The Dissertation Committee for Vladimir Traianov Trifonov certifies that this is the approved version of the following dissertation:

Techniques for Analyzing the Computational Power of Constant-Depth Circuits and Space-Bounded Computation

Committee:

Anna Gál, Supervisor

Eric Allender

Greg Plaxton

Vijaya Ramachandran

David Zuckerman

Techniques for Analyzing the Computational Power of Constant-Depth Circuits and Space-Bounded Computation

by

Vladimir Traianov Trifonov, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor Of Philosophy

The University of Texas at Austin

August 2006

Techniques for Analyzing the Computational Power of Constant-Depth Circuits and Space-Bounded Computation

Publication No. _____

Vladimir Traianov Trifonov, Ph.D.

The University of Texas at Austin, 2006

Supervisor: Anna Gál

The subject of computational complexity theory is to analyze the difficulty of solving computational problems within different models of computation. Proving lower bounds is easier in less powerful models and proving upper bounds is easier in the more powerful models. This dissertation studies techniques for analyzing the power of models of computation which are at the frontier of currently existing methods.

First, we study the power of certain classes of depth-three circuits. The power of such circuits is largely not understood and studying them under further restrictions has received a lot of attention. We prove exponential lower bounds on the size of certain depth-three circuits computing parity. Our approach is based on relating the lower bounds to correlation between parity and modular polynomials, and expressing the correlation with exponential sums. We show a new expression for the exponential sum which involves a certain affine space corresponding to the polynomial. This technique gives a unified treatment and generalization of bounds which include the results of Goldmann on linear polynomials and Cai, Green, and Thierauf on symmetric polynomials. We obtain bounds on the exponential sums for classes of polynomials of large degree and with a large number of terms, which previous techniques did not apply to.

Second, we study the space complexity of undirected st-connectivity. We prove an $O(\log n \log\log n)$ upper bound on the space complexity of undirected st-connectivity.

This improves the previous $O(\log^{4/3} n)$ bound due to Armoni et al. and is a big step towards the conjectured optimal $O(\log n)$ bound. Independently of our work and using different techniques recently Reingold proved the optimal bound. Interest in this question comes from the fact that undirected st-connectivity is complete for **SL**, a class of problems between **L** and **NL**. It has been noticed that questions in the space context tend to be easier to answer than the corresponding questions in the time context. Since understanding the power of non-determinism over determinism presents a major challenge to complexity theory, studying complexity classes between **L** and **NL**, which are the smallest natural classes capturing deterministic and non-deterministic space-bounded computation, is important.

Contents

Chapter 1

Introduction

The subject of computational complexity theory is to analyze the difficulty of solving computational problems. The complexity of a problem is analyzed within a fixed model of computation. Generally speaking a model of computation fixes the operations which are permitted when solving a particular problem. One such model is the Turing Machine (TM) model, which according to the Church-Turing thesis is also the most general one, in the sense that everything which is intuitively solvable by a discrete algorithm can also be solved by a Turing Machine. A Turing Machine has an infinite tape, divided into cells, a head which can read, write, and move on the tape, a set of internal states, and a program which specifies the behavior of the Turing Machine. Another model of computation is the boolean circuit model. In this model, the computation is performed by a circuit specