a telescoping sum). Since f is monic, f_R has a constant term which is 1 and hence $(1-f_R(x))^{m-d+1}$ does not contain a monomial of degree less than $(m-d+1)$. This is also the case with $x^{m-d+1}\tilde{f}_R(x)r_R(x)$, and hence all the monomials of degree less than $(m-d+1)$ belong to $q_R(x)$.

Now we justify why the algorithm above is amenable to a $\widehat{\mathsf{TC}}^0_1$ implementation: Firstly, note that given $f(x)$ and $g(x)$, the coefficients of $f_R(x)$ and $g_R(x)$ can be computed in NC^0. To compute the coefficients of $\tilde{f}_R(x)$, we use interpolation via the discrete Fourier transform (DFT) using arithmetic modulo p. Find a generator w of the multiplicative group modulo p and substitute $x = \{w^1, w^2, \ldots, w^{p-1}\}$ to obtain a system of linear equations in the coefficients F of $\tilde{f}_R(x) : V \cdot F = Y$, where Y is the vector consisting of $\tilde{f}_R(w^i)$ evaluated at the various powers of w. Since the underlying linear transformation $V(w)$ is a DFT, it is invertible; the inverse DFT $V^{-1}(w)$ is equal to $V(w^{-1}) \cdot (p-1)^{-1}$, which is equivalent to $-V(w^{-1}) \bmod p$. We can find each coefficient of $\tilde{f}_R(x)$ evaluating $V^{-1}Y$, i.e., by an inner product of a row of the inverse DFT-matrix with the vector formed by evaluating $\sum_{i=1}^{(m-d+1)}(1-f_R(x))^{i-1}$ at various powers of w and dividing by $p-1$. The terms in this sum can be computed in AC^0, and then the sum can be computed in majority-depth one, to obtain the coefficients of $\tilde{f}_R(x)$. The coefficients of $h(x)$ in step 2 could be obtained by iterated addition of the product of certain coefficients of $\tilde{f}_R$ and g_R, but since the coefficients of $\tilde{f}_R$ are themselves obtained by iterated addition of certain terms t, we roll steps 1 and 2 together by multiplying these terms t by the appropriate coefficients of g_R. Thus steps 1 and 2 can be accomplished in majority-depth 1. Then step 3 can be computed using AC^0 circuitry. □

Proof. (of Theorem 3.18)

Our $\widehat{\mathsf{TC}}^0_3$ circuit C that implements the ideas above is the following:

0. At the input, we have the d^2 entries A_{ij}, $i, j \in [d]$ of A, a set Π of short primes (such that Π can be partitioned in to n^c sets Π_i that are pairwise disjoint, i.e., $\Pi = \cup_{i=1}^{n^c}\Pi_i$), the numbers $I = \{1, 2, \ldots, (m-d+1)\}$.
1. In majority-depth one, we obtain (1) $A_{ij} \mod p$ for each prime p in our basis, and (2) $M_i = \prod_{p \in \Pi_i} p$ for all the n^c sets that constitute Π, and (3) the CRR of the characteristic polynomial of A (via appeal to Lemma 3.19).
2. In the next layer of threshold gates, we compute (1) $\prod_i^{n^c}(M_i - 1)/2$ in CRR, and (2) the coefficients of the polynomial r in CRR, by appeal to Lemma 3.21.
3. At this point, by Lemma 3.20, AC^0 circuitry can obtain $r(A) = A^m$ in CRR, and with one more layer of MAJORITY gates we can convert to binary, by appeal to Corollary 3.10.

□

3.3 Sum of Square Roots

The Sum-of-Square-Roots problem is a classic problem that is a key subroutine in many algorithms in computational geometry [10]. It was posed explicitly by Joseph O'Rourke in [O1] and has been open ever since. [ABKPM09] show that SSQRT reduces in Polynomial time to PosSLP. Here we show a weak ($\mathsf{PH}^{\mathsf{PP}}$) reduction to (2×2) matrix powering. We now formally define the problem:

Definition 3.22. [The Sum-of-Square-Roots Problem] Let $\boldsymbol{a} = (a_1, \ldots, a_n)$ be a list of n-bit positive integers, and let $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n) \in \{-1, +1\}^n$. Define $\mathsf{SSQRT}(\boldsymbol{a}, \boldsymbol{\sigma})$ to be the problem of determining if:

$$\sum_{i=1}^{n} \sigma_i \sqrt{a_i} > 0$$

3.3.1 Reducing Sum-of-square-roots to Matrix Powering

In this section, we present a reduction, showing that the Sum-of-Square-Roots problem is reducible, in some sense, to the problem of computing large powers of 2-by-2 integer matrices.

Definition 3.23. $\mathsf{Bit\text{-}2\text{-}MatPow}(A, N, B, i, j)$ (where A is a 2×2 matrix of n-bit integers and N, B are n-bit positive integers) is the problem of determining the B^{th} bit of $(A^N)_{i,j}$.

Theorem 3.24. $\mathsf{SSQRT} \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{Bit\text{-}2\text{-}MatPow}}}$.

Our proof makes use of Linear Fractional Transformations (LFTs), which in turn correspond directly to 2-by-2 matrices. Appendix A contains the necessary background concerning LFTs, including the proof of the following lemma:

Lemma 3.25. *If $[p_n(a), q_n(a)]$ denotes the n^{th} convergent for the matrix sequence $M_1, M_2, \ldots$ where each $M_i = L(a) = \begin{pmatrix} a & a \\ 1 & a \end{pmatrix}$, then $q_n(a) - p_n(a) < a\left(1 - \frac{1}{a}\right)^{n+1}$. Thus if $a \in [1, 2]$, then $0 \leqslant q_n(a) - p_n(a) < 2^{-n}$, and for all $n, \sqrt{a} \in [p_n(a), q_n(a)]$. Furthermore, $p_n(a) = (L(a)^n)_{1,2} / (L(a)^n)_{2,2}$.*

Proof. (of Theorem 3.24) Let $(\boldsymbol{a}, \boldsymbol{\sigma})$ be an input instance for SSQRT. Let α_i be a positive integer satisfying $2^{\alpha_i} < a_i \leqslant 2^{\alpha_i + 1}$. Further, let $a_i' = a_i / 2^{\alpha_i}$ be a rational in $(1, 2]$. Hence, by

[10] To quote Jeff Erickson, "This problem arises naturally in computational geometry, especially in problems involving Euclidean shortest paths, and is a significant stumbling block in transferring algorithms for those problems from the real RAM to the standard integer RAM." See http://cstheory.stackexchange.com/questions/4053/sum-of-square-roots-hard-problems for connections between SSQRT and other areas like Algorithmic Game Theory. See http://cs.smith.edu/~orourke/TOPP/P33.html for a detailed account of advances made so far.

an application of Lemma 3.25, any number, say $p_M(a_i')$ in the M^{th} convergent interval of $L(a_i)$ approximates $\sqrt{a_i'}$ with an error of at most 2^{-M}. To obtain an approximation of $\sqrt{a_i}$ from this we need to multiply $p_M(a_i')$ by $2^{\lfloor \frac{\alpha_i}{2} \rfloor}$ (and if α_i is odd then we must also multiply this by an approximation to $\sqrt{2}$ – which can also be approximated in this way by setting $a = 2$ and $\alpha = 0$).

How good an approximation is needed? That is, how large must M be? Tiwari has shown [Tiw92] that if a sum of n square roots $\pm\sqrt{a_i}$ is not zero, where each a_i has binary representation of length at most s, then the sum is bounded from below by

$$2^{-(s+1)2^n}$$

Thus taking $M = 2(\log n)(s+1)2^n$ and obtaining an approximation of each $\sqrt{a_i}$ to within 2^{-M} provides enough accuracy to determine the sign of the result. By Lemma 3.25, a suitable approximation is provided by $(L(a_i)^M)_{1,2}/(L(a_i)^M)_{2,2}$ (or – if α_i is odd – by the expression $(L(a_i)^M)_{1,2}(L(2)^M)_{1,2}/(L(a_i)^M)_{2,2}(L(2)^M)_{2,2}$). Denote this fraction by C_i/D_i. (Note that each $D_i > 0$.)

Note that

$$\begin{aligned}\sum_{i=1}^{n} \sigma_i \sqrt{a_i} > 0 \quad &\Leftrightarrow \quad \sum_{i=1}^{n} \sigma_i \frac{C_i}{D_i} > 0 \\ &\Leftrightarrow \quad \sum_{i=1}^{n} \sigma_i C_i \prod_{j \neq i} D_j > 0\end{aligned}$$

We will need to re-write the expression $\sum_{i=1}^{n} \sigma_i C_i \prod_{j\neq i} D_j$ in order to make it easier to evaluate. First, note that this expression is of the form $\sum_{i=1}^{n} \prod_{j=1}^{n} Z_{i,j}$ for integers $Z_{i,j}$ whose binary representation is of length less than 2^{n^2}. Thus this expression can written in the form $\sum_{i=1}^{n} \prod_{j=1}^{n} \sum_{k=1}^{2^{n^2}} b_{i,j,k} 2^k$ where each $b_{i,j,k} \in \{-1, 0, 1\}$ is easily computable from the input and from the oracle Bit-2-MatPow. Via the distributive law, this can be rewritten as $\sum_{i=1}^{n} \sum_{(k_1,k_2,\ldots,k_n)\in[2^{n^2}]^n} \prod_{j=1}^{n} b_{i,j,k_j} 2^{k_j}$.

Thus there is a function f computable in polynomial time with an oracle for Bit-2-MatPow that, on input $(\boldsymbol{a}, \boldsymbol{\sigma}, i, k_1, k_2, \ldots, k_n, j, \ell)$ outputs the ℓ^{th} bit of the number $\prod_{j=1}^{n} b_{i,j,k_j} 2^{k_j}$. (Namely, the algorithm queries the n oracle bits corresponding to b_{i,j,k_j} and combines this information with $\boldsymbol{\sigma}$ to obtain the sign $\in \{-1, 0, 1\}$, and computes the value of the exponent $\sum_j k_j$, and from this easily determines the value of bit ℓ of the binary representation.)

Since addition of m numbers, each consisting of m bits is computable in $\widehat{TC}_1^0$, it is now immediate that the bits of this expression are computable in PH^{PP}. Thus in PH, using the bits of this expression as an oracle, one can determine if the number represented in this manner is positive or not. (Namely, is there some bit that is non-zero, and is the sign bit positive?)

□

3.4 Conditional results

In this section, we show some conditional hardness results for PosSLP and BitSLP. More specifically, we show that,

Theorem 3.26. *Let N be an n- bit number. If $\binom{2N}{N}$ has small($\mathsf{poly}(n)$-sized) arithmetic circuits, then* BitSLP *is in* $\mathsf{PH}^{\mathsf{PP}}$.

In fact, we just need the weaker condition that $\binom{2N}{N}$ has small arithmetic circuits for $N = 2^{n-1}$ to obtain the claimed upper bound for BitSLP. The main idea behind the proof, originally due to Beame et al.[BCH86] is that $M = \binom{2N}{N} = \prod_{i=1}^{k} p_i{}^{e_i}$ is a good CRR basis in the following sense: For any number $X \leqslant \binom{2N}{N}$, the list of moduli $X \bmod p_i{}^{e_i}$ is sufficient to uniquely recover X. Recall from Fact 5 that the reconstruction formula is

$$X = M\left(\sum_q \frac{X_q h_q}{q} - \left\lfloor \sum_q \frac{X_q h_q}{q} \right\rfloor\right)$$

where $q = p^e$ are relatively prime, prime powers, $M = \binom{2N}{N}$ $h_q = (M/q)^{-1} \bmod q$ and $X_q = X \bmod q$. First we show that for an SLP X and modulus $q = p_i{}^{e_i}$, $X \bmod q$ can be computed efficiently:

Lemma 3.27. *Let X be a number computed by SLP(X) of size $\mathsf{poly}(n)$ and let q be an n-bit number. Then $Z = \left\lfloor \frac{X}{q} \right\rfloor$ has an SLP(Z) of size polynomial in n.*

Proof. Abbreviate as A' the number $\left\lfloor \frac{A}{q} \right\rfloor$ and as a the number ($A \bmod q$) (where q is fixed)

Then, using the fact that $A = A'q + a$, we have:

$$(A+B)' = A' + B' + (a+b)'$$

and

$$\begin{aligned}
(AB)' &= ((A'q + a)B)' \\
&= A'B + (aB)' + (aB \bmod q)' \\
&= A'B + (a(B'q + b))' + 0 \\
&= A'B + aB' + (ab)'
\end{aligned}$$

Now for every gate G in SLP(X), we can compute g in P. Also for a small (i.e. n-bit) number c, computing c' is easy (in $\mathsf{TC}^0 \subseteq \mathsf{P}$). Thus in P we can produce SLP(G′) for every gate G in SLP(X) using the equations above recursively. Notice that size of SLP(G′) is at most a polynomial in size of SLP(G). □

The following fact shows that when N is a power of 2, e_i such that ${p_i}^{e_i} \| M$ for every prime p_i can be computed efficiently:

Fact 6. Let $M = \binom{2^n}{2^{n-1}}$, then for every prime $p < 2^n$ the exponent e such that $p^e \| M$ can be determined in P. Further, $e < n$ i.e. p^e is $\mathsf{poly}(n)$ bits long.

Proof.

$$e = \sum_{i=0}^{n} \lfloor (2^n/p^i) \rfloor - 2\lfloor 2^{n-1}/p^i \rfloor$$

The proof follows by observing that $\lfloor 2x \rfloor - 2\lfloor x \rfloor \in \{0, 1\}$ and that each term in the sequence is in TC^0. □

Proposition 3.28. *The first* $O(n)$ *bits of* $1/q$ *can be computed in* TC^0 *(and hence in* P*) where* q *is a* $\mathsf{poly}(n)$*-bit odd prime power.*

Proof. Firstly note that for our purpose, a P algorithm is sufficient, which is trivial via long division. However, via [HAB02], we can get the first $O(n)$ bits even in TC^0 as follows: To compute the k-th bit of $1/q$ for an odd prime q, write $2^k = aq + b$ where $b < q$ and $b \equiv 2^k \bmod q$. The k-th bit of the binary expansion of the rational number $1/q$ is the lower order bit of a. Since 2^k is even and q is odd, both a and b have the same parity. Hence the parity of b gives the k-th bit of $1/q$. But this is easy since we just need to find $b \equiv 2^k \bmod q$, which is easy by repeated squaring. □

Lemma 3.29. *If* M *is computable by polynomial size arithmetic circuits, then* $h_q = (M/q)^{-1} \bmod q$ *for every prime power* q *can be computed in* $\mathsf{PH}^{\mathsf{PP}}$.

Proof. Guess circuits of size n^c and verify that it indeed computes $M = \binom{2N}{N} = \prod_i p_i^{e_i}$, i.e., the purported circuit X computing M does not have any spurious primes (the ones other than p_i) dividing it. Note that since we have a bound on N, this also translates to a bound on the p_i involved, i.e., $\max_i\ p_i \leqslant N$. Firstly, in NP, guess such an SLP X for M and verify that $X \bmod p_i^{e_i} \equiv 0$ for every i (Note that the e_i's can be already found in P via Fact 6). Simultaneously, check for primes $2^{n^c} > r > N \geqslant \max_i p_i$ that $\binom{2N}{N} \equiv X \bmod r$. Note that this is amenable to a $\mathsf{PH}^{\mathsf{PP}}$ implementation: Overall, if a polynomial-sized SLP computing M is guaranteed to exist, then for all $q = {p_i}^{e_i}$, h_q can be computed in $\mathsf{PH}^{\mathsf{PP}}$ (by iterated addition on the exponents). □

We are now ready to prove Theorem 3.26.

Proof. (of Theorem 3.26) Given a BitSLP instance C representing a (potentially) 2^s-bit number where s is the size of C, we can compute $C_q = C \bmod q$ and $1/q$ in P by Lemma 3.27 and Proposition 3.28 respectively. From the premise of the theorem that M has small arithmetic circuits (of size n^c for a fixed c, where c depends only on the input SLP X), $h_q = (M/q)^{-1} \bmod q$ can be computed in $\mathsf{PH}^{\mathsf{PP}}$ by Lemma 3.29. Now we have to compute the iterated sum to recover a bit of the SLP C. But this is in $\mathsf{PH}^{\mathsf{PP}}$ by Lemma 3.2.

$\square$

We know that BitSLP is hard for #P. Our assumption that $\binom{2N}{N}$ (even just for $N = 2^n$) has small arithmetic circuits gets us closer to the goal of giving a tight bound for BitSLP. Burgisser [Bür09] notes that the assumption that Permanent has small arithmetic circuits (which in turn implies small boolean circuits) yields that certain sequences like $N!$ are easy to compute ($\mathsf{poly}(\log N)$ size arithmetic circuits). However Permanent having small arithmetic circuits yields a collapse of the Counting Hierarchy and in our case puts BitSLP in P/poly. How strong in comparison is the assumption that $\binom{2N}{N}$ has small arithmetic circuits? It seems to yield the conclusion that $N!$ is easy. This can be obtained as follows: If the sequence $\binom{2N}{N}$ is easy, then the product

$$\prod_{i=1}^{\log N} \binom{\frac{N}{2^{i-1}}}{\frac{N}{2^i}}^{2^i} = N!$$

(because N is a power of 2) is also easy. Specifically if $\binom{2N}{N}$ has circuits of size $\mathsf{poly}(\log N)$, then so does $N!$.

3.5 Variations on PosSLP

In addition to BitSLP, there has also been interest in the related problem PosSLP [11][EY10, KP07, KS12, KP11]. By Theorem 1.1, PosSLP lies in the third level of the counting hierarchy, and is not known to be in PH[ABKPM09], but in contrast to BitSLP, it is not known (or believed [EY10]) to be NP-hard. Our theorems do not imply any new bounds on the complexity of PosSLP, but we do conjecture that BitSLP and PosSLP both lie in $\mathsf{PH}^{\mathsf{PP}}$. This conjecture is based mainly on the heuristic that says that, for problems of interest, if a nonuniform circuit is known, then corresponding dlogtime-uniform circuits usually also exist.

[11] Recall from Chapter 1 that PosSLP = $\{C :$ the number represented by arithmetic circuit C is positive$\}$

However the best lower bound currently known is P[12]. In this section we present some evidence that the current upper bounds for PosSLP might not be optimal. Our approach in this regard is to try to augment the basis of the SLP given as input to PosSLP and show that the positivity problems for even these seemingly more expressive SLPs are solved in $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$.

3.5.1 Straight Line Programs with PosSLP oracle

Consider the following version of PosSLP: An input instance consists of an SLP O over the basis $\{+, \times, -, >\}$. Here a gate labeled by $>$ takes in a single integer (which is computed by the SLP feeding in to it) as input, and tests if it is greater than 0. Denote by PosSLP_O this variation on the usual model in which the input to the PosSLP instance is an SLP which makes intermediate PosSLP queries. We show that this seemingly powerful variation Turing reduces (in NP) to PosSLP, and hence we can place it in the third level of the counting hierarchy. This gives further evidence that the current upper bounds for PosSLP are probably not optimal.

Lemma 3.30. $\mathsf{PosSLP}_O \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$

Proof. Given an oracle SLP of size s, it can have at most s PosSLP gates, which in turn can have a string of length at most s feeding in to them – this restricts the PosSLP instances queried in the oracle SLP to work over numbers of magnitude 2^{2^s}. Let $>_1, >_2, \ldots, >_r$ be the $>$ gates in this oracle SLP. In NP guess an r -bit string which gives the outputs of all the r $>$ gates. Now verify with a PosSLP oracle (that works over 2^s-bit numbers) if indeed the oracle SLP outputs a value that is greater than 0. Note that with the aid of the PosSLP oracle, the consistency of the output of the oracle SLP can be simultaneously checked with the consistency of the values output at the r intermediate $>$ gates. $\square$

3.5.2 Permanent with succinctly represented entries

Given that there are no hardness results known for PosSLP apart from P-hardness, we create an artificial variation of PosSLP that is PP-hard. However, like in the previous scenario, even this stronger variation has an algorithm that places it in the third level of counting hierarchy.

Given an $n \times n$ matrix A whose entries are themselves SLPs of size at most s ($s = \mathsf{poly}(n)$), we call $\mathsf{PosPermSLP}(A, s)$ the problem of determining if $\mathtt{Perm}(A) > R$ where R is a number

[12]P-hardness follows by reduction from the Circuit Value Problem: Given a circuit C and input x, we want to check if C accepts x. To this end, let D be the following circuit: In C, push all the negations to the leaves (with at most a constant factor increase in size of the circuit), arithmetize C and plug in x. It is easy to see that $D(x) > 0$ iff C accepts x.

representable by an SLP of size $\mathsf{poly}(n)$. Denote by A_{ij} the SLP that represents the (i,j)-th entry of A.

Lemma 3.31. $\mathsf{PosPermSLP} \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$

Proof. Our proof is loosely based on Theorem 4.2 of [ABKPM09]. Since $s = \mathsf{poly}(n)$, $|\mathrm{Perm}(A)| \leqslant 2^{2^{n^c}}$ for some constant c and also $|R| \leqslant 2^{2^{n^d}}$ for some constant d. Hence $\mathsf{poly}(n)$-bit primes suffice for the representing $\mathrm{Perm}(A)$ and R in the CRR basis. Let M denote the product of the CRR basis. To uniquely represent $\mathrm{Perm}(A)$ and R, we require that $M > \mathrm{Perm}(A)$ and also $M > R$. Now, notice that if $\mathrm{Perm}(A) > R$, then $\frac{\mathrm{Perm}(A)}{M}$ and $\frac{R}{M}$ differ by at least $1/M$. Hence we need to compute both the quantities $\mathrm{Perm}(A)$ and R to an appropriate approximation and compare them. We proceed as follows:

1. In NP guess the CRR basis, and in P compute each entry of A modulo p and also R modulo p. For each p in the CRR basis, compute the generator of $\mathbb{Z}^*{}_p$ in NP and compute each monomial of $\mathrm{Perm}(A)$ mod p in TC^0.
2. In $\mathsf{PH}^{\mathsf{PP}}$, compute the iterated sum of monomials of the permanent modulo a prime p (which yields $\mathrm{Perm}(A)$ mod p) for every prime p. Simultaneously, compute M_{p_i} which is the product $\prod_{j \neq i} p_j$ mod p_i. Note that further in polynomial time, we can find $h_{p_i} = (M/p_i)^{-1}$ mod p_i
3. In $\mathsf{PH}^{\mathsf{PP}}$, we can implement $\sum_p (\mathrm{Perm}(A) \bmod p) h_p \sigma_p$ and simultaneously the sum $\sum_p R_p h_p \sigma_p$ where σ_p is the result of truncating $1/p$ to $2^{n^{cd}}$ bits. Note that these two quantities are exactly $\frac{\mathrm{Perm}(A)}{M}$ and $\frac{R}{M}$, and hence it suffices to compare their bits up to an accuracy of $1/M$, which is again in PH.

Overall the complexity of this procedure is $\mathsf{NP}^{\mathsf{PH}^{\mathsf{PP}^{\mathsf{PH}^{\mathsf{PP}}}}}$ which yields a bound of $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$ as claimed. □

These variations seem to strengthen the belief that there is a 'gap' between PosSLP and $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$. Similar conclusions hold for appropriately defined variations of BitSLP too.

3.6 Discussion

We gave an improved upper bound of $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$ for BitSLP using Theorem 3.3 and Proposition 3.14 in Corollary 3.15. The previous best upper bound due to Allender and Schnoor [AS05] was $\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}}$. Non-uniformly there is a $\widehat{\mathsf{TC}}^0_1$ circuit for division, which places the complexity of BitSLP in $\mathsf{PH}^{\mathsf{PP}}$. Since BitSLP is #P-complete, this will more or less close the gap between the known upper bounds and lower bounds. We conclude with some intriguing questions that still remain unresolved:

1. *Is conversion from CRR to binary in dlogtime-uniform* $\widehat{\mathsf{TC}}_1^0$? This problem has been known to be in P-uniform $\widehat{\mathsf{TC}}_1^0$ starting with the seminal work of Beame, Cook, and Hoover [BCH86], but the subsequent improvements on the uniformity condition [CDL01, HAB02] introduced additional complexity that translated into increased majority-depth. We have been able to reduce the majority-depth by rearranging the algorithmic components introduced in this line of research, but it appears to us that a fresh approach will be needed, in order to decrease the depth further.

2. *Is* BitSLP *in* $\mathsf{PH}^{\mathsf{PP}}$? An affirmative answer to the first question implies an affirmative answer to the second, and this would pin down the complexity of BitSLP between $\mathsf{P}^{\#\mathsf{P}}$ and $\mathsf{PH}^{\mathsf{PP}}$. We have not attempted to determine a small value of k such that BitSLP $\in (\Sigma_k^p)^A$ for some set $A \in \mathsf{CH}_3$, because we suspect that BitSLP does reside lower in CH, and any improvement in majority-depth will be more significant than optimizing the depth of AC^0 circuitry, since $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$.

3. *Is* PosSLP *in* PH? Some interesting observations related to this problem were announced recently [Ete15, JS12] – Let $\tau(n)$ denote the size of the smallest SLP over $\{+, -, \times\}$ taking as input only the constants $\{0, 1\}$, representing n. Let $\tau^+(n)$ denote the size of the smallest SLP over $\{+, \times\}$ taking as input only the constants $\{0, 1\}$, representing n. The latter are called *addition-multiplication chains* (AMC) and it is easy to observe that they are the monotone versions of SLPs. If $\tau(n)$ and $\tau^+(n)$ are polynomially related, then PosSLP is in PH as follows: Given a PosSLP instance C of size s, guess an AMC A of size s^c for a fixed constant c (such an AMC always exists if $\tau(n)$ and $\tau^+(n)$ are polynomially related), and verify in coRP using an EqSLP[13] oracle that it indeed computes the same integer n. This will place PosSLP in $\mathsf{NP}^{\mathsf{coRP}} \subseteq \mathsf{PH}$. However it is not believed that $\tau(n)$ and $\tau^+(n)$ are polynomially related, but the best known current separation is $\tau^+(2^n - 1) > \tau(2^n - 1) + 1$ due to [JS12]. Can we make this separation exponential and rule out this approach to prove PosSLP $\in$ PH?

4. *Is it easy to compute bits of large powers of small matrices?* The well-studied Sum-of-Square-Roots problem reduces to PosSLP [ABKPM09], which in turn reduces to BitSLP. But the relationship between PosSLP and the matrix powering problem (given a matrix A and n-bit integer j, output the jth bit of a given entry of A^j) is unclear, since matrix powering corresponds to evaluating *very restricted* arithmetic circuits. Recently, Galby

[13]See [ABKPM09] for more details about the problem of EqSLP which is equivalent to Polynomial Identity Testing

et al. [GOW15] give a polynomial-time algorithm for testing positivity of 2×2 and 3×3 matrices. However, this does not yield a $\mathsf{PH}^{\mathsf{PP}}$ algorithm for the Sum of Square Roots problem via our method yet – given a matrix A, and an n-bit number m, their algorithm can test positivity of A^m, whereas we need an algorithm that can test positivity of a linear combination of matrix powers, namely $\sum_{i=1}^{n} A_i^m$. Can the algorithm of [GOW15] be extended to handle testing positivity of a linear combination of matrix powers? Can *any* bit of large powers of a matrix be obtained in polynomial-time or at least PH?

Chapter 4

Skew Circuits of Small Width

In 1986, Barrington showed how to count with constant memory: More precisely, he showed that NC^1 and hence the Majority function can be computed by branching programs of constant width. As explained in chapter 2, the key idea here is the notion of permutation branching programs. Barrington had earlier showed that width three PBPs cannot compute the $\wedge_n$ function, whereas at width five, they can compute all of NC^1. It has been a tantalizing open question to understand the power of PBPs at width four. In this chapter, we study Barrington's theorem via skew circuits: First we show that skew circuits of width 7 are sufficient to capture width five PBPs and hence NC^1. We then show separations between width $1, 2, 3$ and 4 skew circuits. We also give an improved simulation of skew circuits by branching programs. The folklore conversion converts a skew circuit of width w to a branching program of width $w + 1$. We convert a skew circuit of width w to branching program of width w.

We would like to point out here one (possibly crucial) distinction that we make when we study skew circuits: Input variables and constants do not account towards the width of skew circuits usually, in previous work on skew circuits (See for example [Vin96]). However, through out our investigation of skew circuits, we have input variables and constants accounting towards the width of the circuit.

Remark 4.1. In the standard setting, the input variables and constants are not accounted towards the width of the circuit: This is because the computation happens at the gates like AND/OR and hence in some sense they should be responsible for the power of the computational model. Our reasons for accounting for inputs and constants towards the width are two fold: Firstly, it gives a fine-grained version of Barrington's theorem – It is conjectured that width-4 PBPs cannot do AND whereas width-5 is known to be capable of doing all of NC^1. Our hope was that we can leverage such a fine grained analysis to get some normal form for these circuits which seems unlikely in the traditional setting.

Indeed, our width-3 lower bound for parity achieves exactly this. In the traditional setting, width-2 branching programs can compute parity, whereas width-1 branching programs are trivial and can only compute a monomial (AND of a bunch of literals), and hence not universal. Our width-3 skew circuit model provides an interesting middle ground. It is a universal model and hence can compute all Boolean functions, provided exponential size; but it is also weak enough to require exponential size to compute the parity function.

Furthermore, this is the strongest model we know of, for which we have truly exponential lower bounds (lower bounds of the form $2^{\Omega(n)}$) for the Parity function: It is stronger than CNFs and DNFs, as it can simulate both (see Lemma 4.12 and 4.10), and it requires $2^{\Omega(n)}$ size to compute the Parity function on n bits (see Theorem 4.16). Obtaining $2^{\Omega(n)}$-size lower bounds for depth-3 circuits for any explicit function is an outstanding open problem. We refer the interested reader to the works of Paturi, Saks and Zane [PSZ00], Håstad, Jukna and Pudlák [HJP93] and more recently, the work of Goldreich and Wigderson [GW13].

4.1 Branching Programs and Skew Circuits

Skew circuits were originally introduced by Venkateswaran [Ven92] to give uniform circuit characterizations of nondeterministic time and space classes. Using these, he turns questions about complexity classes like $\mathsf{P}, \mathsf{NP}, \mathsf{PSPACE}$ to questions about the relative power of Boolean circuits with structural restrictions. The Branching Program model lends itself to an easy skew circuit characterization and vice-versa. We sketch this correspondence in this section and improve upon the known conversion from skew circuits to branching programs.

4.1.1 Branching Programs to Skew Circuits

First we discuss the following folklore conversion of branching programs to skew circuits:

Lemma 4.2. *(Folklore) Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function computed by a width-w length-ℓ branching program with at most k edges between any two consecutive layers and with the additional property that each layer reads at most one variable or its negation. Then there is a skew circuit of width $\max\{w+2, k\}$ and size $O((k+w)\ell)$ computing f.*

Proof. Given a branching program B of width w, we convert it to a skew circuit C as follows:

1. For every node g in B (except s), the circuit C has an OR gate $\vee_g$. The node corresponding to s in C is the gate that computes the constant function 1. The output gate of C is the node corresponding to t in B.

2. Suppose there are incoming edges $e_1, e_2, \dots, e_k$ to the node g from gates $g_1, g_2, \dots, g_k$ respectively. From our assumption they read the same input variable or its negation. For these wires we create AND gates $\wedge_1, \wedge_2, \dots, \wedge_k$ which feed into $\vee_g$ and each AND gate $\wedge_i$ receives two inputs: g_i and the variable (or its negation) labeling the edge e_i, respectively.

Every vertex in the BP gives rise to an OR gate in the skew circuit. And every wire in the BP gives rise to an AND gate in the skew circuit. As every wire in any layer reads the same variable or its negation, we need to add two vertices corresponding to this variable and its negation on the layer below the AND layer, i.e. in the OR layer just below it. Therefore, the width of the OR layer is at most $w + 2$, whereas the width of the AND layer is at most k. This immediately yields width and size bounds of $\max\{w + 2, k\}$ and $O((k + w)\ell)$ respectively. It is easy to see that for every $x \in \{0, 1\}^n$, $f(x) = B(x) = C(x)$. □

4.1.2 Permutation Branching Programs to skew circuits

A permutation θ is called a *transposition* if either it is the identity permutation or there exists $i \neq j$ such that $\theta(i) = j$, $\theta(j) = i$ and for all $k \neq i \neq j$, $\theta(k) = k$. We call a transposition non-trivial if it is not the identity permutation, trivial otherwise.

Definition 4.3. (Transposition Branching Programs, TBP) A width-w length-ℓ TBP is a program given by a set of ℓ instructions in which for any $1 \leqslant i \leqslant \ell$, the ith instruction is a three tuple $\langle j_i, \theta^i, \sigma^i \rangle$, where j_i is an index from $\{1, 2, \dots, |X|\}$, θ_i, σ_i are transpositions of $\{1, 2, \dots, w\}$. The output of the instruction is θ_i if $x_{j_i} = 1$ and it is σ_i if $x_{j_i} = 0$. The output of the program on input x is the product of the output of each instruction of the program on x.

Lemma 4.4. *Given a width-w PBP of length ℓ there is an equivalent width-w TBP of length* $O(w\ell)$.

Proof. It is known (see for example [Her06]) that any permutation of $\{1, 2, \dots, w\}$ can be written as a product of W transpositions of $\{1, 2, \dots, w\}$, where $W = O(w)$. Let P be a width-w PBP of length ℓ. Consider the ith instruction in the program, say $\langle j_i, \theta_i, \sigma_i \rangle$. We know that we can write θ_i as a product of W transpositions, i.e. $\theta_i = t_{i,1} \cdot t_{i,2} \dots t_{i,W}$, where for $1 \leqslant j \leqslant W$ $t_{i,j}$ is a transposition. Similarly, we have $\sigma_i = s_{i,1} \cdot s_{i,2} \dots s_{i,W}$, where $s_{i,j}$ is a transposition for $1 \leqslant j \leqslant W$.

To give a TBP equivalent to P, we replace every instruction $\langle j_i, \theta_i, \sigma_i \rangle$ in P by the following: $\langle j_i, t_{i,1}, id \rangle \cdot \langle j_i, t_{i,2}, id \rangle \dots \langle j_i, t_{i,W}, id \rangle \cdot \langle j_i, id, s_{i,1} \rangle \cdot \langle j_i, id, s_{i,2} \rangle \dots \langle j_i, id, s_{i,W} \rangle$. By a simple inductive argument we can prove that the the transposition branching program thus obtained is equivalent to P. As $W = O(w)$, the upper bound on the length of the resulting branching program follows. □

We defined TBPs as a set of instructions. Like in the case of PBPs, the definition of TBPs can be rephrased in terms of the underlying DAG. We observe the following about the DAG resulting from TBPs that result in the proof of Lemma 4.4:

A width-w TBP can be converted to an equivalent layered width w PBP in which the following conditions hold:

- There are designated source vertices $s_1, s_2, \dots, s_w$ in the first layer, say layer 1 (of in-degree 0) and sink vertices $t_1, t_2, \dots, t_w$ (of out-degree 0) in the last layer, layer ℓ.
- Each layer has exactly w vertices.
- In each layer $1 \leqslant i < \ell$, all the edges are labeled by a unique variable, say x_{j_i}.
- In each layer $1 \leqslant i \leqslant \ell$, one of the following holds:
  - either the edges corresponding to $x_{j_i} = 1$ form a non-trivial transposition and the edges corresponding to $x_{j_i} = 0$ form the identity permutation
  - or the edges corresponding to $x_{j_i} = 0$ form a non-trivial transposition and the edges corresponding to $x_{j_i} = 1$ form the identity permutation

Remark 1. As a result of the above properties of the TBP the total number of distinct edges between any two layers in a width-w TBP is at most $w+2$: there are w edges corresponding to the identity permutation, 2 edges corresponding to the transposition of two elements, and $w-2$ edges corresponding to the identity maps for all but the two transposed elements. The $w-2$ last edges overlap with the w edges corresponding to the identity permutation.

Lemma 4.5. $\mathsf{PBP}^w \subseteq \mathsf{SK}^{w+2}$

Proof. Given a PBP^w for a language L of size s, by Lemma 4.4 we know that L also has a width-w TBP. By Remark 1 the underlying DAG for the TBP has at most $w+2$ edges between any two consecutive layers. Using Lemma 4.2 we get a skew circuit of width $w+2$ for L. Note that the size of such a circuit is $O(ws)$ □

Using Barrington's characterization of NC^1 and Lemma 4.5 we get the following: $\mathsf{NC}^1 = \mathsf{BP}^5 = \mathsf{PBP}^5 \subseteq \mathsf{SK}^7$

4.1.3 Skew Circuits to Branching Programs

In this section we start from a skew circuit of bounded width and convert it into a branching program of bounded width. Formally, we prove the following:

Theorem 4.6. *If* C *is a skew circuit of width* w *and length* ℓ *then there is a branching program* P *of width* w *and size* $O(w\ell)$ *computing the same function as* C.

Proof. Recall that in a skew circuit C, AND gates have fan-in 2 and at least one child is an input variable whereas OR gates have arbitrary fan-in and arbitrary predecessors.

Given a skew circuit C of width w and length ℓ_0 we will construct a branching program P of width w that will recognize the same language. Let $G_{i1}, \dots, G_{iw}$ be the gates of C on layer i for $i = 1, \dots, \ell_0$.

Let $X = \{\ell_{i_1}, \ell_{i_2}, \dots, \ell_{i_L}\}$ ($|X| = L$) be the set of layers on which there is at least one input gate. Without loss of generality we assume that in each $j \in [L]$ the gate $G_{\ell_{i_j} w}$ is an input gate. (There may be other input gates as well.)

We will construct a branching program of length $s = L + 2$ and width w. The nodes in the branching program in layer $\ell_{i_j} \in X$ will be called $N_{j0}, \dots, N_{j(w-1)}$. The node N_{00} is the initial node and the node $N_{(s-1)(w-1)}$ will be the target node.

The nodes $N_{11}, \dots, N_{(s-1)(w-1)}$ will by our construction compute the value of the nodes in a layer in X. More formally, for every input x, the gate $G_{\ell_j c}$ in layer ℓ_j of the circuit (and layer j in X) evaluated to 1 iff the node N_{jc} can be reached from the initial node. Since the gate G_{iw} in X is an input gate we will not add corresponding gate in the branching program. We have completely specified the vertex set of the branching program P.

We now describe the edge set of P. We add an edge from $N_{(j-1)0}$ to N_{j0} labeled by 1 for every $1 \leqslant j \leqslant s - 1$. This ensures that all nodes N_{j0} are always reachable from the initial node.

Suppose that the layer ℓ_j and $\ell_j + 1$ are both in X, i.e. $\ell_{j+1} = \ell_j + 1$, then the edges between the nodes in the layer ℓ_j and ℓ_{j+1} in the branching program are easy to state. A node N_{j+1c} is connected to N_{jd} if there is an edge between the corresponding gates $G_{\ell_{j+1}c}$ and $G_{\ell_j d}$. Also the edge in the branching program is labeled by 1 if the gate $G_{\ell_j d}$ is an OR gate, and labeled by the variable x_i (or its negation $\neg x_i$) if $G_{\ell_j d}$ is an AND gate querying x_i, resp. $\neg x_i$. If an OR gate in ℓ_j is connected to an input gate, we generate an edge to N_{j0} labeled by the literal queried by the input gate.

Now assume that the layer ℓ_j is in X and ℓ_{j+1} is the next layer in X and $\ell_{j+1} > \ell_j + 1$. Then in the skew circuit, no input gates occur strictly between the layers ℓ_j and ℓ_{j+1}. This implies that there are no AND gates in the layers $\ell_j + 2, \dots, \ell_{j+1}$. Hence the functions computed by the gates in layer ℓ_{j+1} are ORs of some gates in layer $\ell_j + 1$. In layer gates in layer $\ell_j + 1$ are ORs of either ANDs of gates in layer $\ell_j + 1$ and an input variable or ORs of directly gates in layer $\ell_j + 1$. Therefore, we add the following edges in the branching program: a node $N_{(j+1)c}$

is connected to N_{jd} if the OR function computed by $G_{\ell_{j+1}c}$ has $G_{\ell_j d}$ as one of the inputs. This edge in the branching program is labeled by 1 if this was a direct OR, it is labeled by the variable x_i (or its negation $\neg x_i$) it was an 'OR' of an 'AND' querying x_i (resp. $\neg x_i$).

It is easy to verify by induction on the layers that N_{jc} is reachable from the initial gate if the corresponding gate evaluates true. Finally we add an edge from the node corresponding to the output gate to $N_{(s-1)(w-1)}$. □

Putting together Lemma 4.5 and Theorem 4.6 we get the following corollary:

Corollary 4.7. $\mathsf{NC}^1 = \mathsf{BP}^5 = \mathsf{PBP}^5 = \mathsf{SK}^7$

Remark 4.8. Notice that in the Proof of Theorem 4.6, we use *reachability* crucially. This makes the simulation non-uniform (or at least NL-uniform!).

4.2 Width $\leqslant 7$ skew circuits

In this section we study the structure of the languages in $\mathsf{NC}1$ by investigating properties of skew circuits of width 7 or less. By definition $\mathsf{SK}^i \subseteq \mathsf{SK}^{i+1}$ for $1 \leqslant i \leqslant 6$.

We start by proving that width 2 circuits are not universal.

Lemma 4.9. *A width 2 skew circuit of arbitrary size cannot compute Parity of 2 bits.*

Proof. Let $f = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$. To show that $f \notin \mathsf{SK}^2$, firstly notice that fixing the values of both the variables is necessary and sufficient to make f a constant function. Let C be a SK^2 circuit of minimal length computing f. The output gate of C cannot have a constant as its input: suppose the output is an AND(OR) gate and receives 0 (resp. 1) as input then the circuit computes a constant function. On the other hand, if the output is an AND(OR) gate and receives 1 (resp. 0) as input then such an output gate is redundant, contradicting the minimality of the circuit.

Now, the output cannot be an AND gate because being skew, it can be fixed by fixing one of its inputs, which contradicts the fact that C computes f. Therefore the output gate of C, say g_0, is an OR gate. Let g_1, g_2 be its two inputs. The following cases arise:

a) Both g_1, g_2 cannot be OR since we can collapse such a layer and still compute the same function, contradicting minimality.

b) Both g_1, g_2 cannot be AND: suppose they are both AND gates, the next layer must have at least one input variable, say x_i. Let the other gate on that layer be h_1. If both query the variable, then by setting that variable to 0, we will make the output zero, which will contradict the assumption that C computes f. Suppose one of them, say g_1, does not query x_1 then the output of the circuit is $g_0 = (h_1 \wedge x_i) \vee h_1 = h_1$. This contradicts the minimality of the circuit.

c) Even one of g_1, g_2 cannot be a variable, else by setting that variable we can fix the output of the circuit.

d) Due to cases (a), (b) and (c) we are only left with a case in where one of the inputs to g_0 is an AND and the other is OR (say g_1 is AND and g_2 is OR). In the layer just below this layer, there will have to be an input gate in a minimal circuit and this will have to be queried by both g_1, g_2. This variable can now be fixed (to make $g_1 = 1$ and therefore $g_0 = 1$) so that the output of the circuit is fixed.

Therefore, no width 2 skew circuit computes f. This proves that width 2 circuits are not universal. □

Recall that a k-*DNF of size* s *on* n *variables* is an OR of s terms, where each term is an AND of at most k literals from $\{x_1, x_2, \ldots, x_n, \neg x_1, \neg x_2, \ldots, \neg x_n\}$. Similarly, k-*CNF of size* s *on* n *variables* is an AND of s clauses, where each clause is an OR of at most k literals from $\{x_1, x_2, \ldots, x_n, \neg x_1, \neg x_2, \ldots, \neg x_n\}$.

Lemma 4.10. *Let* f *be a* k-*DNF of size* s *on* n *variables. Then* f *has a width-*3 *skew circuit of length* $O(sk)$.

Proof. A term is an AND of at most k literals and hence can be computed by a skew circuit of width 2 and length $O(k)$. Suppose $C_1, C_2, \ldots, C_s$ are width-2 skew circuits of length $\ell_1, \ell_2, \ldots, \ell_s$, respectively computing the s terms in the k-DNF of size s. We need to compute an OR of these. This can be done using the one extra width: stagger circuits $C_1, C_2, \ldots, C_s$ one after the other. This requires width 2 and length $L := k \cdot \sum_{i=1}^{s} \ell_i = O(sk)$. Now add one OR gate per layer alongside these C_is. Let us call them $g_1, g_2, \ldots, g_L$. Feed output of g_i to g_{i+1} for all $1 \leqslant i \leqslant L$ and feed output of C_i to g_{ℓ_i+1}. It is easy to see that g_{L+1} computes f. (See Figure 4.1.) □

As any Boolean function on n variables has an n-DNF of size at most 2^n, we get the following corollary.

Corollary 4.11. *Let* $f : \{0,1\}^n \to \{0,1\}$. *Then* f *can be computed by width-*3 *skew circuit of length* $O(n2^n)$, *i.e. width-*3 *skew circuits are universal.*

Lemma 4.12. *Let* f *be a* k-*CNF of size* s *on* n *variables. Then* f *has a width-*3 *skew circuit of length* $O(sk)$.

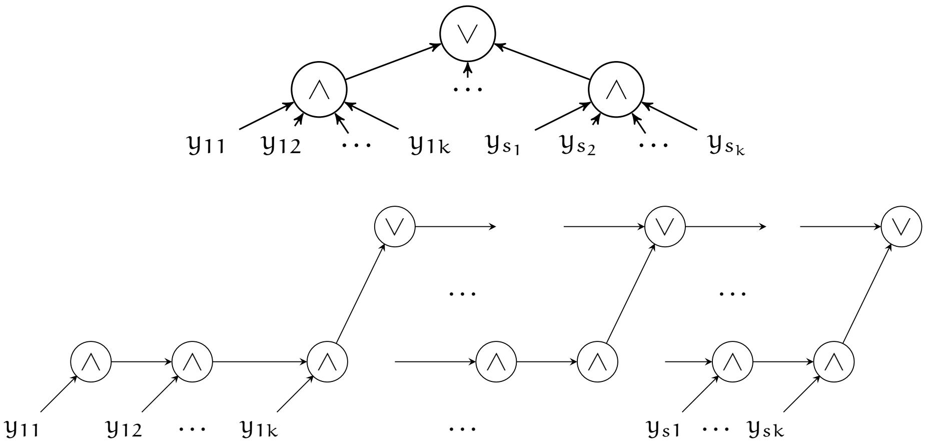

FIGURE 4.1: Width 3 skew circuits for DNFs

Proof. Note that in a CNF, the top AND gate gets clauses as inputs. That is, the AND gate is not skew. However, it is still possible to get a skew circuit for CNFs. We prove this by induction on the number of clauses, i.e. s. The base case is $s = 1$. This is just an OR of literals, which is computable by width-2 skew circuit of length $O(k)$. Let $f_i(x_1, \ldots, x_n) = C_1 \wedge \ldots \wedge C_i$ be computable by width-3 skew circuit of length $O(ik)$. Now $f_{i+1} = f_i \wedge C_{i+1}$ (where $C_{i+1} = x_{j_1} \vee x_{j_2} \ldots \vee x_{j_k}$) is computed as follows: $f_{i+1} = (\ldots((f_i \wedge x_{j_1}) \vee (f_i \wedge x_{j_2}) \ldots (f_i \wedge x_{j_k}))\ldots)$. Note that we need width 1 each for the f_i, the AND gate and the input variable. (Even though we require width 3 to compute f_i, after the computation, it just requires width 1 to carry around the value of the function to the next stage). (See Figure 4.2.) $\square$

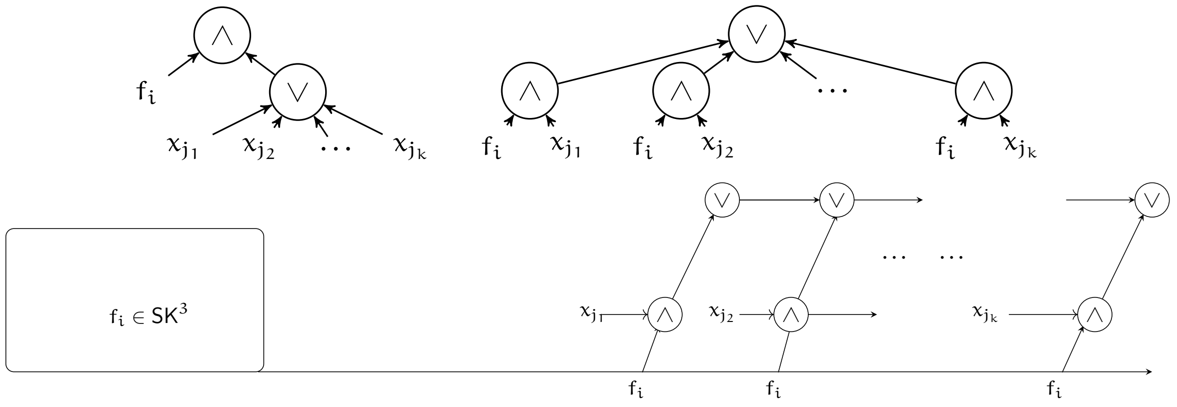

FIGURE 4.2: Width 3 skew circuits for CNFs

Definition 4.13. (Approximate Majority) Approximate Majority $\mathsf{ApproxMaj}_{a,n} : \{0,1\}^n \to \{0,1\}$ is the promise problem defined as:

$$\mathsf{ApproxMaj}_{a,n}(x) := \begin{cases} 0, & \text{if } x \text{ has Hamming weight at most } a \\ 1, & \text{if } x \text{ has Hamming weight at least } n - a \end{cases}$$

By a result of Viola [Vio09], it is known that Approximate Majority is computable by P-uniform depth-3 circuits of polynomial size with alternating AND/OR layers with the output gate being an OR gate. This along with Lemma 4.12 above yields:

Corollary 4.14. $\mathsf{ApproxMaj}_{a,n}$ *has skew circuits of width* 4 *and polynomial length.*

In a long line of work in the 80's starting with Furst, Saxe, Sipser [FSS84], Ajtai [Ajt83], Yao [Yao85], Hastad [Has86], Razborov[Raz87] and Smolensky[Smo87] show that Parity does not have constant depth circuit of polynomial size. This also implies that Parity does not have polynomial size CNFs or DNFs . However, we now show that Parity has skew circuits of width 4 and polynomial size.

Lemma 4.15. *Parity on* n *variables has a skew circuit of width* 4 *and length* $O(n)$.

Proof. This is an easy observation which comes from the fact that Parity has a branching program of width 2 and length $O(n)$. This fact along with part 3 of Lemma 2.5 proves the result. (Figure 4.3 shows the width 4 skew circuit for Parity.) □

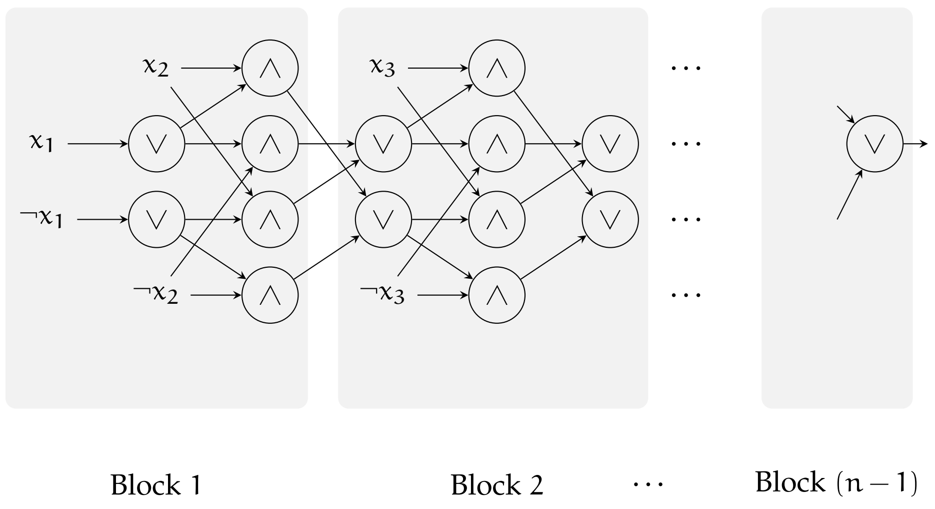

Figure 4.3: Width-4 skew circuit for Parity

4.3 Parity and SK^3

In this section we prove that Parity does not have polynomial length width-3 skew circuits. As Parity has width 4 skew circuits of linear length (Lemma 4.15), this separates SK^3 from SK^4.

Theorem 4.16. *Any* SK^3 *circuit computing the Parity function on* n *bits requires size* $2^{\Omega(n)}$.

In order to prove this we first show that any width three skew circuit computing Parity can be converted into a normal form. We then show that any polynomial sized circuit of that normal form cannot compute Parity.

Lemma 4.17. *Let C be any width 3 skew circuit with size $s(n)$ computing* $\textsc{Parity}_n$*. The circuit C can be converted into another circuit D such that D computes Parity of at least $n-4$ bits and has the following structure:*

1. *The top gate of D is an OR of at most $3s(n)^2$ disjoint skew circuits, say $C_1, C_2, \dots, C_m$, where $m \leqslant 3s(n)^2$.*
2. *The sum of sizes of all C_is is at most $O(s(n)^3)$*
3. *At most three of these circuits have width 3 and all the other have width at most 2.*

Lemma 4.18. *Let D be a circuit satisfying properties 1,2,3 from Lemma 4.17 and computing* $\textsc{Parity}_n$*. Then there exists a constant c such that the size of D is at least $2^n/n^c$.*

Using Lemma 4.17 and Lemma 4.18, the lower bound for Parity follows.

4.3.1 Proof of Lemma 4.17

Let C be any width three circuit computing Parity of n bits of size $s(n)$. The top gate of C cannot be AND. This is because, by fixing the input wires of the AND gate, we could fix the output of the circuit, however, Parity of n bits cannot be fixed by fixing $< n$ bits. Therefore, we can assume that the top gate is an OR gate.

Proposition 4.19. *Let C be any width three skew circuit computing Parity of n bits. Let k be the highest layer in C consisting of three AND gates, say g_k^1, g_k^2, g_k^3. We can convert this into another width 3 skew circuit D computing Parity of at least $n-2$ bits such that no layer of D contains three AND gates.*

Proof. Let k be the highest layer (closest to the output gate) with all three AND gates, say g_k^1, g_k^2, g_k^3. Note that, due to skewness of the circuit, the layer $k+1$ cannot have any AND gates. If any one of the gates at layer k has fan-in 1, then we can replace that gate by an OR gate. Therefore, we will assume that each gate has fan-in 2. Let gates at layer below, i.e. $k-1$, be $g_{k-1}^1, g_{k-1}^2, g_{k-1}^3$. As we are dealing with skew circuits, at least one of $g_{k-1}^1, g_{k-1}^2, g_{k-1}^3$ is an input gate. Suppose there is only one input gate, then all gates at layer k must read this input bit. And therefore, the three gates at layer k can be fixed by fixing this input bit. If there are two input gates, then either both read the same literal (a variable and its negation) or they read two distinct input bits. In the latter case, by fixing the two distinct

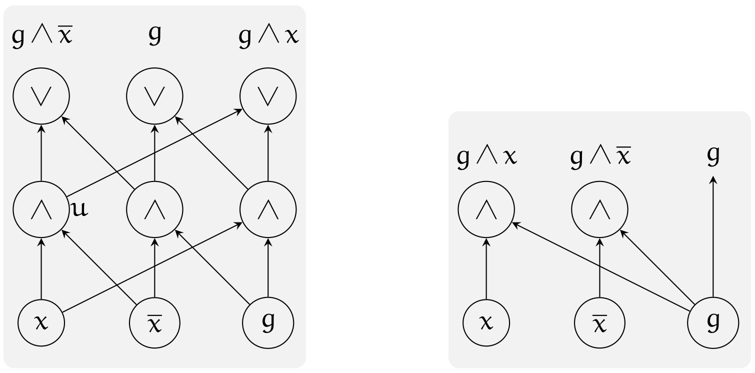

FIGURE 4.4: Normal form for width-3 circuits computing Parity

bits, all the three gates can be fixed. Finally, if the two variables being read are x and $\neg x$ then fixing x may not fix all three gates. For example, suppose $g^1_{k-1} = x, g^2_{k-1} = \neg x, g^3_{k-1} = g$, and $g^1_k = (g \text{ AND } x)$ and $g^2_k = (g \text{ AND } \neg x)$. Then fixing x to any value will not fix both g^1_k and g^2_k. Figure 4.4 (the left part) depicts an instance of this case where the $(k+1)$-th layer above has three OR gates.

This case can be handled as follows: if all three gates at layer-k compute distinct functions, then one of them computes $x \wedge \neg x$ and hence can be replaced by the constant 0, otherwise two gates are forced to compute the same function and hence one of them can be removed. We have not fixed any bits in this case, and we can now move on to the next 3-ANDs layer and repeat the same process.

Therefore, any layer consisting of three AND gates in a width-3 skew circuit computing parity can be completely removed from the circuit by fixing at most 2 bits.

□

Let us now assume that we have a minimal width-3 circuit D which computes parity of at least $n-2$ bits in which no layer consists of three AND gates.

Let $G = (V, E)$ denote the DAG underlying the circuit D. Let $X \subseteq V$ denote the subset of gates which have a path to the top gate via only OR gates. Note that all the vertices in this set are themselves OR gates. We refer to the set of vertices X as ORSET.

Let $X_{in} \subseteq V \setminus X$ denote the set of vertices which have an edge incident from it to some vertex in X. Similarly, let $X_{out} \subseteq V \setminus X$ denote the set of vertices which have an edge incident to them from some vertex in X (Notice that a vertex can belong to both X_{in} and X_{out}. See for example vertices C_4 and C_m in Figure 4.5).

(a) Disconnect all edges incident to the set X_{out} from X. Let the new dangling wires be labeled with the constant 0 input.

(b) If after step (a) any AND gates receives constant input 0 then delete the gate and if any OR gate receives constant input 0 then delete this input of the OR gate.

(c) In the graph obtained after steps (a) and (b), consider $V \setminus X$. This disconnects the DAG G and gives rise to some connected components, say $X_1, X_2, \ldots, X_\ell$. For each $i \in [\ell]$ and edge (u, v) such that $u \in X_i$ and $v \in X$ let $C_{i,u}$ be the subgraph (DAG) of X_i with u as its sink.

Proposition 4.20. *The step (a) above does not change the function computed by the circuit.*

Proof. Let u be a gate in X which feeds into a gate v in X_{out}. Step (a) above disconnects u from v and feeds value 0 to v. Suppose g_u, the OR gate corresponding to u evaluates to 1, then the circuit will evaluate to 1 irrespective of all other gates. On the other hand, if it evaluates to 0, then that is the value that we have fed into v and therefore, the modification in Step (a) does not change the output of the circuit in either case.

However, we also need to establish that such local surgeries will not disconnect the circuit and affect the final output computed. That is, for every x such that $C(x) = 1$, there is a path to the output gate which evaluates to 1. This implies that there is a path with a single non-ORSET-to-ORSET transition, and all inputs coming into the path are 1. Let g be the lowest ORSET node on this path. Note that, in D, g must evaluate to 1. To prove this, assume for the sake of contradiction that g does not evaluate to 1 in D. Consider the lowest node on path below g that flips from 1 to 0 in D (call it g'). g' can't be AND, because its skew input hasn't changed and its non-skew input isn't in ORSET. If g' is OR, then all its children must have flipped, even the child on the path, but this is a contradiction because we assumed that g' is the lowest gate that flipped. Otherwise, if ORSET node is 1 in D, then $D = 1$. This is because connections within ORSET are intact. □

Note that $V \setminus X$ is partitioned by X_is. Therefore, $\ell \leqslant s(n)$. Also, $\cup_{i=1}^{m} X_i \subseteq V \setminus X$. Therefore, $\sum_{i=1}^{m} |X_i| \leqslant s(n)$. The number of edges in G is at most $3s(n)$. Therefore, the total number of edges between any X_i and X is at most $3s(n)$. Therefore, the number of circuits $C_{u,i}$ is at most $s(n)^2$ and each such circuit is of size at most $s(n)$. This proves parts $1, 2$ of Lemma 4.17.

We will now prove part 3 of Lemma 4.17. In the circuit D, obtained after steps (a) and (b), let L_r be the last layer (farthest from the output gate) in which the ORSET is present. Let g and g' be the two other gates in L_r. Note that there are no edges incident from the ORSET X into $V \setminus X$ in D after steps a) and b).

Firstly, we observe the following about our circuit D:

(Obs1) Inputs cannot feed straight into ORSET, because if they do then parity can be always set to 1 by setting this particular input.

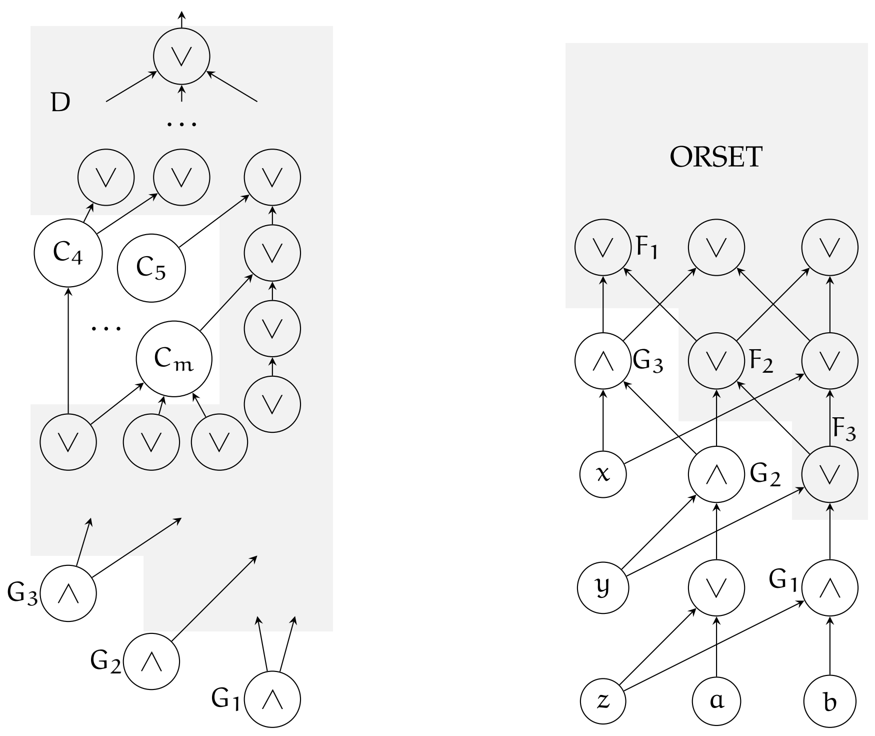

Figure 4.5: Width-3 circuits

(**Obs2**) No OR gate outside of the ORSET can feed straight into the ORSET, because if they do, by the definition of ORSET, they would be part of the ORSET. Hence OR gates outside of the ORSET can be connected to the ORSET necessarily via AND gates.

(**Obs3**) If a gate g feeds straight in to the ORSET via wire w, and simultaneously to a non-ORSET gate g' via wire w', we can disconnect w' if g' is an OR gate – because if g evaluates to 1, the parity is set to 1 anyway via the ORSET, otherwise g evaluatess to 0, and this does not affect the output of g' since it is an OR gate. On the other hand, if g' is an AND gate, then we cannot disconnect w' like in the previous case. To see this, let us assume the other input to the AND gate is a literal x (set to 1) and let g evaluate to 0. Now if we cut out w', g' will output 1 whereas with w' present, the output of g' will be 0. However in this case, notice that since g' is an AND gate (and hence is skew), we can fix x to 0, and eliminate g'.

Therefore, for all AND gates g (using **Obs2**) such that g feeds in to ORSET via wire w and also in to a non-ORSET gate g' via wire w', we remove w' and propogate 0 via w' into g' using **Obs3** above.

To this end, we look at various cases based on the number (and further cases within them depending on the type) of gates in L_r. Let us call the gates in L_r g_1, g_2, g_3.

1. There are 3 OR gates in L_r, all of them belonging to the ORSET (**OR, OR, OR**): In this case there can be at most three points of attachment, but by Proposition 4.19 and **Obs1** above, there can be only at most two SK^3 circuits that can be attached to the ORSET.

2. There are 2 OR gates from the ORSET in L_r: There are two further cases:

   a) The third gate is an AND gate (**OR, OR,** AND): In this case, this gate should query a variable. Set this variable to 0 and remove the AND gate. Feed 0 to the gates that the AND gate was feeding in to. Now there are at most two SK^3 circuits attached to the ORSET (namely at the two OR gates in L_r). The remaining circuits attached to the OR gates are SK^2 circuits.

   b) The third gate is an OR gate (**OR, OR,** OR): Let the gates in the layer below be A, B, C, then there could be further cases based on different ways in which A,B,C feed in to the three OR gates in L_r. For example, if $g_1 = A, g_2 = B, g_3 = C$ makes L_r a redundant layer and hence here we can remove this layer while computing the same function. If $g_1 = A, g_2 = B \vee C, g_3 = C$, then we can g_3 is redundant since $g_3 = 1$ implies $g_2 = 1$. If $g_1 = A \vee B$, $g_2 = B \vee C$ and $g_3 = C \vee A$, then notice that if one of them evaluates to 1, a 1 is fed in to the ORSET and the circuit will eventually output 1. So remove the third (non-ORSET) OR gate and propogate 0 through the gates that this OR gate was feeding in to. This does not change the output of the circuit. The other cases can be analyzed similarly and they can be seen to yield at most two SK^3 circuits being attached to the ORSET.

3. There is one OR gate from the ORSET in L_r: Let us call the circuitry parallel to the ORSET, which is of width 2, C_1, and the circuitry below the ORSET of width 3 (starting from the layer below L_r) C_2. We will now try to disconnect C_1 from C_2. Firstly, notice that in the circuit we have currently, ORSET gates feed in to ORSET gates only. Hence, it suffices to analyze C_1 so that we can disconnect it from C_2. Furthermore, any gate that feeds in to the ORSET (in our case, they are always AND gates), do not feed in to any other gate in the circuit (since we have disconnected such wires and fed 0 using **Obs3**. We can further simplify by noting that if there is a layer in which both $g \wedge x$, $g \vee x$ are being computed, then the only possible function computed by a width-2 circuit using these two functions can be the OR function (the AND gates are skew and hence need at least one of the two inputs to be variables), which is $(g \wedge x) \vee (g \vee x) = (g \vee x)$. Hence every occurence of such a layer in C_1 can be simplified to compute only $g \vee x$. All the different cases that arise by having different combinations of width-2 circuitry that forms C_1 with the one OR gate in L_r can be analyzed to show that by fixing at most 2 more bits we an ensure that at most three width-3 circuits feed in to the ORSET.

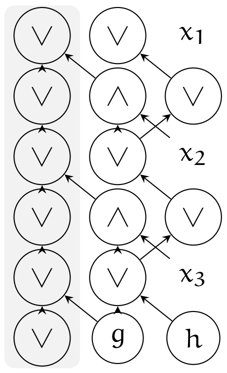

FIGURE 4.6: The case of one exactly one OR gate in L_r

The only case that seems to not simplify is the one given by Figure 4.6 where by cascading the pattern in the width-2 circuitry C_1, it seems to require as many variables be set as the length of the ORSET in order to be set to a constant.

However, we can handle this case by noting that this whole circuit can be written as an OR as $(g \vee ((g \vee h) \wedge x_3) \vee (g \vee h) \wedge x_2) = g \wedge (1 \vee x_3 \vee (h \wedge x_3) \vee (h \wedge x_3)) = g \wedge 1 = g$. Hence this reduces to the case of a single width-3 circuit being attached to the ORSET.

Finally, analyzing all the cases, we have that by fixing at most two more input bits, we can make sure that our circuit D is an OR of at most $3s(n)^2$ disjoint skew circuits out of which at most three have width-3 and the rest have width-2. This completes Part 3 of Lemma 4.17.

4.3.2 Proof of Lemma 4.18

Let D be the circuit given by Lemma 4.17. Let n_0 denote the number of unfixed variables. Let $C_1, C_2, \ldots, C_m$ be circuits given by Lemma 4.17. We know that at most three circuits among these have width 3. Let us assume without loss of generality that the two circuits are C_1, C_2, C_3. The output gates of these circuits are AND gates, say G_1, G_2, G_3, respectively. These being skew circuits, all but one of the inputs of the AND gates are input gates. Notice that since D is a minimal circuit computing Parity[1], and Lemma 4.17 has fixed at most 2 bits to get D from C, we can go further down from G_1 in the circuit D to find a suitable variable to fix, even if the input to G_1 is already fixed to 1 previously[2]. More specifically, let G_1 be of the form $1 \wedge G_1'$. We recurse on G_1' to find an AND/OR gate querying a unset literal and set it appropriately so that G_1' evaluates to 0 (notice that G_1' can be written as $\vee_{i=1}^{a}(H_i \wedge x_{i_1} \wedge x_{i_2})$ where x_{i_1} and x_{i_2} are distinct literals, H_i is a subcircuit of C_1 and by minimality of the circuit a can be at most a constant); that such literals should exist, is

[1]Parity of an n-bit string is not fixed until $n-1$ bits of the string are fixed. An important consequence of this property of the Parity function is that if a circuit claims to compute it, then the output of the circuit is not fixed until $n-1$ input variables are fixed.

[2]this can happen because for example G_1 could query a variable $\neg x$ when x was already set to 0 in Lemma 4.17

guaranteed by the sensitivity of the parity function and the minimality of D. Note that we cannot set G_1' to 1 given that the other input to G_1 is already 1, because this would set the output of the circuit to 1 without setting all the variables. By symmetry, this argument holds for G_2 and G_3 as well. Hence from now on, we will concern only with the cases where the variables queried by G_1, G_2, G_3 are not already fixed by Lemma 4.17. The following cases arise:

1. G_1, G_2, G_3 **query distinct literals**: Then all three of them can be set to 0 by setting exactly 3 variables.
2. G_1 **queries** x, G_2 **queries** $\neg x$, G_3 **queries** y: Set y to 0. Handle the other two AND gates via Proposition 4.21.
3. G_1, G_2 **query** x, G_3 **queries** $\neg x$: Set x to 0. This leaves us to handle G_3 of the form $G_3 = G_3' \wedge 1$. This is handled similar to the case of $G_1 = 1 \wedge G_1'$ discussed above.

We are left with the case 2 above. We prove:

Proposition 4.21. *By fixing at most two variables both G_1 and G_2 can be set to zero.*

Proof. Let $G_1, G_2, \ldots, G_m$ be top gates of the circuits $C_1, C_2, \ldots, C_m$. Note that G_is are AND gates because if they were OR gates they would have been in the ORSET. Say they appear on layers $k_1, k_2, \ldots, k_m$ in the original width 3 circuit C. If both C_1, C_2 are width 3 circuits, then it is easy to see that $k_1 \leqslant k_i$ for $3 \leqslant i \leqslant m$ and $k_2 \leqslant k_i$ for $3 \leqslant i \leqslant m$. Let us assume without loss of generality $k_1 \leqslant k_2$, i.e the gate G_1 appears on a lower layer (closer to Layer 1) than G_2.

Let x_i, Y_1, Y_2 be the gates on the layer $k_1 - 1$. As G_1 is an AND gate, it must query one variable. Let that be x_i. Let the other input to G_1 be Y_1 without loss of generality.

Now note that Y_2 must be connected to the ORSET via G_2. If it is not connected via G_2 then C_2 cannot have width 3 (since $k_2 > k_1$ and ORSET extends all the way upto $k_1 + 1$, therefore, the circuit C_2, which starts at k_2 and is disjoint from the ORSET and disjoint from the path which connected Y_2 to the ORSET, can have width at most 2.)

Our proof for the proposition is a case analysis of the various settings for x_i, Y_1, and Y_2.

In our case, $y = \neg x_i$. Here, by minimality of the circuit we can assume that the output of the entire circuit can be written as $\bigvee_{i=3}^{O(m)} C_i \vee (Y_1 \wedge x_i) \vee (Y_2 \wedge \neg x_i)$. To handle this we need to look at the next layer below $k_1 - 1$ that reads an input variable. Say the variable read is x_j and the layer number is k_0. Let the other two gates on this layer be Z_1, Z_2. Here again, by observing that there cannot be any redundant gates in the minimal circuit (and using distributivity of AND/ORs) it follows that $x_j \neq x_i$. If either of Y_1, Y_2 is an AND gate, then $k_0 = k_1 - 1$ else both are OR gates.

Therefore, Y_1, Y_2 are ORs over $\{(x_j \wedge Z_1), Z_1, Z_2\}$. (The analysis for the case of ORs over $\{(x_j \wedge Z_2), Z_1, Z_2\}$ is symmetric. The case of ORs over simply $\{x_j, Z_1, Z_2\}$ cannot arise as otherwise Y_1 or Y_2 will be connected to the ORSET directly, but this is not possible as we have that both are connected to the ORSET via ANDs.)

(i) If both Y_1 and Y_2 do not query $x_j \wedge Z_1$, then the AND is redundant, which is not possible in a minimal circuit.

(ii) If Y_2 (Y_1) queries only $x_j \wedge Z_1$. Then by setting $x_i = 0, x_j = 0$ ($\neg x_i = 0, x_j = 0$, respectively) we can set both G_1 and G_2 to zero.

(iii) If Y_2 queries $(x_j \wedge Z_1)$ and Z_1, but not Z_2. Then again the AND is redundant. (The case that Y_1 queries $(x_j \wedge Z_1)$ and Z_1, but not Z_2 is similar.)

(iv) If Y_2 queries $(x_j \wedge Z_1)$ and Z_2, but not Z_1 and Y_1 queries Z_2 (Y_1 may query Z_1 and $x_j \wedge Z_1$ as well) then $G_1 \vee G_2 = \big(\big((x_j \wedge Z_1) \vee Z_2\big) \wedge x_i\big) \vee (Z_2 \wedge x_i) = \big((x_j \wedge Z_1) \wedge x_i\big) \vee (Z_2 \wedge x_i) \vee (Z \wedge \neg x_i) = \big((x_j \wedge Z_1) \wedge x_i\big) \vee Z_2$, i.e. Z_2 directly feeds into the ORSET, but this contradicts minimality. (The case that Y_1 queries $(x_j \wedge Z_1)$ and Z_2, but not Z_1 and Y_2 queries Z_2 is similar).

(v) If Y_2 queries $(x_j \wedge Z_1)$ and Z_2, but not Z_1 and Y_1 queries Z_1 but nothing else then Y_1 is redundant.

This exhausts all the cases and finally by setting at most 2 variables we have managed to eliminate both G_1, G_2. This finishes the proof of the proposition.

□

Let N denote the number of variables which were not set. Here, $N \geqslant n_0 - 3$. The new circuit, say D', is now an OR of $C_4, \ldots, C_m$ and by our assumption, it computes Parity of N variables. We will show that OR of polynomially many polynomial size width-2 skew circuits cannot compute Parity.

Let us fix some notation. Let $L_\oplus$ denote the 1 set of parity, i.e. $L_\oplus = \{x \in \{0,1\}^N \mid \mathrm{Parity}(x) = 1\}$. We know that $|L_\oplus| = 2^{N-1}$. For any Boolean circuit C, let L_C denote $\{x \in \{0,1\}^N \mid C(x) = 1\}$. Note that as D' above is an OR of $C_4 \ldots, C_m$, we have $L_{D'} = \cup_{i=4}^m L_{C_i}$.

Definition 4.22. We say that a Boolean circuit C *α-approximates* a function $f : \{0,1\}^n \to \{0,1\}$ if the following conditions hold:

- $\forall x \in \{0,1\}^n$, if $f(x) = 0$ then $C(x) = 0$, i.e. C has no false positives.
- The ratio of $|\{x \mid C(x) = 1\}|$ to $|\{x \mid f(x) = 1\}|$ is at least α

For the sake of contradiction we have assumed that D' computes parity of N bits. Assuming this and from the fact that $L_{D'} = \cup_{i=4}^{m} L_{C_i}$, we get that there exists an $i \in \{4, \dots, m\}$ such that C_i $1/m$-approximates parity of N bits. We will now prove that no such C_i exists, which will give us the contradiction. Formally, we prove the following:

Claim 4.23. Let D' and $C_4, \dots, C_m$ be defined as above. There does not exists $i \in \{4, \dots, m\}$ such that C_i $1/m$-approximates Parity of N bits.

Proof. Suppose there exists a C_i which $1/m$-approximates parity of N bits. Recall that C_i is a width 2 skew circuit. Let the last layer be L and the first layer be 1. Let $\ell_{i_1}, \ell_{i_2}, \dots, \ell_{i_t}$ be the layers in which there is one input gate, with ℓ_{i_1} being closest to layer 1 and ℓ_{i_t} being closest to layer L. (Note that, we can assume without loss of generality that layer 1 is the only layer which has two input gates.) Let the variables queried by these gates be $x_{i_1}, x_{i_2}, \dots, x_{i_t}$, respectively. Let $h_{i_{t+1}}$ denote the output gate in layer L. Similarly, let h_{i_1} be the gate in layer ℓ_{i_1} (other than the input gate), h_{i_2} be the gate in layer ℓ_{i_2} (other than the input gate) and so on till h_{i_t} be the gate in layer ℓ_{i_t} (other than the input gate).

As there are no NOT gates in the circuit, $h_{i_{j+1}}$ is a monotone function of x_{i_j}, h_{i_j} for every $1 \leqslant j \leqslant t$. There is a unique value of x_{i_j}, say $b_{i_j} \in \{0,1\}$, such that by setting $x_{i_j} = b_{i_j}$, $h_{i_{j+1}}$ becomes a non-trivial function of h_{i_j}. (This is because, there are at most 6 different monotone functions on two bits, two of which cannot occur in a minimal circuit. And the other four (AND, OR, NAND, NOR) have this property.)

Note that, the setting of $x_{i_t} = b_{i_t}$ will not fix the value of h_{i_t}. Suppose h_{i_t} gets fixed due to this setting. In that case, value of $h_{i_{t+1}}$ will also get fixed. Suppose the value of $h_{i_{t+1}}$ becomes 1, then for all settings of $x \neq x_{i_t}$, $h_{i_{t+1}}$ will continue to have value 1. But we have assumed that C_i has no false positives. Therefore, this is not possible. On the other hand, if the value of $h_{i_{t+1}}$ gets fixed to 0, then for all settings of variables $x \neq x_{i_t}$ the circuit will output 0. That is, for 2^{N-1} different inputs the circuit will output 0. However, we have assumed that the circuit outputs 1 for at least $2^{N-1}/m$ many inputs.

Assuming $x_{i_t} = b_{i_t}$ and $x_{i_t} \neq x_{i_{t-1}}$, we will repeat this argument for $x_{i_{t-1}}$. Let $x_{i_{t-1}} = b_{i_{t-1}}$ be the setting of $x_{i_{t-1}}$ which makes h_{i_t} a function of $h_{i_{t-1}}$. Suppose this setting of $x_{i_{t-1}}$ fixes h_{i_t} then that will in turn fix $h_{i_{t+1}}$. As before, to avoid false positives, the value of $h_{i_{t+1}}$ cannot be fixed to 1. And to ensure that the circuit evaluated to 1 on at least $2^{N-1}/m$ inputs, it cannot be fixed to 0.

In this way, we can repeat the argument for k distinct variables as long as $k < (N-1) - \lceil \log m \rceil$. Let k_0 be such that $k_0 = \omega(\log m)$ and $k_0 < (N-1) - \lceil \log m \rceil$. We fix k_0 distinct variables as above. But now note that any other setting of these k_0 variables fixes the value of $h_{i_{t+1}}$ to 0. Therefore, the circuit can be 1 on at most $O(2^{N-k_0})$ inputs. But this contradicts our assumption that $h_{i_{t+1}}$ evaluated to 1 on at least $2^N/m$ inputs from $L_\oplus$.

$\square$

4.4 Discussion

Plenty of gaps remain in our understanding of bounded width branching programs and skew circuits:

- Can width 4 Permutation Branching Programs compute the AND function? In this regard, it is known [Bar85] that width 3 permutation branching programs require exponential size to compute the AND function (It is known that for width 5 and above, permutation branching programs are as powerful as branching programs).
- At what width are skew circuits capable of computing the Majority function? It is clear from Barrington's work that width 7 suffices. Can we do better?
- Can we prove an exponential lower bound for width-3 or width-4 skew circuits computing the Majority function? Note that even though Parity reduces to Majority and we have shown that Parity requires exponential size width 3 skew circuits, we do not know of a reduction between Parity to Majority that can be implemented by width 3 skew circuits. Towards this end, the first step might be to obtain a normal form for any width 3 circuit computing Majority, like the one we have for Parity in Lemma 4.17.
- Can we show the hierarchy between width 4 to width 7 skew circuits is strict under a plausible Complexity-theoretic assumption (say for e.g. $\mathsf{NC}^1 \not\subseteq \mathsf{TC}^0 \not\subseteq \mathsf{ACC}^0$)?

Chapter 5

Linear Algebra in Bounded Treewidth

In this chapter, we will study the complexity of computing certain linear algebraic invariants of a matrix using tools from graph theory. When the underlying digraph/bipartite graph is bounded treewidth/planar, we call their adjacency matrices *bounded treewidth matrices*/*planar matrices*.

As elucidated in Chapter 1 (Section 1.3), a combinatorial interpretation of the determinant of a square matrix is that of a function that counts the signed sum of cycle covers of the digraph underlying the matrix. The determinant is complete for the class GapL when the matrices represent general dense graphs. A few years ago, Datta, Kulkarni, Limaye and Mahajan [DKLM10] had proved that determinant computation is GapL-hard for planar matrices. We study the analogous question for bounded treewidth graphs. Note that, bounded treewidth graphs and planar graphs are incomparable, i.e., there are planar graphs that have unbounded treewidth, and there are classes of bounded treewidth graphs that are non-planar. We prove that determinant computation on bounded treewidth matrices is complete for Logspace. We use this Logspace algorithm as a subroutine to compute several related linear algebraic invariants of bounded treewidth matrices and also to count certain substructures in bounded treewidth graphs.

5.1 Determinant Computation

Given a square $\{0, 1\}$-matrix A, we can view it as the biadjacency matrix of a bipartite graph H_A. The permanent of this matrix A counts the number of perfect matchings in H_A, while the determinant counts the signed sum of perfect matchings in H_A. If G is a bounded treewidth graph then we can count the number of perfect matchings in G in L(See Theorems 2.12, 2.13) (see also [DDN13]).

The determinant of a $\{0,1\}$-matrix G reduces to counting the number of paths in another graph G_{MV} (see [MV97]). While counting s,t-paths is MSO-expressible and hence on bounded treewidth graphs in L via Theorems 2.12, 2.13 (see also [DDN13]), the graph G_{MV} obtained by such a reduction need not be bounded treewidth even if G is bounded treewidth.

However, we can also view A as the adjacency matrix of a directed graph G_A. If G_A has bounded treewidth[1], and if we can write an MSO-formula that evaluates to true on any cycle cover of G_A then we have a way of computing the determinant of A by invoking Theorems 2.11, 2.12, 2.13. This is precisely what we do:

Lemma 5.1. *There is an* MSO*-formula* $\phi(X,Y)$ *with free variables* X,Y *that take values from the set of subsets of vertices and edges respectively, such that* $\phi(X,Y)$ *is true exactly when* X *is the set of heads of a cycle cover* Y *of the given graph.*

Proof. We write an MSO formula ϕ on free variables X,Y, where $Y \subseteq E$ and $X \subseteq V$, such that ϕ evaluates to true on any set of heads of a cycle cover S. The MSO predicate essentially verifies that the subgraph induced by Y indeed forms a cycle cover of G. To this end, we have to check that every vertex in the graph $C = (V(Y), Y)$ has indegree and outdegree exactly equal to two and $V(Y) = V(G)$, here we crucially use the notion of the *head* of a cycle. Recall (from Definition 2.20) that the head of a cycle is the least numbered vertex in the cycle from which every other vertex in the cycle is reachable. This necessitates that there we enforce a total order on the vertices of G. Note that for Theorem 2.12 to work, any relation used in the formula also has to have bounded treewidth (which a total order is not). We get around this issue by constructing a total order NXT^* which can be inferred from the partial order NXT by locally adding vertices to the bags of the tree decomposition (See Lemma 5.2).

Our MSO formula is of the form [2]:

$$\phi(X,Y) \equiv (\forall v \in V)(\exists! h \in X)[\mathsf{DEG}(v,Y) \wedge \mathsf{PATH}(h,v,Y) \wedge (\mathsf{NXT}^*(h,v) \vee (h = v))]$$

where,

1. $\mathsf{DEG}(v,Y)$ is the predicate that says that the in-degree and out-degree of v (in the subgraph induced by the edges in Y) is 1. $\mathsf{DEG}(v,Y)$ is MSO-definable : We have to assert that for every vertex v in the induced graph $G[Y]$, there are unique vertices v', v'' such that (v', v) and (v, v'') are the only edges incident to v in the graph $G[Y]$[3]. Writing

[1] We say a directed graph has treewidth k if the underlying undirected graph has treewidth k.

[2] Note that since we require that for a given X,Y, every $v \in V$ has a unique $h \in X$, our formula is not monotone, i.e., If $X \subseteq X'$ are two sets of heads then if $\phi(X,Y)$ is true doesn't imply $\phi(X',Y)$ is also true (consider vertices in $X' \setminus X$, since $X' \subseteq X$, they will have two different h, h' such that the PATH and NXT^* predicates are true contradicting uniqueness of h

[3] We introduce new notation here: Given a graph $G = (V,E)$, and a set of edges $Y \subseteq E$, we define $G[E]$ to be the graph induced by E on the vertices $V(E) \subseteq V(G)$.

this in MSO:

$$\mathsf{DEG}(v, Y) \equiv \exists! v', v''[E(v', v) \wedge E(v, v'')]$$

and the predicate that there exists a unique element p in a set X satisfying a property $\mathcal{P}$ can be expressed as

$$\exists p \in X, [\mathcal{P}(p) = 1] \wedge \forall q \neq p[\mathcal{P}(q) = 0]$$

2. $\mathsf{PATH}(h, v, Y)$ is the predicate that says that there is a unique path from h to v in the graph induced by edges of Y. $\mathsf{PATH}(x, y, Y)$ is MSO-definable : We have to assert that there is a minimal subset of edges $P \subseteq Y$ such that in the graph induced on the vertices of P, h, v have in-degree 0 (respectively 1) and outdegree 1 (respectively 0) and h is connected to v using exactly the edges in P.

$$\mathsf{PATH}(h, v, Y) \equiv (\exists P \subseteq Y)\, \mathsf{CONN}(h, v) \wedge \left(\forall P' \subsetneq P\right) \neg \mathsf{CONN}(h, v)$$

where

$$\begin{aligned} \mathsf{CONN}(h, v) \quad &\equiv \quad \left(\exists! z, z' \in V(P)[E(h, z) \wedge E(z', v)]\right) \wedge \left(\forall z, z' \in V(P)[\neg E(z, h) \wedge \neg E(v, z')]\right) \\ &\wedge \quad (\forall w \in V(P) \setminus \{h, v\} \mathsf{DEG}(w, P)) \end{aligned}$$

3. $\mathsf{NXT}^*(h, v) \vee (h = v)$ is the predicate that says that in a total ordering of the vertices of G (NXT^*) every cycle in the cycle cover induced by Y has a unique least numbered vertex namely h.

We define NXT^* and show that it is MSO-expressible in the subsequent Lemma 5.2. □

Let G be the input graph of bounded treewidth. We will augment G with some new vertices and edges to yield a graph G' again with a tree decomposition T' of bounded treewidth. Then we have:

Lemma 5.2. *There exists a relation* NXT *on vertices of* G' *which satisfies the following:*

1. NXT *is compatible*[4] *with the tree decomposition* T'
2. NXT *is a partial order on the vertices of* G'
3. NXT *is computable in* L
4. *The transitive closure* NXT^* *is a total order when restricted to the vertices of* G

[4]Binary relation R is said to be compatible with the tree decomposition T' of G if the Gaifman graph of R has T' as its tree decomposition.

5. NXT^* *is expressible as an* MSO*-formula over the vocabulary of* G' *along with* NXT.

The construction of such a relation is fairly straight forward and considered folklore in the Finite Model Theory literature (See for example Theorem VI.4 in [CF12]). Here we include a proof of Lemma 5.2 (obtained independently) for the sake of completeness.

Proof. We first define the graph G' in which we add three vertices b_l, b_0, b_r for every bag B in the tree decomposition of G. We will use the following running example, let A be a parent bag in the tree decomposition with left child B and right child C. The scheme we propose adds vertices a_l, a_0, a_r, b_l, b_0, b_r and c_l, c_0, c_r, with $a_l, a_0, a_r, b_r, c_l \in A$, $a_l, b_l, b_0, b_r \in B$ and $a_r, c_l, c_0, c_r \in C$.

We put an edge between every bag vertex $b \in V(G') \setminus V(G)$ and every vertex v sharing a bag with b. Clearly this is a valid tree decomposition of tree-width which is at most 6[5] greater than that of G.

We now traverse the tree T' using an Euler Traversal (see [CM87]). Every internal bag of the the tree is visited thrice in an Euler traversal - first when the traversal visits the bag the first time, second after exploring its left child, and finally third when it is done exploring its right child. For a bag B in the tree decomposition, if b_l, b_0, b_r are the vertices added, the vertices are visited in the following order in the Euler traversal of the tree: First b_l is visited, after which the left child of B is explored. Following this, the traversal returns to B via b_0, after which the right child of B is explored. Finally we visit B again via b_r and proceed to B's sibling via B's parent.

Define NXT_{Bag} to be the following ordering of the bag vertices: $a_l < b_l < b_0 < b_r < a_0 < c_l < c_0 < c_r < a_r$. In the case when b has children, they are explored between b_l and b_0 (left child) and b_0 and b_r (right child). Note that by definition, NXT_{Bag} is a relation of bounded treewidth.

Next we extend the relation NXT_{Bag} to a relation NXT_1 on pairs which include vertices of G. For every vertex v occurring in bag A if v does not belong to the left child B of A but belongs to the right child C then add the tuple $a_0 < v$ to NXT_1 else we just add $a_l < v$. Symmetrically, we add $v < a_0$ or $v < a_r$ depending on whether v belongs to the left child B but not to the right child C, or not. If it does not belong to either child, we add the tuples $b_r < v$ and $v < c_l$. Since all these changes are local, NXT_1 too is a relation of bounded treewidth.

Now consider the transitive closure NXT_1^* of NXT_1. This may well not be a total order on the vertices of G'. But we have the following:

[5]Alternately, following scheme also works: Add all the 6 bag vertices(b_l, b_0, b_r and c_l, c_0, c_r) of the children to the parent bag A. Symmetrically, add the bag vertices a_l, a_0, a_r to both B and C. Note that this is a local operation and it increases the treewidth of every bag by at most 9: 6 from the children and 3 from the parent. The final graph G' obtained this way has treewidth at most 12 more than G.

Claim 5.3. Bag vertices are totally ordered under NXT_1^*. If two vertices of G are incomparable under NXT_1^*, then there must exist a common bag to which both must belong.

Proof. That bag vertices are totally ordered follows from the definition of Euler traversal of a tree. To see the other part of the claim let $u, v \in V(G)$ be unordered by NXT_1^*. Let the least common ancestor (LCA) bag of all the bags to which u belong be Q and the LCA bag of all bags containing v be R and further P is the LCA bag of Q, R. From the assumption that u, v do not belong to the same bag not all three P, Q, R can be the same, in particular Q, R are distinct. If P is distinct from Q, R then from NXT_1^* we know that: $q_r < p_0 < r_l$ (assuming, wlog that Q is a left descendant and R a right descendant of P). Also, $u < q_r$ (or possibly even $u < q_0 < q_r$) and $r_l < v$ (or even $r_l < r_0 < v$) are present in NXT_1. Thus $u < v$ is present in NXT_1^*. This leaves the case when $P = Q$ (the other case, $P = R$, is symmetric). Let P' be the bag containing u which is nearest along the tree to R. Without (much) loss of generality suppose R is a descendant of the left child of P'. Let R' be the left child of P' (R' may be the same as R). From the fact that R is the highest bag containing v, we know that $v < r_r$. Because u does not occur in either child of P' we have that $r'_r < u$. Now r_r is either the same as r'_r (if $R = R'$) or $r_r < r'_r$ (if R is a proper descendant of R'). In either case we get $v < u$ in the transitive closure of NXT_1^*. Other cases are similar. □

Now we justify why NXT^* is MSO-expressible: notice that NXT^* is *not* MSO-expressible, given just the tree decomposition T of the graph G. In fact, what we have shown above is that there is a Logspace transducer that takes in the tree decomposition of T and produces a graph G' with tree decomposition T' and a relation NXT that is compatible with T and is of bounded treewidth such that NXT^* is expressible as an MSO-formula over the vocabulary of G' along with NXT.

It is important that the relation NXT we build should be of bounded treewidth[6] in order to invoke Courcelle's Theorem. To this end,

- Firstly notice that, we can find NXT_1^* in L by using Theorem 2.12 because NXT_1 is a relation of bounded treewidth and computing NXT_1^*, the trasitive closure of NXT_1 is precisely directed reachability which is MSO-expressible and hence computable in L on bounded treewidth graphs.
- From this we find pairs $u, v \in V(G)$ which are incomparable under NXT_1^*. Next we order any unordered u, v in L by their binary values and add these tuples (i.e. one of $u < v$ and $v < u$ for every unordered u, v) to NXT_1 to yield NXT. From Claim 5.3, we

[6]Recall that, we say a relation is of bounded treewidth if the Gaifman graph associated with the relation is a bounded treewidth graph. Courcelle's theorem and its variations allow us to use any relation in our MSO-formula provided the relation is of bounded treewidth.

know that if two vertices of G are incomparable under NXT_1^*, then there must exist a common bag to which both must belong. This ensures that the treewidth of NXT is the same as treewidth of NXT_1 and hence of bounded treewidth. This also implies that NXT is *compatible* with the tree decomposition T'.

- Putting the two points above together, we have that the vertices of G are totally ordered under NXT^* (and since NXT^* is just the transitive closure of NXT, it is expressible as an MSO-formula over the vocabulary of G' along with NXT via directed reachability).

□

Notice that we applied the Logspace version of Courcelle's theorem in order to build NXT, which is used in our MSO formula to recognize cycle covers.

Lemma 5.1 along with the Fact 1 yields:

Lemma 5.4. *For any matrix $A_{n\times n}$ of treewidth $k \geqslant 2$ and having integer entries, there is a Logspace algorithm that constructs an $(m \times m)$(where $m = \mathrm{poly}(n)$) matrix B with entries from $\{0, 1\}$, such that det$(A) =$ det(B) and the treewidth of B is the same as the treewidth of A.*

Proof. We replace each edge (u, v) in G_A by a series-parallel graph with edge weights from $\{0, 1\}$ such that the sum of weights of all paths between u and v exactly equals the weight of (u, v) and the construction doesn't alter the sign of the cycle cover in which (u, v) is present.

We use a construction similar to [Tod91, DKLM10] – If the weight of the edge (u, v) is an n-bit integer $w = w_n w_{n-1} \dots w_1$, we create a gadget of weight 2^i for each w_i and add edges (u, s_i) and (t_i, v). We add self loops on all the vertices in this gadget except u and v. We also ensure that all paths between u and v are of equal length (say l). If n' is the number of new vertices added, then a cycle cover of G_B will consist of the original cycles in G_A along with the $n' - l$ self loops that result from the series-parallel graph now representing (u, v). Since the sign of a monomial in the determinant is decided by the number of cycles (equivalently the number of heads in a cycle cover) corresponding to the permutation of the monomial, if we can ensure that $n' - l$ is always even, then the sign of the cycle cover will be $(-1)^k = (-1)^{n'-l+k} = (-1)^k$. This is easily taken care of by introducing v by a new vertex v', removing all the edges (t_i, v), introducing edges (t_i, v') and (v', v).

It remains to show that from the tree decomposition of G_A, we can obtain a tree decomposition of G_B in L. This is easily accomplished as follows: For an edge (u, v) in G_A, find the highest bag b in the tree where it occurs and add a new bag b_{uv} as a child to b which just contains u and v and the edge (u, v). Attach the tree decomposition of the series-parallel graph gadget corresponding to (u, v) in G_B as a child of b_{uv}. It is easy to verify that what results is still a tree decomposition – Every edge in B is present in some bag. If (u, v) is present in bags b_1 and b_2, it is present in all the bags in the unique path between b_1 and b_2 in the tree. Here one has to argue two cases: If (u, v) is an edge in the series-parallel gadget, then it is only present in the bags corresponding to the tree decomposition of the series parallel graph (children of b_{uv}). Else (u, v) must have been an edge in G_A. In this case, after appending the tree decomposition of the series-parallel graph to b_{uv}, it still stays a tree decomposition (Note that u, v must both be in the root of the tree decomposition of the (u, v) edge gadget). The size of any bag in this decomposition is $\max(k, 2)$ (since any series-parallel graph has treewidth at most 2). □

Thus, using the histogram version of Courcelle's theorem (Theorem 2.13) and Lemma 5.4, we get:

Theorem 5.5. *The determinant of a bounded treewidth matrix A with integer entries is computable in L.*

Proof. Firstly, obtain the matrix B from A using Lemma 5.4. The histogram version of Courcelle's theorem as described in Theorem 2.13 when applied to the formula $\phi(X, Y)$ above yields the number of cycle covers of G_B parametrized on $|X|, |Y|$. But in the notation of Fact 1 above, $|X| = k$ and $|Y| = n$, so we can easily compute the determinant as the alternating sum of these counts. □

5.2 Applications

We use Theorem 5.5 to compute the following linear algebraic quantities in the context of bounded treewidth matrices :

1. Characteristic Polynomial (Corollary 5.6).
2. Rank of a matrix (Corollary 5.8).
3. Feasibility of a system of linear equations (Corollary 5.9).
4. Inverse of a matrix (Corollary 5.10).
5. Powers of a matrix (Corollary 5.11).
6. Arborescences and directed Euler tours (Corollary 5.12).

5.2.1 Linear Algebra Applications

Corollary 5.6. *There is a Logspace algorithm that takes as input a* $(n \times n)$ *bounded treewidth matrix* A, 1^m, *where* $1 \leqslant m \leqslant n$ *and computes the coefficient of* x^m *in the characteristic polynomial* ($\chi_A(x) = \det(xI - A)$) *of* A.

The characteristic polynomial of an $(n \times n)$ matrix A is the determinant of the matrix $A(x) = xI - A$. We could use Theorem 5.5 to compute this quantity (since $A(x)$ is bounded treewidth, if A is bounded treewidth). However, Theorem 5.5 holds only for matrices with integer entries while the matrix $A(x)$ contains entries in the diagonal involving the indeterminate x.

We proceed as follows: In the directed graph corresponding to A, replace a self loop on a vertex of weight $x - d$ by a gadget of weight $-d$ in parallel with a self loop of weight x (In the event that there is no self loop on a vertex in A, add a self loop of weight x on the vertex). Replace the weights on the other edges according to the gadget in Lemma 5.4. We have added exactly n self loops, each of weight x (for the original vertices of A).

We first consider a generalization of the determinant of $\{0, 1\}$-matrices of bounded treewidth viz. the determinant of matrices where the entries are from a set whose size is a fixed universal constant and the underlying graph consisting of the non-zero entries of A is of bounded treewidth.

Lemma 5.7. *Let* A *be a matrix whose entries belong to a set* S *of fixed size independent of the input or its length. If the underlying digraph with adjacency matrix* A', *where* $A'_{ij} = 1$ *iff* $A_{ij} \neq 0$, *is of bounded treewidth then the determinant of* A *can be computed in* L.

Proof. Let $s = |S|$ be a universal constant, $S = \{c_1, \dots, c_s\}$ and let val_i be the predicates that partitions the edges of G according to their values i.e. $\mathrm{val}_i(e)$ is true iff the edge e has value $c_i \in S$. Our modified formula $\psi(X, Y_1, \dots, Y_s)$ will contain s unquantified new edge-set variables $Y_1, \dots, Y_s$ along with the old vertex variable X, and is given by:

$$\forall e \in E((e \in Y_i \rightarrow \mathrm{val}_i(e)) \wedge (e \in Y \leftrightarrow \vee_{i=1}^{s}(e \in Y_i) \wedge \phi(X, Y))$$

Notice that we verify that the edges in the set Y_i belong to the i^{th} partition and each edge in Y is in one of the Y_i's. The fact that the Y_i's form a partition of Y follows from the assumption that $\mathrm{val}_i(e)$ is true for exactly one $i \in [s]$ for any edge e.

To obtain the determinant we consider the histogram parameterized on the s variables $Y_1, \dots, Y_s$ and the heads X. For an entry indexed by $x, y_1, \dots, y_s$, we multiply the entry by $(-1)^{n+x} \prod_{i=1}^{s} {c_i}^{y_i}$ and take a sum over all entries. $\square$

In light of the Lemma above, we can compute the characteristic polynomial as follows:

Proof. (of Corollary 5.6) While counting the number of cycle covers with k cycles, we can keep track of the number of self-loops occurring in a cycle cover. By doing this, we can obtain the coefficient of x^r in the characteristic polynomial from the appropriate histogram entry via the procedure outlined in Lemma 5.7. Hence we can also compute the characteristic polynomial in L. $\square$

Corollary 5.8. *Given a bounded treewidth matrix $A_{m\times n}$ the rank of A can be computed in L.*

Proof. A can be interpreted as the biadjacency matrix of a bipartite graph. Now, consider the matrix $B = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}$ – this is a matrix of dimension $(m+n)\times(m+n)$. The matrix B corresponds to the adjacency matrix of A. Let row-rank(A) = column-rank$(A) = r$. Since A and A^T have the same rank, rank(A) + rank$(A^T) = 2r$ = rank(B). Now we use Mulmuley's method[Mul87]: Let $Z_{ii} = z^{i-1}$ be a diagonal matrix in the indeterminate z. Compute the characteristic polynomial $\det(xI - ZB)$ of ZB and use the fact that the rank of ZB is a number r such that x^{n-r} is the smallest power of x with a non-zero coefficient. We are now done with the help of Corollary 5.6. $\square$

$\mathsf{FSLE}(A, b)$ is the following problem: Given a system of m linear equations (with integer coefficients, w.l.o.g) in variables $z_1, \ldots, z_n$ and a target vector b, we want to check if there is a feasible solution to $Az = b$. That is, we want to decide if there is a setting of the variable vector $z \in \mathbb{Q}^n$ such that, $Az = b$ holds for a bounded treewidth matrix $A \in \mathbb{Z}^{m\times n}$ (when we say a rectangular matrix is bounded treewidth, we mean the underlying bipartite graph on $(m+n)$ vertices has bounded treewidth).

Corollary 5.9. *For a bounded treewidth matrix $A_{m\times n}$ and vector $b_{n\times 1}$, $\mathsf{FSLE}(A, b)$ is in L.*

Proof. We know that the system of linear equations given by A, b is feasible if and only if rank(A) = rank$([A : b])$. Therefore, we can use the Logspace procedure for matrix rank given by Corollary 5.8 to decide FSLE. $\square$

Corollary 5.10. *There is a Logspace algorithm that takes as input a $(n \times n)$ bounded treewidth matrix A, $1^i, 1^j, 1^k$ and computes the k-th bit of A^{-1}_{ij}.*

Proof. The inverse of a matrix A is the matrix $B = \frac{\mathbf{C}^T}{\det(A)}$ where $\mathbf{C} = (C_{ij})_{1\leqslant i,j\leqslant n}$ is the cofactor matrix, whose (i, j)-th entry $C_{ij} = (-1)^{i+j}\det(A_{ij})$ is the determinant of the $(n-1)\times(n-1)$ matrix obtained from A by deleting the i-th row and j-th column. If we can compute C_{ij} in L, we can compute the entries of B via integer division which is known to be in L

from [HAB02]. To this end, consider the directed graph G_A represented by A. To compute $\det(A_{ij})$, swap the columns of A such that the j-th column becomes the i-th column. The graph so obtained is of bounded treewidth (To see this, notice that the swapping operation just re-routes all incoming edges of j to i and those of i to j. The tree decomposition of this graph is just obtained by adding vertices (i, j) to every bag in the tree decomposition of G_A and also the edges rerouted to the respective bags. This increases the treewidth by 2). Now, remove the i-th vertex in G_A and all edges incident to it to get a graph $G_{A'_{ij}}$ on $(n-1)$ vertices. The swapping operation changes the determinant of A_{ij} by a sign that is $(-1)^{i-j} = (-1)^{i+j}$. Computing the determinant of this modified matrix A'_{ij} yields C_{ij} as required. Since A'_{ij} is obtained from A by removing a vertex and all the edges incident on it, the treewidth of A'_{ij} is at most the treewidth of A. By Theorem 5.5, C_{ij} is in FL. □

Corollary 5.11. *There is a Logspace algorithm that on input an* $(n \times n)$ *bounded treewidth matrix* A, $1^m, 1^i, 1^j, 1^k$ *gives the* k*-th bit of* (i, j)*-th entry of* A^m.

Proof. Consider $A' = (I - tA)^{-1}$ where I is the $(n \times n)$ identity matrix and t is a small constant to be chosen later. Notice that $A' = (I - tA)^{-1} = \sum_{j \geqslant 0} t^j A^j$. By choosing t as a suitably small power of 2 (say $2^{-p} = t$ such that $2^p > \|A\|$) and computing A' to a suitable accuracy, we can read the (i, j)-th entry of A^m off the appropriate bit positions of the (i, j)-th entry of A'. So, in essence the problem of powering bounded treewidth matrix A reduces to the problem of computing the inverse of a related matrix which is known to be in L via Corollary 5.10. □

5.2.2 Spanning Trees and Directed Euler Tours

Fact 7. The number of arborescences[7] of a digraph equals any cofactor of its Laplacian.

where the Laplacian of a directed graph G is $D - A$ where D is the diagonal matrix with the D_{ii} being the out-degree of vertex i and A is the adjacency matrix of the underlying undirected graph. The BEST Theorem states:

Fact 8 ([AEB87][TS41]). The number of Euler Tours in a directed Eulerian graph K is exactly: $t(K) \prod_{v \in V} (\deg(v) - 1)!$ where $t(K)$ is the number of arborescences in K rooted at an arbitrary vertex of K and $\deg(v)$ is the indegree as well as the outdegree of the vertex v.

We combine Facts 7 and 8with Theorem 5.5 to compute the number of directed Euler Tours in a directed Eulerian graph in L.

Use the Kirchoff Matrix Tree theorem[Sta13] and Fact 8:

[7]Alternately, arborescences are MSO_2-definable and, thus, counting them in bounded treewidth graphs can be implemented in L via Theorem 2.13.

Corollary 5.12. *Counting arborescences and directed Euler Tours in a directed Eulerian graph* G *(where the underlying undirected graph is bounded treewidth) is in* L.

5.3 Hardness Results

We now show some hardness results to complement our Logspace upper bounds.

5.3.1 Hardness of Bounded Treewidth Determinant

Proposition 5.13 (Hardness of Bounded Treewidth Determinant). *For all constant* $k \geqslant 2$, *computing the determinant of an* $(n \times n)$ *matrix* A *whose underlying undirected graph has treewidth at most* k *is* L*-hard.*

Proof. We reduce the problem ORD of deciding for a directed path P and two vertices $s, t \in V(P)$ if there is a path from s to t (known to be L-complete via [Ete97]) to computing the determinant of bounded treewidth matrices (Note that P is a path and hence it has treewidth 1). Our reduction is as follows: Given a directed path P with source a, sink b and distinguished vertices s and t, we construct a new graph P' as follows: Add edges $(s', a), (s', t), (t, s), (s, a)$ and (b, t') and remove edges $(s', s), (t, t')$ where s' and t' are vertices in P such that $(s', s), (t, t') \in E(P)$ (See Figure 5.1).

We claim that there is a directed path between s and t if and only if the determinant of the adjacency matrix of P' is zero. If there is a directed path from s to t in P, then there are two cycle covers in P' : $(a, s')(s, t)(t', b)$, with three cycles and $(a, s', t, s), (t', b)$, with two cycles. Using Fact 1, the signed sum of these cycle covers is $(-1)^{n+3} + (-1)^{n+2} = 0$, which is the determinant of P'.

In the case that P has a directed path from t to s (see Figure 5.2), then there is one cycle namely (a, t, s, b, t', s'). We argue as follows: The edges $(t, s), (s, b), (b, t'), (t', s')$ are in the cycle cover since they are the only incoming edges to s, b, t', s' respectively. So (t, s, b, t', s') is a part of any cycle cover of the graph. This forces one to pick the edge (s', a) and hence we have one cycle in the cycle cover for P'. □

Remark 5.14. There has been some work on understanding some graph polynomials like the Permanent, Determinant, Perfect Matching, Hamiltonian cycle, etc in the context of Valiant's algebraic complexity classes. It is known that the Permanent, Perfect Matching and Hamiltonian cycle polynomials are complete for VNP, the algebraic analogue of NP whereas the Determinant is complete for VBP the algebraic analogue of NL. Recently, Flarup,

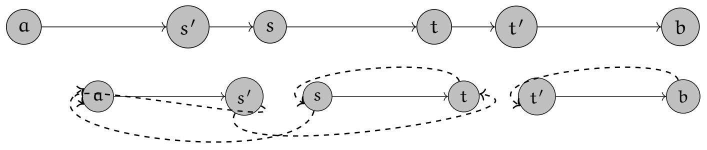

Figure 5.1: s occurs before t

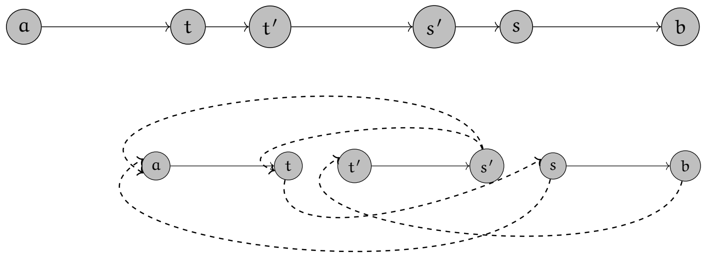

Figure 5.2: t occurs before s

Koiran and Lyaudet [FKL07] showed that the Permanent and Hamiltonian cycle polynomials of bounded treewidth graphs are complete for VF, the class of polynomials computed by arithmetic formulas of polynomial size. Naturally, this gives an upper bound of GapNC^1 for computing the value of Determinant/Permanent of bounded treewidth matrices (just initialize the variables in the arithmetic formula with appropriate weights). So did we accidentally prove $\mathsf{GapNC}^1 = \mathsf{L}$? A few comments are in order: Firstly, the work of Flarup et al. [FKL07] assumes that that the graph is provided via a balanced tree decomposition of depth $O(\log n)$ and it computes the Permanent polynomial (a similar technique is employed to compute the Hamiltonian cycle polynomial as well) by computing partial cycle covers along the tree decomposition. This completely disregards the complexity of obtaining such a tree decomposition; In fact, Elberfeld et al. show that the language $\mathsf{TREEWIDTH} - k$ consisting of all those graphs of treewidth at most k is hard for Logspace (Lemma 1.4 in [EJT10]).

On the other hand, the subsequent work of Elberfel, Jakoby and Tantau [EJT12] shows that MSO_2-definable properties on bounded treewidth graphs can be decided, in fact, in NC^1 and the histogram computable in[8] $\#\mathsf{NC}^1$. This yields a GapNC^1 upper bound for computing both the Permanent and the Determinant of bounded treewidth matrices since both these quantities can be computed as histograms of a MSO_2-definable quantity. In some sense, the VF upper bounds of computing the Permanent of Flarup et al. can be interpreted as working over the term representation since their formula (algorithm) is built over a balanced tree decomposition (provided for free) of the input graph.

[8]There is yet another caveat with this statement, namely that the representation of the graphs is now in term representation using parentheses

Notice that our L-hardness of bounded treewidth Determinant works equally well for the bounded treewidth Permanent. Also, the tree decomposition of P' can be inferred from that of P (since only constantly many edges are added, locally) via projections. From the preceeding discussion, it seems to us that the hardness of bounded treewidth determinant presented above is in some sense the hardness of obtaining a tree decomposition of a graph given its adjacency matrix.

5.3.2 Hardness of Bounded Treewidth Matrix Powering

Proposition 5.15 (Hardness of Bounded Treewidth Matrix Powering)**.** *Bounded Treewidth Powering is* L*-hard under disjunctive truth table reductions.*

Proof. We reduce ORD to matrix powering. Given an directed path P on n vertices and distinguished vertices s and t, we argue as follows: There is a directed path between s and t, then it must be of length i for an unique $i \in [n]$. Consider the matrix $(I + A_P)^n$: s and t are connected by a path if and only if $(I + A_P)^n_{s,t} \neq 0$. This is because $(I + A_P)^n_{s,t}$ gives the walks from s to t, and if at all there is a path from s to t, then there is definitely a walk of length at most n between them. Checking if this entry is zero can be done by a DNF which takes as input the bits of $(I + A_P)^n_{s,t}$. □

5.3.3 Hardness of Bounded Treewidth IMM

Proposition 5.16 (Hardness of Bounded Treewidth IMM)**.** *Given a sequence of bounded treewidth matrices with rational entries $M_1, M_2, \ldots, M_n$ and $1^i, 1^j, 1^k$ as input, computing the k-th bit of (i,j)-th entry of $\prod_{l=1}^{n} M_l$ is #L-hard.*

Proof. (of Proposition 5.16) We reduce $\{0,1\}$-matrix powering (over $\mathbb{N}$) to iterated matrix multiplication (over $\mathbb{N}$) of bounded treewidth matrices with entries from $\{0,1\}$. Given an $(n \times n)$ matrix A with $\{0,1\}$-entries, and $1^m, 1^i, 1^j$ the matrix powering problem is to find the (i,j)-th entry of A^m. From the underlying digraph $G_A = (V = \{v_1, v_2, \ldots, v_n\}, E)$, we construct a sequence of bounded treewidth matrices as follows: We construct two gadgets (see Figures 5.3, 5.4) $\mathcal{V}_l$ and $\mathcal{V}_u$ – Both are graphs on $2n^2$ vertices divided in to $2n$ partitions where each partition is a copy of V (such that there are no edges between vertices in the partition): $U = \sqcup_{i=1}^{n} U_i$ and $L = \sqcup_{i=1}^{n} L_i$ where each $L_i = U_i = V$. We also have the edges between:

1. $v_i \in U_j$ and $v_i \in U_{j+1}$

2. $v_i \in L_j$ and $v_i \in L_{j+1}$

for all $i \in [n], j \in [n-1]$.The edges are basically an identity perfect matching between U_i and U_{i+1} and also L_i and L_{i+1}. Now we add edges in $\mathcal{V}_l$ and $\mathcal{V}_u$ according to edges present in G_A: If $(v_i, v_1), \ldots, (u, v_r)$ are edges in G_A out of vertex v_i, then for all $i \in [n]$,

1. In $\mathcal{V}_l$, we add an edge between $v_i \in L_i$ to $v_1, \ldots, v_r \in U_{i+1}$.
2. In $\mathcal{V}_u$, we add an edge between $v_i \in U_i$ to $v_{i_1}, \ldots, v_{i_r} \in L_{i+1}$

We now construct a walk gadget $\mathcal{W}$ using alternating copies of the vertex gadgets $\mathcal{V}_l$ and $\mathcal{V}_u$. To raise A to the m-th power, construct: $\mathcal{V}_1, \ldots, \mathcal{V}_m$ where $\mathcal{V}_1 = \mathcal{V}_l$, $\mathcal{V}_2 = \mathcal{V}_u$ and so on. We will now follow the convention where we refer to a L_j or U_j that is a part of $\mathcal{V}_i$ as $L_{i,j}$ and $U_{i,j}$ respectively. Connect $\mathcal{V}_i$ and $\mathcal{V}_{i+1}$ by the following edges:

1. $v_i \in U_{i,n}$ and $v_i \in U_{i+1,1} \forall i \in [m-1]$.
2. $v_i \in L_{i,n}$ and $v_i \in L_{i+1,1} \forall i \in [m-1]$.

Additionally if (v_n, v_{i_1}), $\ldots, (v_n, v_{i_k})$ are edges out of v_n, then add those corresponding edges between $L_{i,n} \in \mathcal{V}_i$ and $U_{i+1,1} \in \mathcal{V}_{i+1}$ if $\mathcal{V}_i$ is a $\mathcal{V}_l$ gadget. Otherwise $\mathcal{V}_i$ is a $\mathcal{V}_u$ gadget and hence add the corresponding edges between $U_{i,n} \in \mathcal{V}_i$ and $L_{i+1,1} \in \mathcal{V}_{i+1}$. It is easy to see that there is a bijection between walks of length m in G_A and paths of length m in $\mathcal{W}$. The gadget $\mathcal{W}$ that results is of constant treewidth. $\square$

5.4 Discussion

We studied linear algebraic invariants of a matrix by interpreting the matrix as an adjacency matrix of a bounded treewidth graph and showed that several invariants like the determinant, rank, characteristic polynomial, etc. are easier than in matrices that represent general dense graphs, where the same invariants are GapL-hard are to compute.

A few questions that remain open here are:

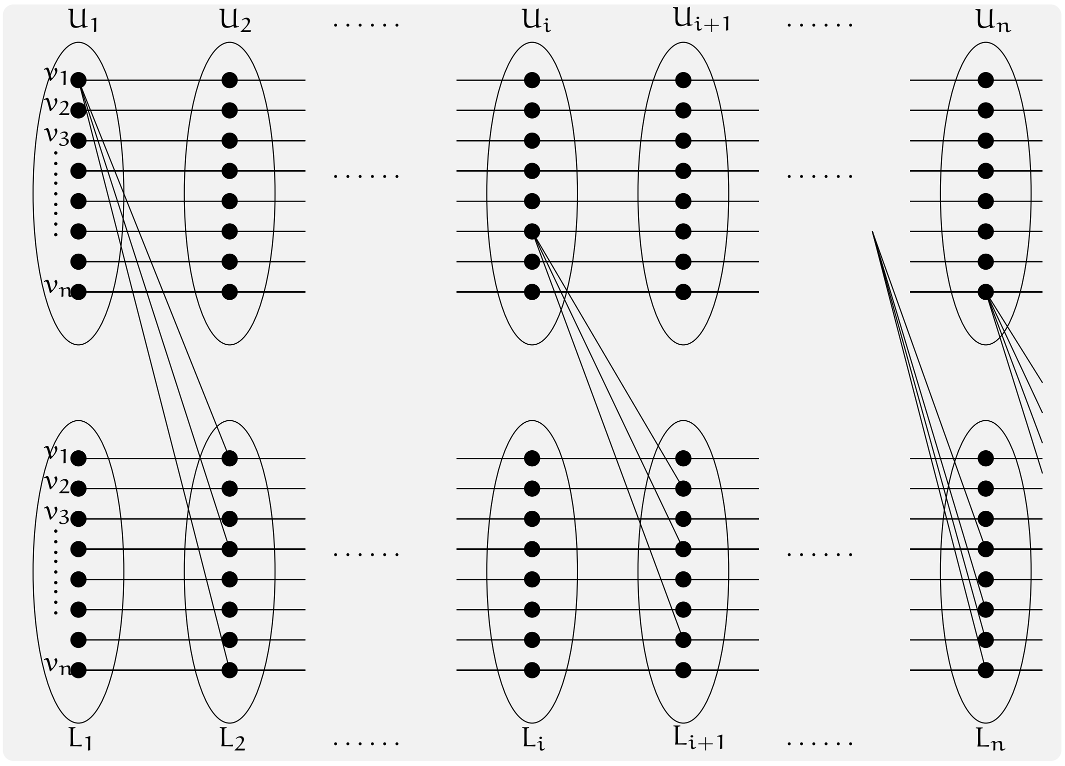

Figure 5.3: The gadget $\mathcal{V}_u$

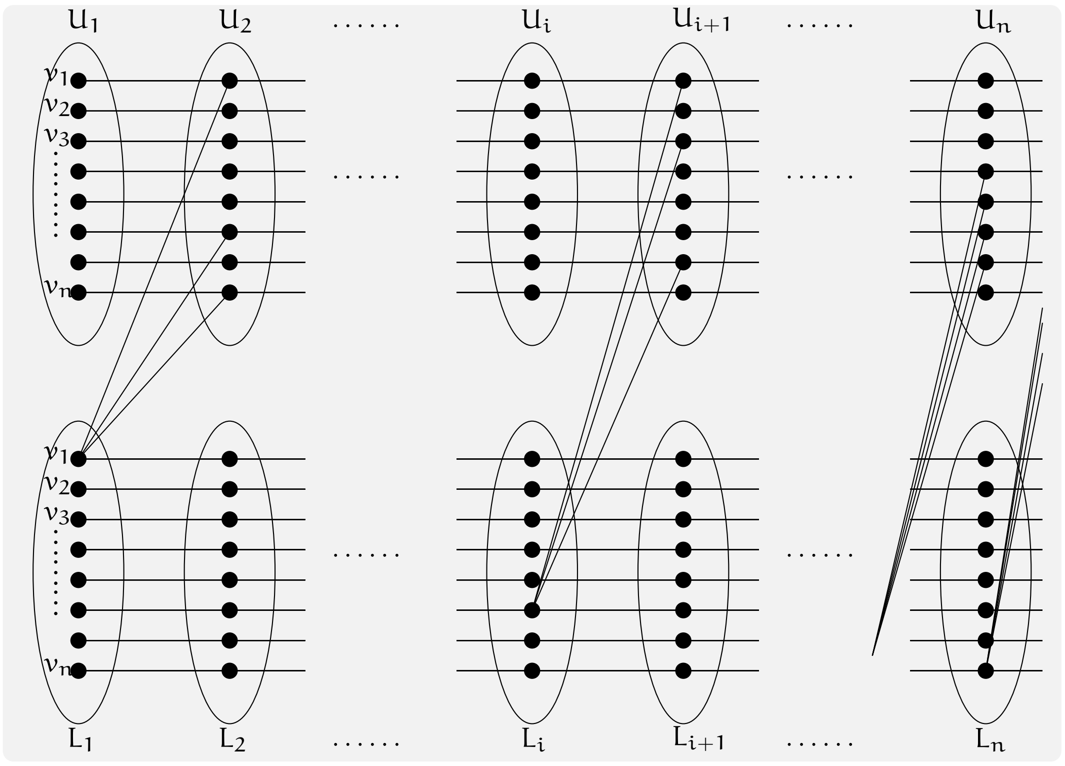

Figure 5.4: The gadget $\mathcal{V}_l$

- Can the determinant of bounded clique-width adjacency matrices be computed in better than GapSAC^1? (it is known to be L-hard for bounded tree-width graphs from Lemma 5.13).
- For what other classes of graphs do these linear algebraic invariants become easy? What is the exact complexity of computing these invariants for bounded clique width graphs? Chordal graphs?

Chapter 6

Counting Euler Tours on Bounded Treewidth Graphs

In this chapter, we are concerned with the problem of counting Euler tours on graphs of bounded treewidth, which turns out to be #P-complete for general graphs. Many problems which are NP-hard for general graphs, can be solved in Polynomial time on bounded treewidth graphs. So it is not overtly optimistic to hope that there is a polynomial time algorithm to count Euler tours on bouned treewidth graphs. One particularly attractive option (especially in light of results in Chapter 5) is to use the powerful meta-theorem machinery of Elberfeld et al. [EJT10] to count the number of Euler Tours. However, Eulerianity is provably not MSO-expressible [EF95] and hence approaches like the one in Theorem 5.5 are not directly applicable in our context.

Our strategy to count Euler tours is as follows: Given a bounded treewidth graph G by its tree decomposition, we count the number of Euler tours of G by counting the number of Hamiltonian tours of the line graph of G, $L(G)$. In general, there is no bijection between these two quantities, but we show that G can be modified to obtain G' ($tw(G') \leqslant tw(G) + 3$) such that G, G' have the same number of Eulerian tours, which equals the number of Hamiltonian tours of $L(G')$. Henceforth, we will be primarily interested in line graphs of bounded treewidth graphs. It is known that such graphs are of bounded clique width [OS06].

We base our proof on a proof that the decision version of Hamiltonicity is polynomial time computable in bounded clique width graphs [EGW01]. We prove that this algorithm can be parallelized and extended to the counting version. Next, we show that for line graphs of bounded tree-width graphs, the clique width decomposition can be inferred from the corresponding tree-width decomposition. The bounded tree-width decomposition is obtainable by the Logspace version of Bodlaender's theorem [EJT10].

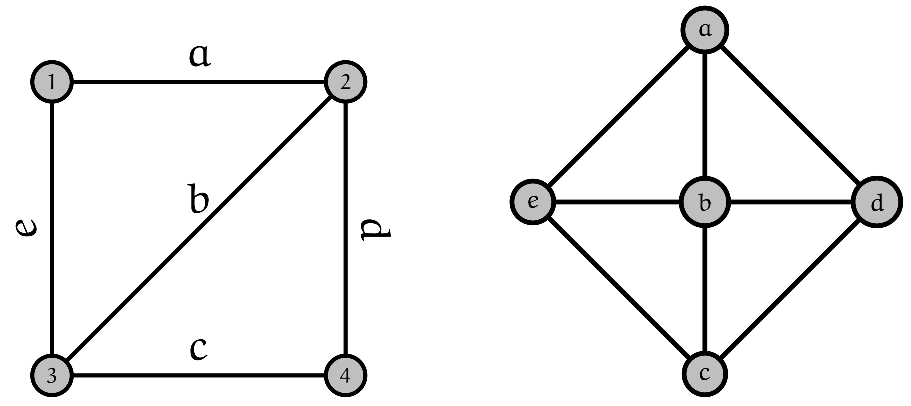

FIGURE 6.1: Graph G with no Euler tour, but $L(G)$ with Hamiltonian cycle

6.1 From Euler Tours to Hamiltonian cycles

It is possible to construct a graph G such that G has no Eulerian tours, but $L(G)$ has a Hamiltonian cycle[1]. Proposition 6.1 gives necessary and sufficient conditions for when a line graph of a given graph is Hamiltonian.

Proposition 6.1 ([HNW65]). *$L(G)$ is Hamiltonian if and only if G has a closed trail that contains at least one end point of every edge.*

Given a graph G, we want to construct a graph G' such that every closed trail in G' that contains at least one end point of every edge is exactly an Eulerian tour of G'. The following Lemma guarantees exactly this:

Lemma 6.2. *Given an undirected graph G, construct a graph $G' = (V', E')$ from G as follows: Replace every edge $e = (u, v)$ of G by path of length three. Then G and G' have the same number of Eulerian tours and the Eulerian tours of G' are in bijection with the Hamiltonian tours of $L(G')$.*

Proof. Recall that $G' = (V', E')$ is obtained from G as follows: Replace every edge $e = (u, v)$ of G by path of length three, namely $(u, x_e), (x_e, y_e), (y_e, v)$. For a graph G, let $\mathcal{E}_G$ and $\mathcal{H}_G$ denote the set of Euler Tours and Hamiltonian tours of G respectively. We claim the following: Consider the map $h : E(G')^m \to V(L(G'))^m$ (where $m = |E(G')|$), defined by $h : ET \mapsto HT$. Here $ET = \langle e_1, e_2, \ldots, e_m \rangle$ is an edge sequence of G' with e_1 being the least edge under an arbitrary but fixed ordering of the edges of G'; $HT = \langle v_{e_1}, v_{e_2}, \ldots, v_{e_m} \rangle$ is the corresponding vertex sequence of $L(G')$ (where we associate the edge $e \in E(G)$ with vertex $v_e \in V(L(G)))$. Then the proof is completed by invoking Lemma 6.3 to show that h is the desired bijection with its domain restricted to the set of Euler tours. $\square$

[1]Indeed, there is a 2-connected graph – K_4 with one of the edges removed – which is non-Eulerian but its line graph is Hamiltonian. See Figure 6.1.

Lemma 6.3. *We have the following properties of the map* h*:*

1. *If* $ET \neq ET'$ *then* $h(ET) \neq h(ET')$
2. $h(ET)$ *is defined for every Euler tour* ET
3. $HT = h(ET)$ *is a Hamiltonian cycle for an Euler tour* ET
4. *If* HT *is a Hamiltonian cycle in* $L(G')$ *then there exists an Euler tour* ET *in* G' *such that* $h(ET) = HT$

Proof.

1. Obvious from definitions of $h, L(G')$.
2. Obvious from definition of h.
3. If e, e' are consecutive edges in ET, then they must share a vertex since ET is an Euler tour and hence $v_e, v_{e'}$ must be adjacent in $L(G')$. Also since ET is a permutation of all the edges of G', therefore $h(ET)$ is a permutation of all the vertices of $L(G')$.
4. From the way G' is obtained from G, if $v_{e_{i-1}}, v_{e_i}, v_{e_{i+1}}$ are successive vertices on an arbitrary Hamiltonian Tour HT of $L(G')$, then e_{i-1}, e_i, e_{i+1} cannot all be incident on a vertex $u \in V(G) \cap V(G')$. For, suppose not, then there exist distinct vertices $a, b, c \in V(G) \cap V(G')$, such that e_{i-1}, e_i, e_{i+1} are subdivision edges of the edges $e' = (u, a), e'' = (u, b), e''' = (u, c)$. But then the edge $(x_{e''}, y_{e''})$ – the middle subdivision edge of e'' – cannot be traversed in ET. This is because $e_i = (u, x_{e''})$, one of its only two neighbours is not used to traverse it.

   This implies that if $HT = v_{e_1}, v_{e_2} \ldots$, then $e_1, e_2, \ldots$ is an Euler Tour of G'. Now, a Hamiltonian path in $L(G')$ is a permutation of the vertices v_e's of $L(G')$, thus induces a permutation of the edges of G'. But a sequence $e_1, \ldots, e_m$ is an Euler tour iff for every i the vertex incident on edges e_{i-1}, e_i and the vertex incident on edges e_i, e_{i+1} are distinct (and form the two endpoints of e_i). This is true because of the previous paragraph, completing the proof.

□

Notice that G is a minor of G', and the tree decomposition of G' can be obtained from that of G by locally adding to each bag containing an edge e of G, the extra vertices and edges of the path of length three. Hence, the following is immediate:

Proposition 6.4. G *is bounded treewidth iff* G' *is bounded treewidth.*

Proposition 6.5 ([GW07]). *If G is of treewidth k, then $L(G)$ has cliquewidth $f(k) = 2k + 2$.*

Proposition 6.6 ([EJT10]). *Given a bounded treewidth graph G, a balanced tree decomposition*[2] *of G is obtainable in L.*

We first need the Perfect Elimination Ordering(PEO) of the vertices of the graph (See Definition 2.19). It is known that a graph has a PEO if and only if it is chordal. Since we can do a chordal completion (See Definition 2.18) of a bounded treewidth graph (while preserving treewidth), such an ordering of the vertices always exists. Recently Arvind et al. gave a Logspace procedure for obtaining a PEO in k-trees (which are maximal treewidth-k graphs). We adapt this for graphs that are chordal completions of bounded treewidth graphs:

Lemma 6.7 (Adapted from [ADKK12]). *Given a balanced tree decomposition of a bounded treewidth graph, H, we can construct another graph G (of bounded treewidth) such that a Perfect Elimination Ordering and the corresponding elimination tree of G that is a balanced binary tree of depth $O(\log n)$, can be computed in L.*

Proof. We first do a chordal completion of H (to obtain G) by adding edges to every bag in the tree decomposition such that each bag contains a simplicial vertex (it could contain more than one, but at most k since the treewidth is at most k). Now, we can find a partition of the vertex set of $G - V(G) = R_0 \dot\cup R_1 \dot\cup \ldots \dot\cup R_l$ as follows: First, pick one simplicial vertex from each bag in the tree decomposition and make the layer R_0. If these are more than one simplicial vertex in a bag, these are picked in to the sub-layers of R_0, of which there could be at most k many of them which we call R_{0j} for $j \in [k]$ (once a vertex is picked this way, it is removed from the bag). Since the graph is chordal, this process results in a chordal graph again, and we now do the same process iteratively, and call the sets of simplicial vertices so obtained, $R_1, R_2, \ldots, R_l$, each of which have appropriate sublayers whenever there are more than one simplicial vertex in the bag (and we will exhaust all the vertices in the process). Note that this process can go on for at most $l = O(\log n)$ steps, which is the diameter of the graph (This is because we started with a balanced binary tree decomposition of height $O(\log n)$ and since every bag is a clique after the chordal completion, the distance between any two nodes in this tree decomposition is $O(\log n)$.

Now we claim that if we order $R_{01}, \ldots, R_{lk}$ in the reverse order and within each of these R_i, we order the vertices arbitrarily, we obtain a PEO of the graph. This follows straight away from the definition of a PEO and the construction of the R_i's.

[2]A tree decomposition of a graph is said to be balanced if the tree underlying the decomposition is balanced

Recall that an elimination tree for a graph $G = (V, E)$ and PEO $o = (v_1, v_2, \ldots, v_n)$ is $T(G, o) = (V_T, E_T)$ and is defined as:

$$\begin{aligned} V_T &= V \\ E_T &= \{(v_i, v_j) \in E | i < j \text{ and } \forall j', i < j' < j, (v_i, v_{j'}) \notin E\} \end{aligned}$$

$T(G, o)$ is a tree because every vertex v_i, $i < n$ is adjacent to exactly one vertex v_j with $j > i$. We can now construct the elimination tree from our PEO obtained from the R_i's. Note that every vertex in the elimination tree has at most k children which happens when a bag has k vertices all of which are simplicial, (they are present one each in each of the sub-layers R_{ij}, $j \in [k]$ of R_i). Hence we can construct an elimination tree of diameter at most $O(\log n)$ in Logspace. □

Lemma 6.8 (Adapted from [GW07]). *Given the tree decomposition of a graph G along with a elimination tree, the clique-width expression X of $L(G)$ is obtainable in L. The parse tree of this clique width expression has height at most $O(\log n)$*

We prove Lemma 6.8 via Lemma 6.9 and Proposition 6.10 below. We provide proofs for Lemma 6.9 and Proposition 6.10 for the sake of completeness(these proofs are exactly the same as the ones in [GW07], we just state it here for the sake of completeness and to highlight their Logspace implementability, which is implicit in the proofs in [GW07]). A crucial point here is to observe that since we have an elimination tree of $O(\log n)$ depth, the subsequent proofs also yield a clique width expression whose tree is $O(\log n)$ depth, which we exploit in our construction of the logspace-uniform $\#SAC^1$ circuits in the proof of Theorem 1.2.

Lemma 6.9 (Adapted from [GW07]). *The NLC-width of the line graph $L(G)$ of a bounded treewidth graph G is at most $k + 2$ and such a NLC-width expression is obtainable in L.*

Proof. Given an undirected graph $G = (V, E)$, let $o = (v_1, v_2, \ldots, v_n)$ be the PEO of the vertices of G. The structure of G can then be characterized by the PEO tree $T(G, o)$. Let $\mathrm{col} : V_G \to [k+1]$ be a $(k+1)$-coloring of G with $\mathrm{col}(v_i) \neq \mathrm{col}(v_j)$ for all $(v_i, v_j) \notin E$. Let $N^-(T(G, o), o, i) = \{v_{j_1}, v_{j_2}, \ldots, v_{j_m}\}$ (defined by the tree $T(G, o)$) and $N^+(G, o, i) = \{v_{l_1}, \ldots, v_{l_r}\}$ (defined by the graph G). For $i = 1, \ldots, n$, an NLC-width $(k+2)$ expression is recursively defined as follows:

1. If $m = 1$, then $Y_i = X_{j_1}$. If $m > 1$, then let

$$Y_i = X_{j_1} \times_I \ldots \times_I X_{j_m}$$

where $I = \{(s,s) | s \in [k+1]\}$. The graph $val(Y_i)$ is the disjoint union of graphs $val(X_{j_1}), \ldots, val(X_{j_m})$ where vertices with the same label in different graphs are connected by an edge. Note that the relation I uses only the labels $1, \ldots, (k+1)$. The label $(k+2)$ is exclusively for vertices that will not be connected with other vertices in any further composition step.

2. If $r > 0$, then let Z_i denote a NLC $(k+1)$-width expression that defines a complete graph with r vertices labeled by $col(v_{l_1}), \ldots, col(v_{l_r})$. Here $r \leqslant k$ labels are distinct and do not include the color $col(v_i)$ of v_i.

3. Now we define

$$X_i = \begin{cases} \circ_R(Y_i \times_S Z_i) & \text{if } m > 0 \text{ and } r > 0 \\ Z_i & \text{if } m = 0 \text{ and } r > 0 \\ \circ_R(Y_i) & \text{if } m > 0 \text{ and } r = 0 \end{cases}$$

where,

$$S = \{(s,s) | s \in [k+1] - col\{(v_i)\}\} \cup \{(col(v_i), s) | s \in [k+1]\}$$

and

$$R(s) = \begin{cases} s & \text{if } s \neq col(v_i) \\ (k+2) & \text{if } s = col(v_i) \end{cases}$$

We refer the reader to [GW07] for a proof of correctness of the observation the NLC-width $(k+2)$ expression X_n defines the line graph of G. To see that the NLC width expression X_n is obtainable in L, we argue as follows:

1. We obtain the tree decomposition of the graph G in L via Proposition 6.6.

2. Using Lemma 6.7, we can obtain the PEO of G and also construct the Elimination tree $T(G,o)$ in L.

3. From $T(G,o)$ and G, we can obtain m and r and subsequently, each element of $N^-(T(G,o),o,i)$ and $N^+(G,o,i)$ in L

4. We can compute the $(k+1)$-coloring of G, $col : V_G \to [k+1]$ in L via [EJT10] (Proof of Lemma 4.1).

We build the NLC width expression for the line graph of G from the elimination tree $T(G,o)$. The NLC width expression X_i is defined for each vertex of $T(G,o)$. This however depends only on $N^-(T(G,o),o,i)$ and $N^+(G,o,i)$ which can be obtained in L. Along with the fact that tree traversal via DFS is in L [CM87], we can obtain the NLC width expression for the line graph of G in L: We can represent the PEO tree using an expression involving '(' and

')'. Note that such an expression can be output by a Logspace transducer. This gives the structure of our NLC width expression, and now we can fill in this expression using the NLC width operations. This only involves local computations: for example at a node v_i of the tree, we compute in Logspace $N^-(T(G, o), o, i), m$ and $N^+(G, o, i), r$ and get the appropriate expressions based on the values of m, r as given in item 3 of the NLC width expression above. Since we build the NLC width expression over the balanced elimination tree of constant arity and depth $O(\log n)$ via Lemma 6.7 and every node in the elimination tree had at most k children, the parse tree of the NLC width expression is also of height $O(\log n)$. □

Gurski and Wanke [GW07] observe that it is sufficient to look at G that are k-trees here because the line graph of every subgraph of G then is an induced subgraph of the line graph of G and the class NLC_k is closed under taking induced subgraphs for every $k \geqslant 1$ (See Theorem 4 in [GW07]). Our method involves dealing with bounded treewidth graphs that are chordal, which are a strict superclass of k-trees and we observe that the property mentioned above still holds in this case.

Proposition 6.10. *Given a graph G of NLC-width at most k by an NLC-width expression Y, we can obtain the clique-width expression X of G, where $|X| \leqslant 2k+2$ in L.*

Proof. For the NLC width-$(k+2)$ expression X_i defined above, there is an equivalent clique width $(2k+2)$ expression X_i'. We prove by induction on i: For $i = 1$, there is an clique width-$(k+1)$ expression X_1' because $val(X_1)$ is just a graph on at most k vertices with labels from the set $[k+1]$. For $i > 1$, an equivalent clique width expression Y_i' for $Y_i = X_{j_1} \times_I \ldots \times_I X_{j_m}$ is obtained from the clique width expressions for $X_{j_1}', \ldots X_{j_m}'$ and k auxillary labels. This is because for $t = 1, \ldots, m$, the vertices of every $val(X_{j_t})$ are labeled by $k+1$ labels from $[k+2]$. Label $col(u_{j_t}) \in [k+1]$ is not used by the vertices of $val(X_{j_t}')$ and lable $k+2$ is not involved in any edge creation. The clique width expression X_i' for $X_i = \circ_R(Y_i \times_S Z_i)$ can finally be defined by clique width expression for Y_i and k auxillary labels because $val(Z_i)$ has at most k vertices. Since all these changes are local, we can the convert the NLC width $k+2$ expression to a clique width $2k+2$ expression by replacing the corresponding subexpressions for NLC width by the ones for clique width, to obtain the line graph of a bounded treewidth graph of treewidth at most k in Logspace. □

To sum up, these are the main preprocessing steps:

1. Obtain a balanced binary tree decomposition of the input treewidth k graph G in Logspace via Proposition 6.6 [EJT10].

2. Perform a chordal completion of G by adding edges to every bag.

3. Obtain a PEO tree of height $O(\log n)$, where every vertex has at most k children via Lemma 6.7.

4. Construct a NLC width $(k+2)$ expression via Lemma 6.9

5. From the NLC width $(k+2)$ expression, construct a clique width $(2k+2)$ expression via Proposition 6.10.

6.2 The $\#\mathsf{SAC}^1$ upper bound

Let X be the clique-width k expression for a labeled graph $G = (V, E, lab)$ such that G is val(X) and let $|V| = n$. Let G be of clique-width k. Hence by Definition 2.16, G can be constructed from the graph with n isolated unlabeled vertices, using at most k labels. Notice that X can be viewed as a tree (we will refer to this as the parse tree of the clique width expression) with the n isolated unlabeled vertices at the leaves and every internal node is labeled with one of the operations $o = \{\bullet_i, \oplus, \eta_{i,j}, \rho_{i\to j} : i, j \in [k] \wedge i \neq j\}$ To each internal vertex of the tree, we can associate a graph (possibly disconnected) which is a subgraph of G, and at the root of the tree, we get G itself. The size of the tree is polynomial in n and k. Our objective in this section will be to count the number of Hamiltonian cycles in G, when provided with the clique width expression X. We will count along the parse tree of the clique width expression.

To this end, we call a subset of edges $E' \subseteq E$ *path-cycle covers*, if in the subgraph $G' = (V, E', lab)$ every vertex in G' has degree at most 2. To every such G', we associate a multiset M consisting of multisets $\langle lab(v_1), lab(v_r)\rangle$ one each for every path/cycle $p = v_1, \ldots, v_r$, $r \geqslant 1$, in G', where v_1, v_r have degree at most 1 in G' if they exist (p being a cycle otherwise). Note that every M consists of at most $|K|$ distinct *types*.

Let $F(X)$ be the set of all multisets M for all such subsets $E' \subseteq E$. Then $F(X)$ has at most $(n+1)^{|K|}$ distinct multisets each with at most n multisets of size 2. Crucial to our algorithm is the notion of *type* of a path-cycle cover. A *type* of a path is the label of its end points. Let $K = K_0 \uplus K_1 \uplus K_2$ be the set of distinct *types* where K_2 accounts for types of the form $\langle i, j\rangle$ (for $i \neq j$) is the label that stands for paths whose end points are i and j; K_0 for the empty type $\langle\rangle = \emptyset$ corresponds to a *cycle*; K_1 for types of the form $\langle i, i\rangle$ which could be either paths whose end points are both labeled i, or isolated vertices with the label i. Observe that, $K_2 = \binom{k}{2}$, $K_1 = 2k$ and $K_0 = 1$, where we distinguish between the cases of single isolated vertex of label i and multiple vertex paths with end points labeled i for technical reasons, leading to the extra factor of 2. Our notation is consistent with [EGW01] in all cases except for the empty type, since in [EGW01] cycles are not permitted.

Our objective is to count the number of path-cycle covers, $\#X[M]$, corresponding to a multiset M in the graph $val(X)$. In particular,

$$\sum_{i,j \in K} \#X(M_{i,j})$$

where $M_{i,j} = \langle\langle i,j \rangle\rangle$ is a multiset containing a single type $\langle i,j \rangle$, yields the number of Hamiltonian paths with end points coloured i, j in $val(X)$. We denote by $\#X$ the vector with $(n+1)^{|K|}$ entries with $\#X[M]$ where $M \in [0,n]^{|K|}$. Let C_o be a $(n+1)^{|K|} \times (n+1)^{|K|}$ matrix which for each pair of multisets M, M' denotes the number of ways to form M' from M under an operation $o \in \{\eta_{i,j}, \rho_{i \to j} : i, j \in [k] \wedge i \neq j\}$. C_o is defined uniquely for the two kinds of operations η, ρ and is independent of the input graph $\mathrm{val}(X)$.

Then the following is an easy consequence of the definitions:

Proposition 6.11. *The value of* $\#X$ *is given by:*

1. *if* $X = \bullet_i$ *then if* $M = \langle\langle i,i \rangle\rangle$ *then* $\#X[M] = 1$*; else* $\#X[M] = 0$.
2. *else if* $X = X_1 \oplus X_2$ *then*

$$\#X[M] = \sum_{M' \in [0,n]^{|K|} : M' \subseteq M} \#X_1[M'] * \#X_2[M \setminus M']$$

3. *else if* $X = \rho_{i \to j}(X_1)$ *then* $(C_{\rho_{i \to j}})^T \#X_1$
4. *else* $X = \eta_{i,j}(X_1)$ *then* $(C_{\eta_{i,j}})^T \#X_1$

Proof. The first item is immediate. For the second, notice that each multiset of types M in the disjoint union of two graphs is formed by picking multisets M', M'' from the two graphs respectively and taking their multi-union. Thus the distinct ways to form M are to consider all possible decompositions into sets M', M'' one from each graph. Since, this is a decomposition $M'' = M \setminus M'$, the correctness of the second item follows.

For the third and the fourth items, notice that we have a matrix C such that $C[M, M']$ is the number of ways to convert a multiset M to a multiset M'. Thus the number of ways to form M' is to take the product of $\#X[M]C[M, M']$ and add up the products over all M. This is the stated form in matrix notation. $\square$

Proposition 6.11 enables us to prove both the $\#\mathsf{SAC}^1$ and GapL upper bounds:

Lemma 6.12. *For a bounded clique-width expression* X*, for every multiset of types,* M*, the value* $\#X[M]$ *of the number of path-cycle covers at any node along the parse tree of the clique width* *expression can be computed in* $\#\mathsf{SAC}^0$ *where the inputs to the* $\#\mathsf{SAC}^0$ *circuit are entries of the matrix* C_o *for* $o \in \{\bullet_i, \oplus, \eta_{i,j}, \rho_{i\to j} : i, j \in [k] \wedge i \neq j\}$*. The number of path-cycle covers in the input graph can hence be counted in* $\#\mathsf{SAC}^1$.

Proof. (of Lemma 6.12) We will use Proposition 6.11 for every node in the clique-width decomposition X to compute $\#X[M]$ for every $M \in [0, n]^{|K|}$. For this we will need the various matrices $C_{\rho_{i\to j}}$ (constructed in Proposition 6.13) and $C_{\eta_{i,j}}$ (constructed in Lemma 6.14). The correctness of this circuit is clear from Proposition 6.11.

Next, we need to argue that the number of path-cycle covers in a bounded clique-width graph is a function in $\#\mathsf{SAC}^1$. We construct our $\#\mathsf{SAC}^1$ circuit by combining the $\#\mathsf{SAC}^0$ circuits for every node in the parse tree as given by Proposition 6.11. Notice that the resulting circuit is monotone (there are no subtractions) and the value at every gate is at most a polynomial (in n) many bits – this is because the number of path-cycle covers of an n-vertex simple graph is at most an exponential function of n, which is representable by $\mathsf{poly}(n)$ many bits. Thus the degree [3] of our circuit must also be polynomial in n. The circuit obtained thus is of arbitrary ($\mathsf{poly}(n)$) depth (since the parse tree is not necessarily balanced), and hence a naive evaluation of such a circuit is in P. However, by Proposition 2.8 this circuit can be depth-reduced to yield an upper bound of $\#\mathsf{SAC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{P} \cap \mathsf{DSPACE}(\log^2 n)$. □

We now turn to the proof of our main Theorem 1.2

Proof. (of Theorem 1.2) To count Euler tours on bounded treewidth graphs, we can count Hamiltonian cycles in the line graph (via Lemma 6.2 and Lemma 6.3). Here we need to compute the quantity $\#X[\langle\emptyset\rangle]$ (since the empty multiset represents a cycle, the path-cycle cover consisting of a single cycle must be a Hamiltonian cycle itself). This follows from Lemma 6.12. □

Proof. (of Theorem 1.3) Hamiltonian cycles can be counted in $\#\mathsf{SAC}^1$ by Lemma 6.12. Longest Cycles (Paths) can be counted by considering multisets which consist of a single cycle (respectively, path) and the minimum number of isolated vertices respectively. To see this observe that for every cycle (respectively, path) C in the graph there is a multiset consisting of a single empty type (respectively, non-empty type) and $|V(G)| - |V(C)|$ isolated vertices respectively.

Counting cycle covers is equally simple. We just need to add up the counts for multisets consisting only of empty types. This, is of course because an empty type represents a cycle.

[3]The degree of a circuit is defined inductively as follows: All input variables (here these correspond to the vertices/ edges in the graph) and constants have degree 1. The degree of a $\times$-gate is the sum of the degree of its children. The degree of a $+$-gate is the maximum of the degrees of its children.

Perfect Matchings in bipartite graphs can therefore be counted by counting the cycle covers in a biadjacency matrix. □

6.2.1 Computing $C_{\rho_{i\to j}}$ and $C_{\eta_{i,j}}$

It is easy to compute $C_{\rho_{i\to j}}$ by the following,

Proposition 6.13. $C(\rho_{i\to j})$ *is a* $\{0,1\}$*-matrix such that the entry corresponding to* M_1, M_2 *is equal to* 1 *iff* $\rho_{i\to j}(M_1) = M_2$ *(it is* 0 *otherwise).*

Let $W_{\vec{\alpha}}(t')$ denote the number of ways to form one path/cycle of type $t' \in K$, given a multiset of paths/cycles consisting of $\vec{\alpha}(t)$ paths/cycles for every type $t \in K$.

Next, we show how to compute $C_{\eta_{i,j}}$:

Lemma 6.14. *There is a Logspace Turing machine that takes input* $W_{\vec{\alpha}}(t')$ *for every* $\vec{\alpha} \in [0,n]^{|K|}, t' \in K$ *and outputs the entries of the matrix* $C_{\eta_{i,j}}$.

Proof. We show that each entry can be computed in DLOGTIME-uniform TC^0 which is contained in L (see e.g. Vollmer [Vol99]). Our main tool in this lemma is an application of polynomial interpolation.

Notice that the rows/columns of the matrix C are indexed by multisets of *types*. Here a *type* is an element from K. Therefore any such multiset can be described by a vector $\vec{\alpha}$ of length $|K|$. Here each entry of the vector represents the number of paths/cycles with that type inside the multiset.

In the following we will consistently make use of the notation, $\vec{z}^{\vec{a}}$ to denote: $\prod_{i\in I} z_i^{a_i}$, where I is the index set for both $\vec{z}, \vec{a}$.

We have the following:

Lemma 6.15. $C[M, M']$ *is the coefficient of* $\vec{x}^{\vec{c}'}\vec{y}^{\vec{c}}$ *in the following polynomial* $p_{\vec{c}',\vec{c}}(\vec{x},\vec{y})$*:*

$$\prod_{t,t'\in K} \prod_{\vec{\alpha}\in[0,n]^{|K|}} \sum_{\vec{d}_{\vec{\alpha}}} \left(W_{\vec{\alpha}}(t') x_{t'} y_{t'}^{\alpha(t)}\right)^{d_{\vec{\alpha}}(t')}$$

To fix the notation we reiterate (items 1, 2, 3, 4 were defined previously and we introduce some new notation in items 5, 6, 7, 8, 9):

1. K is the set of types, where $|K| = \binom{k}{2} + 2k + 1$.
2. $t, t' \in K$ are types in the input, output multiset (respectively M, M').

3. $\alpha(t) \in [0, n]$ is an allocation of type $t \in K$.

4. $\vec{\alpha} \in [0, n]^{|K|}$ is a possible allocation vector indexed by K in which each entry is $\alpha(t)$

5. $d_{\vec{\alpha}}(t') \in [0, n]$ is the number of paths of type t' formed from each allocation of type $\vec{\alpha}$

6. $\vec{d}_{\vec{\alpha}} \in [0, n]^{|K|}$ is a vector indexed by K in which each entry is one of $d_{\vec{\alpha}}(t')$.

7. $W_{\vec{\alpha}}(t')$ is the number of ways to form a single path/cycle of type t' from an allocation vector $\vec{\alpha}$

8. $\vec{W}_{\vec{\alpha}}$ is the vector indexed by K in which each entry is one of $W_{\vec{\alpha}}(t')$

9. $\vec{c}, \vec{c'} \in [0, n]^{|K|}$ are vectors indicating number of paths/cycles in M, M' respectively.

To see that Lemma 6.14 follows from Lemma 6.15 we use Kronecker substitution (see Fact 3) to convert the multivariate polynomial $p_{\vec{c'},\vec{c}}(\vec{x}, \vec{y})$ with $2|K|$ variables to a univariate polynomial. Then we use Lagrange interpolation to find the coefficient of an arbitrary term - in particular, the term corresponding to $\vec{x}^{\vec{c'}}\vec{y}^{\vec{c}}$ in TC^0 (see e.g. Corollary 6.5 in [HAB02]). □

Proof. (of Lemma 6.15) Consider the following expression:

$$\sum_{\vec{d} \in \mathbf{D}} \prod_{t' \in K} \prod_{\vec{\alpha} \in [0,n]^{|K|}} W_{\vec{\alpha}}(t')^{d_{\vec{\alpha}}(t')} \tag{6.1}$$

where the sum is taken over $\mathbf{D} \subseteq [0, n]^{|K|}$ consisting of all $\vec{d}$'s satisfying:

$$\forall t', \sum_{\vec{\alpha} \in [0,n]^{|K|}} d_{\vec{\alpha}}(t') = c'(t') \tag{6.2}$$

$$\forall t, \sum_{\vec{\alpha} \in [0,n]^{|K|}} \sum_{t' \in K} \alpha(t) d_{\vec{\alpha}}(t') = c(t) \tag{6.3}$$

Claim 6.16. $C[M, M']$ equals Expression (6.1).

Proof. (of Claim) The Condition 6.2 above asserts that the number of paths/cycles of type t' present in M' equals the sum over all $\vec{\alpha}$ of the number of paths/cycles of type t' using resources described by $\vec{\alpha}$; the Condition 6.3 is essentially a conservation of resource equation for every type t saying that all the resources present in M are used one way or the other in M'.

Let P, P' be path-cycle covers represented by M, M' respectively such that we can obtain P' from P. This transformation is described by a unique $\vec{d}$. Then the pair contributes precisely one to $C[M, M']$. On the other hand $\{P, P'\}$ satisfies (6.2),(6.3) so contributes exactly one to the summand corresponding to the unique $\vec{d}$ in Expression 6.1. Since the pair P, P' corresponds to a unique $\vec{d}$ and contributes exactly one, the remaining summands would evaluate to zero. This can be explained by observing that for all $\vec{d'} \neq \vec{d}$, the number of paths of type t in $\vec{d'}$ is not equal to the corresponding number in $\vec{d}$ for at least one t. Hence, they would contribute nothing to pair P, P'. $\square$

To complete the proof notice that the coefficient of $\vec{x}^{\vec{c'}}\vec{y}^{\vec{c}}$ is precisely expression 6.1 under the conditions 6.2, 6.3. Now, we explain the reasoning behind expression 6.1. We have $d_{\vec{\alpha}}(t')$ paths of type t', each of which can be formed in $W_{\vec{\alpha}}(t')$ ways. Note that each of these $d_{\vec{\alpha}}(t')$ paths are formed from different $\vec{\alpha}$ (Though the signature of each $\vec{\alpha}$ is the same, they are inherently different as they are composed of mutually exclusive vertex sets) and we consider each valid set of $d_{\vec{\alpha}}(t')$ $\vec{\alpha}$ vectors exactly once. Hence, we multiply $W_{\vec{\alpha}}(t')^{d_{\vec{\alpha}}(t')}$ to get the final count. $\square$

6.2.2 Calculation of $W_{\vec{\alpha}}(t')$

$W_{\vec{\alpha}}(t')$ denotes the number of ways to form exactly one type $t' \in K$ in M' given a multiset of types consisting of $\vec{\alpha}(t)$ types for every type $t \in K$ in M. For simplicity of notation, let $t = \langle i, j \rangle \in K$ be a type and let $\beta(i) = \alpha(\langle i, i \rangle) + \alpha^{(=0)}(\langle i, i \rangle)$ be the total number of multisets of type $\langle i, i \rangle$, where $\alpha(\langle i, i \rangle)$ (respectively $\alpha^{(=0)}(\langle i, i \rangle)$) denote paths (respectively single nodes) labeled $\langle i, i \rangle$ in $\vec{\alpha}$. Note that this distinction is not needed for types where the end points have different labels.

Lemma 6.17. *For an operation η_{i_0,j_0} in the clique-width expression and for any type $t' = \langle i, j \rangle$, $W_{\vec{\alpha}}(t')$ is given by*

$$W_{\vec{\alpha}}(\langle i, j \rangle) = [\![\langle i, j \rangle]\!]_{\vec{\alpha}} W_{\vec{\alpha}}$$

where,

$$W_{\vec{\alpha}} = \binom{\alpha(\langle i_0, j_0 \rangle) + \beta(i_0) + \beta(j_0)}{\alpha(\langle i_0, j_0 \rangle)} \alpha(\langle i_0, j_0 \rangle)!\beta(i_0)!\beta(j_0)!2^{\alpha(\langle i_0, i_0 \rangle) + \alpha(\langle j_0, j_0 \rangle)}$$

and, $[\![\langle i, j \rangle]\!]_{\vec{\alpha}}$ is given by [4], [5]

- $[\![\langle a, b \rangle]\!]_{\vec{\alpha}} = [\![\beta(a) = \beta(b)]\!]$
- $[\![\langle a, a \rangle^{(=0)}]\!]_{\vec{\alpha}} = [\![\alpha^{(=0)}_{a,a} = 1 \wedge \alpha(\langle a, b \rangle) = 0]\!]$
- $[\![\langle a, a \rangle]\!]_{\vec{\alpha}} = [\![\beta(a) = \beta(b) + 1]\!]$

[4] In this section, the notation $[\![S]\!]$ represents the Boolean value of the statement S. $[\![t]\!]_{\vec{\alpha}}$ represents a Boolean valued normalizing factor associated with the type t under the allocation vector $\vec{\alpha}$.

[5] We adopt a convention in which types t' (other than the type $\langle i_0, j_0 \rangle$) not explicitly included in the expressions have an allocation $\alpha_{t'}$ equalling zero.

- $[\![\langle i,a\rangle]\!]_{\vec{\alpha}} = [\![(\alpha(\langle i,a\rangle) = 1 \wedge \beta(a) = \beta(b)) \vee (\alpha(\langle i,b\rangle) = 1 \wedge \beta(a) = \beta(b)+1)]\!]$
- $[\![\langle i,j\rangle]\!]_{\vec{\alpha}} = [\![(\alpha(\langle i,a\rangle) = 1 \wedge \alpha(\langle a,j\rangle) = 1 \wedge \beta(a) = \beta(b)+1) \vee (\alpha(\langle i,a\rangle) = 1 \wedge \alpha(\langle b,j\rangle) = 1 \wedge \beta(a) = \beta(b))]\!]$
- $[\![\langle i,i\rangle]\!]_{\vec{\alpha}} = [\![(\alpha(\langle i,a\rangle) = 2 \wedge \beta(a) = \beta(b)+1) \vee (\alpha(\langle i,a\rangle) = 1 \wedge \alpha(\langle i,b\rangle) = 1 \wedge \beta(a) = \beta(b))]\!]$
- $[\![\langle\emptyset\rangle]\!]_{\vec{\alpha}} = [\![\beta(a) = \beta(b)]\!]$

where, $\{a,b\} = \{i_0, j_0\}$ *in some order.*

Proof. (of Lemma 6.17) Let's look at $W_{\vec{\alpha}}(\langle a,a\rangle)$ in detail. The $W_{\vec{\alpha}}(t)$ for all the other types t are computed similarly. Type $\langle a,a\rangle$ can be formed from the alternating sequence of types $\langle a,a\rangle, \langle b,b\rangle, \langle a,a\rangle \ldots \langle a,a\rangle$ interleaved with some (possibly zero) $\langle a,b\rangle$ types. Thus, the equality $\beta(a) = \beta(b)+1$ should hold while $\alpha(\langle a,b\rangle)$ can be any arbitrary non-negative integer. When $\alpha(\langle a,b\rangle) = 0$, the condition $\alpha(\langle a,a\rangle) \geqslant 1 \vee \alpha(\langle a,a\rangle)^{(=0)} > 1$ should hold to ensure that we are not considering the type $(\langle a,a\rangle)^{(=0)}$.

The number of ways of interspersing $\alpha(\langle a,b\rangle)$ types among $\beta(a)+\beta(b)$ types is $\binom{\alpha(\langle a,b\rangle)+\beta(a)+\beta(b)}{\alpha(\langle a,b\rangle)}$. We can do this for all permutations of the $\langle a,b\rangle, \langle a,a\rangle$ and $\langle b,b\rangle$ types hence we multiply by: $\alpha(\langle a,b\rangle)!\beta(a)!\beta(b)!$. Finally, we can flip the orientation of paths of types $\langle a,a\rangle$ and $\langle b,b\rangle$ as they are equivalent respectively to their flipped orientations. Note that single nodes cannot be flipped. The proof is therefore completed by multiplying with: $2^{\alpha(\langle a,a\rangle)+\alpha(\langle b,b\rangle)}$. Lastly, a boundary case occurs when $\alpha(\langle a,b\rangle) = 0$ where every path can be flipped. Here, it is easy to see that in considering every permutation of types while accounting for flips, we end up counting each path twice (including its reverse). Hence, in this case we divide by 2. □

6.3 Discussion

We gave #SAC^1 and GapL bounds for counting Euler tours on bounded treewidth graphs, by reducing it to the problem of counting Hamiltonian cycles in bounded clique width graphs. Along the way, these techniques also yield algorithms to count perfect matchings (in bipartite graphs), cycle covers and longest paths in graphs of bounded clique width in #SAC^1.

- Can the #SAC^1 and GapL bounds be improved, to say, Logspace?
- How far can the Euler tour result be extended? To bounded clique-width graphs? Chordal graphs?

- For the problem of counting Euler tours on bounded treewidth graphs, we have a GapL bound whereas for longest paths, perfect matchings and cycle covers our bound is still[6] $\#\mathsf{SAC}^1$ for arbitrary bounded clique width graphs. This is because, for bounded clique width graphs that are line graphs of bounded treewidth graphs we know how to get a balanced parse tree for the clique width expression whereas we do not know how to get such a parse tree for arbitrary bounded clique width graphs. A result of Courcelle and Kanté [CK07] shows that every graph of clique width k is the value of a 3-balanced clique width expression of clique-width at most $k2^{k+1}$. Can such an expression be found in Logspace? If yes, then we can show a GapL bound for all the aforementioned problems on bounded clique width graphs.
- What is the parameterized complexity of counting Euler tours? Our algorithm yields an $O(n^{k^2})$ time algorithm for counting Euler Tours on graphs of treewidth k. Is there a $2^{f(k)}\mathsf{poly}(n)$ algorithm for the same? What is the complexity of Euler tours parameterized on other parameters like vertex cover?

[6]Note however that $\#\mathsf{SAC}^1$ and GapL are orthogonal bounds, both containing NL and are contained in NC^2

Chapter 7

Conclusion and Open Ends

We summarize and conclude this thesis in this chapter. As alluded to in Chapter 1, our main thrust in this thesis has been to understand the complexity of counting problems in some restricted settings. We mentioned some concrete open problems related to the content of the specific chapter in the discussion section of each chapter. Here we revisit some of them and mention some more open problems that are more general in scope.

We started in Chapter 3 with the problem of inferring properties of succinctly represented numbers and matrices:

Bits of SLPs: We gave improved upper bounds for BitSLP, a problem first introduced in [ABKPM09]. We showed that this problem lies in the fourth level of the counting hierarchy. From [ABKPM09], it is known that this problem is #P-hard, and non-uniformly it lies in $\mathsf{PH}^{\mathsf{PP}}$, albeit with an exponential-size advice[1]. Can we solve BitSLP with polynomial-sized advice? Can we obtain a uniform $\mathsf{PH}^{\mathsf{PP}}$ upper bound?

Succinct Integer Powering: We showed that given an n-bit integer, raising it to an n-bit exponent is in the fourth level of the counting hierarchy. Can this bound be improved? We remark in this regard, that there are some surprising things that one can compute, regarding large powers of integers [HKR10] – the most significant bits of 2^n (n specified in binary) in base 3 and that of the Fibonacci numbers can be computed in time polynomial in n. These results rely on Baker's inapproximability results on transcendental numbers [Bak90].

Succinct Matrix Powering: We showed that obtaining a bit of large powers of a constant size matrix can also be done in the fourth level of counting hierarchy via two different algorithms – one using the Cayley-Hamilton Theorem and the other by observing that we can construct

[1]Note that this non-uniform bound is obtained from the non-uniform TC^0circuits for division, where the advice is just the CRR basis product. In the case of the succinct version, this product is an exponential-bit number.

a small arithmetic circuit that computes such powers, and hence is an instance of BitSLP itself. Is evaluating restricted SLPs like the matrix powering SLP also $\#\mathsf{P}$-hard? Note in this context that we know polynomial-time algorithms for checking positivity of 2×2 and 3×3 matrix powers [GOW15], but this still doesn't rule out the $\#\mathsf{P}$-hardness of say computing an arbitrary bit of such small matrix powers. It is also interesting to note that even for non-constant sized matrices (say $n \times n$ with $\mathsf{poly}(n)$-bit entries, raised to $\mathsf{poly}(n)$-bit powers), the same CH_3 upper bound holds by producing a small arithmetic circuit on which we can run a BitSLP procedure. However, in the verbose version, this problem is known to be hard for GapL [MV97] (and hence also hard for NC^1), whereas constant size matrix powering is in TC^0 [MP00]. This suggests that the succinct and the verbose versions behave differently for these problems, and gives some hope that there might be more efficient algorithms for evaluating such restricted arithmetic circuits.

Sum of Square Roots: We gave a $\mathsf{PH}^{\mathsf{PP}}$ reduction of the Sum of Square Roots problem to obtaining a bit of 2×2 matrix powers. Can we reduce the complexity of this reduction to just PH? A completely mathematical approach to settle the sum of square roots problem would be to prove a statement of the form: Given $\boldsymbol{a} = \langle a_1, a_2, \ldots, a_n \rangle$, where each of the a_i's are distinct n-bit integers and $\boldsymbol{\sigma} = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$, there exists a constant c such that $\sum_{i=1}^{n} \sigma_i a_i \geqslant r(n, c)$. If this statement were true for $r(n, c) = \frac{1}{2^{n^c}}$, then we can just calculate each square root to a polynomial-bit precision and evaluate the sum and be guaranteed to decide the positivity of the sum correctly. Currently we know that $r(n, c)$ has to be at least $\frac{1}{2^{2^{n^c}}}$. Can $r(n, c)$ be improved to $\frac{1}{2^{n^{(\log^c n)}}}$?

Integer SLP minimization: What is the complexity of the following problem : Given an SLP C representing a number and a size bound s, is there an SLP C' of size at most s that also computes the same number? It is easy to see that this problem is in $\mathsf{NP}^{\mathsf{coRP}}$: Guess an SLP of size at most s and verify using an EqSLP oracle that they are the same. Is there a better upper bound? Is this problem NP-hard? Again, in the world of Boolean circuits, circuit minimization is a tantalizing open problem: It is unlikely to be in P: A polynomial time algorithm would break any pseudorandom function generator; and a proof of NP-hardness will yield elusive circuit lower bounds[KC00]. It is possible that these problems are easier in the case of integer sequences.

Lower bounds for Addition-Multiplication chains (AMC): Monotone circuit lower bounds are one of the success stories of the Circuit Lower Bounds enterprise – We do not know if $\mathsf{P} \neq \mathsf{NP}$, much less $\mathsf{NP} \not\subseteq \mathsf{P/poly}$, but we do know that $\mathsf{NP} \not\subseteq \mathsf{monotone-P/poly}$[Raz90]. This paradigm has yielded successful results even in the setting of arithmetic circuit complexity, where we know that monotone arithmetic circuits require exponential size to compute the Permanent polynomial[JS82], whereas in the non-monotone setting, it is still open whether the Permanent polynomial requires large arithmetic circuits[Agr06]. In the setting of integer sequences, the analogous notion is that of Addition-Multiplication chains which are arithmetic circuits without subtraction. Let $\tau^+(N)$ denote the size of the smallest AMC computing N. Saranurak and Jindal [JS12] have shown that there is an explicit integer sequence N such that $\tau^+(N) > \tau(N) + 1$. Can this gap be made exponential?

In Chapter 4, we studied Boolean skew circuits of width $\leqslant 7$, i.e., SK^1 to SK^7. We showed that $\mathsf{SK}^1 \subsetneq \mathsf{SK}^2 \subsetneq \mathsf{SK}^3 \subsetneq \mathsf{SK}^4$ by exhibiting appropriate functions that witness the separations. Our mail goal in studying skew circuits was to get a finer classification of problems in NC^1.

Lower bounds for Majority: We exhibited exponential lower bounds for any width 3 skew circuit computing the Parity function on n variables. However, we do not get the same lower bound for Majority since we do not know of a reduction from Parity to Majority that is implementable via width 3 skew circuits. Is there such a reduction, or can we show exponential lower bounds for Majority using an alternate approach? What about arbitrary symmetric functions? These are known to be in TC^0. Giving a skew circuit characterization of symmetric functions will help us understand the fine structure inside NC^1 better.

Lower bounds for width 4 PBPs: A longstanding open problem since Barrington's result on the equivalence of PBPs and BPs is to understand the power of width 4 PBPs. Can they compute the AND function?

AC^0, ACC^0 *and skew circuits:* It is easy to see that a depth-d AC^0 circuit can be simulated by a width $d+1$ skew circuit and by our result, all of AC^0 is in SK^7. Is there a $4 \leqslant r < 7$ such that $\mathsf{AC}^0 \subseteq \mathsf{SK}^r$? Similarly, we know that parity can be computed by width 4 skew circuits. Extending the same argument, one can show that MOD_m is in $\mathsf{BP}^m \subseteq \mathsf{SK}^{2m}$ and all of this is contained in SK^7. We conjecture that this inclusion cannot be improved.

Arithmetic skew circuits: Analyzing the result of Ben-Or and Cleve via arithmetic skew circuits, one gets that all of GapNC^1 is contained in arithmetic skew circuits ($a\mathsf{SK}^4$) of width 4. Is there a finer separation of these classes? The question of whether $\mathsf{GapNC}^1 = \mathsf{NC}^1$ has been open for the last three decades – the current best inclusion being $\mathsf{GapNC}^1 \subseteq \mathsf{NC}[\log n \log^* n]$ which has two proofs, one due to Jung [Jun85] and yet another very simple proof due to Agrawal, Allender and Datta [All04]. Does the skew circuit characterizations of GapNC^1 and NC^1 help in this regard? Can we prove that $a\mathsf{SK}^4 \subseteq \mathsf{SK}^7$?

In Chapter 5, we studied the parallel complexity of computing certain linear algebraic invariants like the Determinant when the underlying graph of the matrix is of bounded treewidth, and proved some tight bounds. Our main tool in this regard was the powerful meta-theorem machinery of Elberfeld, Jakoby and Tantau [EJT10].

Avoiding Meta Theorems: A natural question regarding our technique to get the L upper bound is if we can side-step the MSO approach. In other words, is there a simple dynamic programming algorithm to compute the Determinant of bounded treewidth matrices? This naturally leads us to the next point:

Parameterized Complexity: Our algorithm utilizes $O(f(k)\log n)$ space, where $f(k)$ is a function of the treewidth k of the underlying graph. Can we make this algorithm run in $O(f(k)+\log n)$ space? Then such an algorithm would have running time $O(f(k)n)$, which is fixed parameter tractable. Many graph problems have been studied via the lens of fixed parameter tractability. Recently, there has been some interest in understanding the complexity of the Permanent parameterized by the number of non-zero entries of the matrix [DHM+14]. To the best of our knowledge, the Determinant and related linear algebraic invariants have not been studied in the parameterized complexity literature.

Kernelization: The use of meta-theorem machinery leads to *galactic* constants. Is there an algorithm for the Determinant and other linear algebraic invariants investigated in this thesis, parameterized on the treewidth of the graph that runs in space $O(f(k)+\log n)$ or at least $O(f(k)\log n)$, where $f(k) = \mathsf{poly}(k)$? If not, can we refute the existence of such algorithms subject to some widely-believed hypotheses like Exponential Time Hypothesis (ETH) or the Strong Exponential Time Hypothesis (SETH)?

All these questions make sense if we instead parameterise the Determinant on other parameters of the underlying graph like clique width, rank width, etc and we believe these investigations will help us understand the complexity of Determinant and associated linear algebraic invariants better.

In Chapter 6, we studied the question of counting Euler tours on bounded treewidth graphs, which is known to be #P-complete on general graphs. We crucially used the fact that line graphs of bounded treewidth graphs are of bounded clique width and used a simple gadget to reduce the problem of counting Euler tours on bounded treewidth graphs to counting Hamiltonian cycles on bounded clique width graphs. We gave two orthogonal upper bounds, namely $\#\mathsf{SAC}^1$ and GapL.

Obtaining Balanced clique width expressions: For the problem of counting Euler tours on bounded treewidth graphs, we have a Logspace-uniform $\#\mathsf{SAC}^1$ bound whereas for longest paths, perfect matchings and cycle covers our bound is P-uniform $\#\mathsf{SAC}^1$ for arbitrary bounded clique width graphs. This is because, for bounded clique width graphs that are line graphs of bounded treewidth graphs we know how to get a balanced parse tree for the clique width expression in logspace, whereas we do not know how to get such a parse tree for arbitrary bounded clique width graphs. A result of Courcelle and Kanté [CK07] shows that every graph of clique width k is the value of a 3-balanced clique width expression of clique-width at most $k2^{k+1}$ and such a balanced expression is obtainable in polynomial time. Can such an expression be found in logspace? If yes, then we can show a logspace-uniform $\#\mathsf{SAC}^1$ bound for all the aforementioned problems on bounded clique width graphs.

FPRAS for counting Euler tours: The Monte Carlo Markov Chain machinery has been extensively used to obtain randomized approximation algorithms for counting problems that are $\#\mathsf{P}$-hard. The idea is to set up a Markov chain on the solution space (in our case these are Euler tours) and show that a random walk on this Markov chain mixes well. This yields a simple and efficient algorithm to approximate the count of Euler tours in a graph and equivalently allows us to sample an Euler tour of the graph uniformly at random. However, since Brightwell and Winkler's result [BW05] that counting Euler tours is $\#\mathsf{P}$-complete, an FPRAS for this problem has been elusive. Recently, Ge and Štefankovič [GŠ12] showed that it suffices to find such an FPRAS for 4-regular planar graphs to solve the case of general graphs.

Appendix A

Linear Fractional Transformations (LFTs)

Here we give a brief introduction to LFTs based on the expositions in [EP97, Pot97, Pot99], concentrating only on the aspects required in our scenario in Chapter 3.

A linear fractional transformation is a function mapping $y \mapsto \frac{ay+c}{by+d}$ for reals (and preferably integers) a, b, c, d and the associated matrix is $\begin{pmatrix} a & c \\ b & d \end{pmatrix}$. The interesting thing about LFTs is that the matrix corresponding to the composition of two LFTs is the usual product of the matrices corresponding to the two LFTs. In other words, if the matrix corresponding to $\phi_i(y)$ is $\begin{pmatrix} a_i & c_i \\ b_i & d_i \end{pmatrix}$ (for $i = 1, 2$), then a matrix corresponding to $\phi_1\phi_2(y)$ (which abbreviates $\phi_1(\phi_2(y))$) is $\begin{pmatrix} a_1 & c_1 \\ b_1 & d_1 \end{pmatrix}\begin{pmatrix} a_2 & c_2 \\ b_2 & d_2 \end{pmatrix}$, as can be easily verified. Here, we deal only with nonsingular LFTs i.e. LFTs whose matrix has a non-zero determinant. An LFT is said to be positive if all four entries in its matrix have the same sign.

Let ϕ be an LFT and let $M = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$ be its matrix. ϕ acts as a bijection between any interval $[p, q]$ and a subset of the extended reals i.e. the usual reals augmented with ∞. Further, this subset is also an interval (possibly including ∞): either $[\phi(p), \phi(q)]$ or $[\phi(q), \phi(p)]$. Notice that we do not claim that there is a linear order on the reals augmented with ∞. Instead, we refer to these sets as “intervals” in the same sense that connected subsets of the unit circle can be called intervals.

For a concrete example, $\phi[0,\infty]$ is the interval $[\frac{a}{b},\frac{c}{d}]$ if $\det(M) < 0$ and the interval $[\frac{c}{d},\frac{a}{b}]$ if $\det(M) > 0$. Notice that $\phi(\infty)$ is taken to be $\lim_{y\to\infty}\phi(y) = \frac{a}{b}$. Notice also that $(-1/x)[-1,1]$ is the interval $[1,-1]$ containing ∞.

An LFT is said to be refining for an interval $[p,q]$ if $\phi[p,q] \subseteq [p,q]$. We will need the following two propositions from [Pot97]:

Proposition A.1. *Given two non-trivial intervals* $[p,q]$ *and* $[r,s]$ *with* $p \neq q$ *and* $r \neq s$*, there exists an LFT* ϕ *with* $\phi[p,q] = [r,s]$.

Proposition A.2. *For LFTs* ϕ *and* ψ *we have* $\phi[0,\infty] \supseteq \psi[0,\infty]$ *iff* $\psi = \phi\gamma$ *for a positive LFT* γ.

Thus for any sequence of nested intervals $[p_0,q_0] \supseteq [p_1,q_1] \supseteq \ldots \supseteq [p_n,q_n] \supseteq \ldots$ we have $[p_n,q_n] = \phi_0\phi_1\ldots\phi_n[0,\infty]$ where ϕ_0 is an LFT and all other ϕ_i's are positive LFTs.[1] Thus if a sequence of nested intervals converges to a real number r, then the corresponding infinite sequence of LFTs or the corresponding infinite product of matrices represents r; and the initial finite subsequence of LFTs applied to the interval $[0,\infty]$ yield increasingly finer approximations to r.

LFTs are closely related to continued fractions; in fact, the continued fraction

$$a_0 + \cfrac{b_0}{a_1 + \cfrac{b_1}{\ddots}}$$

corresponds to the LFT $\begin{pmatrix} a_0 & b_0 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} a_1 & b_1 \\ 1 & 0 \end{pmatrix}\ldots$.

An LFT for the square root function is:

$$\sqrt{x} \quad \equiv \quad \prod_{n=0}^{\infty} \begin{pmatrix} x & x \\ 1 & x \end{pmatrix}$$

for $x \in [1,\infty]$. This differs slightly from the LFT specified in [Pot97, Pot99]. We establish its correctness below, and conclude by proving Lemma 3.25.

To see that this LFT ϕ is an LFT for the square root function, we first establish a bound on the length of the nth convergent. We use the following notation: $\|[p,q]\| = q-p$ denotes the length of the interval $[p,q]$. The following two subsections show that $\|\phi^n[0,\infty]\| \to 0$ as $n \to \infty$.

[1]We call $[p_n,q_n]$, the nth convergent of the LFT sequence ϕ.

A.1 Length of the n^{th} convergent

Let $M_i = \begin{pmatrix} a_i & c_i \\ b_i & d_i \end{pmatrix}$ and $P_i = \prod_{j=0}^{i-1} M_j = \begin{pmatrix} A_i & C_i \\ B_i & D_i \end{pmatrix}$. Then the length of the interval $[p_n, q_n] = \prod_{i=0}^{n} M_i[0,\infty] = P_n M_n[0,\infty]$ is given by:

$$\begin{aligned}
\|P_n M_n[0,\infty]\| &= \left\| \begin{pmatrix} A_n & C_n \\ B_n & D_n \end{pmatrix} \begin{pmatrix} a_n & c_n \\ b_n & d_n \end{pmatrix} ([0,\infty]) \right\| \\
&= \left\| \begin{pmatrix} A_n a_n + C_n b_n & A_n c_n + C_n d_n \\ B_n a_n + D_n b_n & B_n c_n + D_n d_n \end{pmatrix} ([0,\infty]) \right\| \\
&= \left| \frac{A_n a_n + C_n b_n}{B_n a_n + D_n b_n} - \frac{A_n c_n + C_n d_n}{B_n c_n + D_n d_n} \right| \\
&= \left| \frac{(A_n D_n - B_n C_n)(a_n d_n - b_n c_n)}{(B_n a_n + D_n b_n)(B_n c_n + D_n d_n)} \right|
\end{aligned}$$

A.2 Length of the n^{th} convergent for Square Root and the Proof of Lemma 3.25

Using the notation above with $M_i = \begin{pmatrix} x & x \\ 1 & x \end{pmatrix}$, we get

$$\begin{aligned}
\|[p_n, q_n]\| &= \left| \frac{(A_n D_n - B_n C_n)(x^2 - x)}{(xB_n + D_n)(xB_n + xD_n)} \right| \\
&< \left| \frac{(A_n D_n - B_n C_n)(x^2 - x)}{(xB_n)(xD_n)} \right| \\
&= \left| \left(\frac{A_n}{B_n} - \frac{C_n}{D_n} \right) \left(1 - \frac{1}{x} \right) \right| \\
&= \left| (q_{n-1} - p_{n-1}) \left(1 - \frac{1}{x} \right) \right|
\end{aligned}$$

Thus, inductively, $q_n - p_n < |(q_0 - p_0)\left(1 - \frac{1}{x}\right)^n|$.

Thus $\phi^n(y) \to y_0$ for some y_0 and all $y \in [0,\infty]$. In particular, $\phi^n(y_0) \to y_0$ and thus $\phi^{n+1}(y) \to \phi(y_0)$ as $n \to \infty$. Thus, $\phi(y_0) = y_0$, so that

$$\frac{xy_0 + x}{y_0 + x} = y_0$$

Hence $x = y_0^2$.

This establishes that ϕ is a LFT for the square root function.

Now recall Lemma 3.25, which states that if $[p_n(x), q_n(x)]$ denotes the n^{th} convergent for the matrix sequence $M_1, M_2, \ldots$ where each $M_i = L(x) = \begin{pmatrix} x & x \\ 1 & x \end{pmatrix}$, then $q_n(x) - p_n(x) < x\left(1 - \frac{1}{x}\right)^{n+1}$. Thus if $x \in [1,2]$, then $0 \leqslant q_n(x) - p_n(x) < 2^{-n}$, and for all n, $\sqrt{x} \in [p_n(x), q_n(x)]$. Furthermore, $p_n(x) = (L(x)^n)_{1,2}/(L(x)^n)_{2,2}$.

From the foregoing, we have that $q_n(x) - p_n(x) < |(q_0(x) - p_0(x))\left(1 - \frac{1}{x}\right)^n|$. But $[p_0(x), q_0(x)] = \begin{pmatrix} x & x \\ 1 & x \end{pmatrix}[0, \infty] = [1, x]$. This yields $q_n(x) - p_n(x) < (x-1)\left(1 - \frac{1}{x}\right)^n \leqslant x\left(1 - \frac{1}{x}\right)\left(1 - \frac{1}{x}\right)^n = \left(1 - \frac{1}{x}\right)^{n+1}$.

The other parts of Lemma 3.25 follow immediately.

Bibliography

[AAD00] Manindra Agrawal, Eric Allender, and Samir Datta. On TC^0, AC^0, and Arithmetic circuits. *Journal of Computer and System Sciences*, 60(2):395–421, 2000.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[ABD14] Eric Allender, Nikhil Balaji, and Samir Datta. Low-depth uniform threshold circuits and the bit-complexity of straight line programs. In *Mathematical Foundations of Computer Science 2014*, pages 13–24. Springer, 2014.

[ABKPM09] Eric Allender, Peter Bürgisser, Johan Kjeldgaard-Pedersen, and Peter Bro Miltersen. On the complexity of numerical analysis. *SIAM J. Comput.*, 38(5):1987–2006, 2009.

[ADKK12] Vikraman Arvind, Bireswar Das, Johannes Köbler, and Sebastian Kuhnert. The isomorphism problem for k-trees is complete for logspace. *Information and Computation*, 217:1–11, 2012.

[AEB87] T. Aardenne-Ehrenfest and N.G. Bruijn. Circuits and trees in oriented linear graphs. In Ira Gessel and Gian-Carlo Rota, editors, *Classic Papers in Combinatorics*, Modern Birkhäuser Classics, pages 149–163. Birkhäuser Boston, 1987.

[Agr06] Manindra Agrawal. Determinant versus permanent. In *Proceedings of the 25th International Congress of Mathematicians (ICM)*, volume 3, pages 985–997, 2006.

[AJMV98] Eric Allender, Jia Jiao, Meena Mahajan, and V Vinay. Non-commutative arithmetic circuits: depth reduction and size lower bounds. *Theoretical Computer Science*, 209(1):47–86, 1998.

[Ajt83] Miklós Ajtai. σ_1^1-formulae on finite structures. *Annals of pure and applied logic*, 24(1):1–48, 1983.

[All04] Eric Allender. Arithmetic circuits and counting complexity classes. *Complexity of computations and proofs*, 13:33–72, 2004.

[AS05] Eric Allender and Henning Schnoor. The complexity of the BitSLP problem. Unpublished Manuscript, 2005.

[Bak90] Alan Baker. *Transcendental number theory*. Cambridge University Press, 1990.

[Bar85] David A Barrington. Width-3 permutation branching programs. Technical Memo MIT/LCS/TM-293, Massachusetts Institute of Technology, Laboratory for Computer Science, 1985.

[Bar89] D.A. Barrington. Bounded-width polynomial-size branching programs can recognize exactly those languages in NC^1. *J. Comp. System Sci.*, 38:150–164, 1989.

[BCH86] P. W. Beame, S. A. Cook, and H. J. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15:994–1003, 1986.

[BD15] Nikhil Balaji and Samir Datta. Bounded treewidth and space-efficient linear algebra. In *Theory and Applications of Models of Computation*, pages 297–308. Springer, 2015.

[BDFP86] Allan Borodin, Danny Dolev, Faith E Fich, and Wolfgang Paul. Bounds for width two branching programs. *SIAM Journal on Computing*, 15(2):549–560, 1986.

[BDG15] Nikhil Balaji, Samir Datta, and Venkatesh Ganesan. Counting euler tours on bounded treewidth graphs. manuscript, 2015.

[Ber84] Stuart J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Inf. Process. Lett.*, 18(3):147–150, 1984.

[BKL15] Nikhil Balaji, Andreas Krebs, and Nutan Limaye. Skew circuits of small width. In *Computing and Combinatorics*, pages 199–210. Springer, 2015.

[Bro05] Alex Brodsky. An impossibility gap between width-4 and width-5 permutation branching programs. *Information Processing Letters*, 94(4):159–164, May 2005.

[BRS95] Richard Beigel, Nick Reingold, and Daniel Spielman. Pp is closed under intersection. *Journal of Computer and System Sciences*, 50(2):191–202, 1995.

[Bür09] Peter Bürgisser. On defining integers and proving arithmetic circuit lower bounds. *Computational Complexity*, 18(1):81–103, 2009.

[BW05] Graham Brightwell and Peter Winkler. Counting eulerian circuits is #p-complete. In *ALENEX/ANALCO*, pages 259–262, 2005.

[CCM12] Prasad Chebolu, Mary Cryan, and Russell Martin. Exact counting of euler tours for generalized series-parallel graphs. *J. Discrete Algorithms*, 10:110–122, 2012.

[CDL01] A. Chiu, G. I. Davida, and B. Litow. Division in logspace-uniform NC^1. *ITA*, 35(3):259–275, 2001.

[CF12] Yijia Chen and Jorg Flum. On the ordered conjecture. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, pages 225–234. IEEE Computer Society, 2012.

[CK07] Bruno Courcelle and Mamadou Moustapha Kanté. Graph operations characterizing rank-width and balanced graph expressions. In *Graph-theoretic concepts in computer science*, pages 66–75. Springer, 2007.

[CLO92] David Cox, John Little, and Donal O'shea. *Ideals, varieties, and algorithms*, volume 3. Springer, 1992.

[CM87] Stephen A. Cook and Pierre McKenzie. Problems complete for deterministic logarithmic space. *J. Algorithms*, 8(3):385–394, 1987.

[CO00] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1):77–114, 2000.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

[DDN13] Bireswar Das, Samir Datta, and Prajakta Nimbhorkar. Log-space algorithms for paths and matchings in k-trees. *Theory Comput. Syst.*, 53(4):669–689, 2013.

[DHM$^+$14] Holger Dell, Thore Husfeldt, Dániel Marx, Nina Taslaman, and Martin Wahlén. Exponential time complexity of the permanent and the tutte polynomial. *ACM Transactions on Algorithms (TALG)*, 10(4):21, 2014.

[DKLM10] Samir Datta, Raghav Kulkarni, Nutan Limaye, and Meena Mahajan. Planarity, determinants, permanents, and (unique) matchings. *TOCT*, 1(3), 2010.

[DMS94] P. Dietz, I Macarie, and J. Seiferas. Bits and relative order from residues, space efficiently. *Information Processing Letters*, 50(3):123–127, 1994.

[DP12] S. Datta and R. Pratap. Computing bits of algebraic numbers. In *TAMC*, pages 189–201, 2012.

[EF95] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Perspectives in Mathematical Logic. Springer, 1995.

[EGW01] Wolfgang Espelage, Frank Gurski, and Egon Wanke. How to solve np-hard graph problems on clique-width bounded graphs in polynomial time. In *WG*, pages 117–128. Springer, 2001.

[EJT10] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *FOCS*, pages 143–152, 2010.

[EJT12] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In *STACS'12 (29th Symposium on Theoretical Aspects of Computer Science)*, volume 14, pages 66–77. LIPIcs, 2012.

[EP97] A. Edalat and P. J. Potts. A new representation for exact real numbers. *Electronic Notes in Theoretical Computer Science*, 6:119–132, 1997.

[Ete95] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. In *Structure in Complexity Theory Conference, 1995., Proceedings of Tenth Annual IEEE*, pages 2–11. IEEE, 1995.

[Ete97] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *J. Comput. Syst. Sci.*, 54(3):400–411, 1997.

[Ete15] K. Etessami. Fixed point computation problems for algebraically-defined functions, and their computational complexity. Talk presented at Indo-UK workshop on Computational Complexity Theory, IMSc, Chennai, 2015.

[EY10] K. Etessami and M. Yannakakis. On the complexity of Nash equilibria and other fixed points. *SIAM J. Comput.*, 39(6):2531–2597, 2010.

[FFK94] Stephen A Fenner, Lance J Fortnow, and Stuart A Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48(1):116–148, 1994.

[FG65] Delbert Fulkerson and Oliver Gross. Incidence matrices and interval graphs. *Pacific journal of mathematics*, 15(3):835–855, 1965.

[FG06] Jörg Flum and Martin Grohe. *Parameterized complexity theory*, volume 3. Springer, 2006.

[FKL07] Uffe Flarup, Pascal Koiran, and Laurent Lyaudet. On the expressive power of planar perfect matching and permanents of bounded treewidth matrices. *Algorithms and Computation*, pages 124–136, 2007.

[FSS84] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[GOW15] Esther Galby, Joël Ouaknine, and James Worrell. On matrix powering in low dimensions. In *32nd International Symposium on Theoretical Aspects of Computer Science*, page 329, 2015.

[GŠ12] Qi Ge and Daniel Štefankovič. The complexity of counting eulerian tours in 4-regular graphs. *Algorithmica*, 63(3):588–601, 2012.

[GW07] Frank Gurski and Egon Wanke. Line graphs of bounded clique-width. *Discrete Mathematics*, 307(22):2734–2754, 2007.

[GW13] Oded Goldreich and Avi Wigderson. On the size of depth-three boolean circuits for computing multilinear functions. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 20, page 43, 2013.

[HAB02] W. Hesse, E. Allender, and D.A.M. Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *Journal of Computer and System Sciences*, 65:695–716, 2002.

[Hal76] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1-2):171–186, 1976.

[Has86] Johan Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 6–20. ACM, 1986.

[HBM$^+$10] P. Hunter, P. Bouyer, N. Markey, J. Ouaknine, and J. Worrell. Computing rational radical sums in uniform TC^0. In *FSTTCS*, pages 308–316, 2010.

[Her06] Israel Nathan Herstein. *Topics in algebra*. John Wiley & Sons, 2006.

[HJP93] J Hastad, Stasys Jukna, and Pavel Pudlák. Top-down lower bounds for depth 3 circuits. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 124–129. IEEE, 1993.

[HKR10] M. Hirvensalo, J. Karhumäki, and A. Rabinovich. Computing partial information out of intractable: Powers of algebraic numbers as an example. *Journal of Number Theory*, 130:232–253, 2010.

[HMP$^+$87] András Hajnal, Wolfgang Maass, Pavel Pudlák, Márló Szegedy, and György Turán. Threshold circuits of bounded depth. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 99–110. IEEE, 1987.

[HNW65] Frank Harary and C St JA Nash-Williams. On eulerian and hamiltonian graphs and line graphs. *Canadian Mathematical Bulletin*, 8:701–709, 1965.

[HV06] A. Healy and E. Viola. Constant-depth circuits for arithmetic in finite fields of characteristic two. In *STACS 2006*, pages 672–683. Springer, 2006.

[Jeř12] Emil Jeřábek. Root finding with threshold circuits. *Theoretical Computer Science*, 462:59–69, 2012.

[Joh98] Öjvind Johansson. Clique-decomposition, nlc-decomposition, and modular decomposition-relationships and results for random graphs. In *Congr. Numer.* Citeseer, 1998.

[Joh01] Öjvind Johansson. *Graph decomposition using node labels*. Tekniska högsk., 2001.

[JS82] Mark Jerrum and Marc Snir. Some exact complexity results for straight-line computations over semirings. *Journal of the ACM (JACM)*, 29(3):874–897, 1982.

[JS89] Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM journal on computing*, 18(6):1149–1178, 1989.

[JS12] G. Jindal and T. Saranurak. Subtraction makes computing integers faster. *CoRR*, abs/1212.2549, 2012.

[Jun85] Hermann Jung. Depth efficient transformations of arithmetic into boolean circuits. In *Fundamentals of Computation Theory*, pages 167–174. Springer, 1985.

[KC00] Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 73–79. ACM, 2000.

[KP07] P. Koiran and S. Perifel. The complexity of two problems on arithmetic circuits. *Theor. Comput. Sci.*, 389(1-2):172–181, 2007.

[KP11] P. Koiran and S. Perifel. Interpolation in Valiant's theory. *Computational Complexity*, 20(1):1–20, 2011.

[KS12] N. Kayal and C. Saha. On the sum of square roots of polynomials and related problems. *TOCT*, 4(4):9, 2012.

[Lee59] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[Lib04] Leonid Libkin. *Elements of finite model theory*. Springer, 2004.

[Mac98] Ioan I Macarie. Space-efficient deterministic simulation of probabilistic automata. *SIAM Journal on Computing*, 27(2):448–465, 1998.

[Mas76] William Joseph Masek. *A fast algorithm for the string editing problem and decision graph complexity*. PhD thesis, Massachusetts Institute of Technology, 1976.

[MP00] C. Mereghetti and B. Palano. Threshold circuits for iterated matrix product and powering. *ITA*, 34(1):39–46, 2000.

[MT98] A. Maciel and D. Thérien. Threshold circuits of small majority-depth. *Inf. Comput.*, 146(1):55–83, 1998.

[Mul87] Ketan Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7(1):101–104, 1987.

[MV97] Meena Mahajan and V. Vinay. Determinant: Combinatorics, algorithms, and complexity. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.

[OS06] Sang-il Oum and Paul D. Seymour. Approximating clique-width and branch-width. *J. Comb. Theory, Ser. B*, 96(4):514–528, 2006.

[O1] Joseph O'Rourke. Advanced problem 6369. *Amer. Math. Monthly*, 88(10):769, 1981.

[Pot97] P. J. Potts. Efficient on-line computation of real functions using exact floating point. *Manuscript, Dept. of Computing, Imperial College, London*, 1997.

[Pot99] P. J. Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD thesis, Imperial College, University of London, 1999.

[PSZ00] Ramamohan Paturi, Michael E Saks, and Francis Zane. Exponential lower bounds for depth three boolean circuits. *Computational Complexity*, 9(1):1–15, 2000.

[Rag10] B V Raghavendra Rao. A study of width bounded arithmetic circuits and the complexity of matroid isomorphism. *[HBNI TH 17]*, 2010.

[Raz87] Alexander A Razborov. Lower bounds on the size of r depth circuits over a complete basis with logical addition. *Mathematical Notes*, 41(4):333–338, 1987.

[Raz90] AA Razborov. Lower bounds for monotone complexity of boolean functions. *American Mathematical Society Translations*, 147:75–84, 1990.

[RS84] Neil Robertson and Paul D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[SJ89] Alistair Sinclair and Mark Jerrum. Approximate counting, uniform generation and rapidly mixing markov chains. *Information and Computation*, 82(1):93–133, 1989.

[Smo87] Roman Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 77–82. ACM, 1987.

[Sta13] Richard P Stanley. *Algebraic Combinatorics*. Springer-Verlag New York, 2013.

[Tiw92] P. Tiwari. A problem that is easier to solve on the unit-cost algebraic ram. *J. Complexity*, 8(4):393–397, 1992.

[Tod91] S. Toda. Counting problems computationally equivalent to the determinant. Technical Report CSIM 91-07, Dept of Comp Sc & Information Mathematics, Univ of Electro-Communications, Chofu-shi, Tokyo, 1991.

[TS41] WT Tutte and CAB Smith. On unicursal paths in a network of degree 4. *The American Mathematical Monthly*, 48(4):233–237, 1941.

[Val79a] Leslie G Valiant. Completeness classes in algebra. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 249–261. ACM, 1979.

[Val79b] Leslie G Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

[Val92] Leslie G Valiant. Why is boolean complexity theory difficult. *Boolean Function Complexity*, 169:84–94, 1992.

[Ven92] H Venkateswaran. Circuit definitions of nondeterministic complexity classes. *SIAM Journal on Computing*, 21(4):655–670, 1992.

[Vin91] V. Vinay. Counting auxiliary pushdown automata. In *Structure in Complexity Theory*, pages 270–284, 1991.

[Vin96] V Vinay. Hierarchies of circuit classes that are closed under complement. In *Computational Complexity, 1996. Proceedings., Eleventh Annual IEEE Conference on*, pages 108–117. IEEE, 1996.

[Vio09] Emanuele Viola. On approximate majority and probabilistic time. *Computational Complexity*, 18(3):337–375, 2009.

[Vol99] Heribert Vollmer. *Introduction to circuit complexity - a uniform approach.* Texts in theoretical computer science. Springer, 1999.

[vzGG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013.

[Wag86] K. W. Wagner. The complexity of combinatorial problems with succinct input representation. *Acta Informatica*, 23:325–356, 1986.

[Wan94] Egon Wanke. k-nlc graphs and polynomial algorithms. *Discrete Applied Mathematics*, 54(2):251–266, 1994.

[Yao85] Andrew Chi-Chih Yao. Separating the polynomial-time hierarchy by oracles. In | *26th Annual Symposium on Foundations of Computer Science*, pages 1–10. IEEE, 1985.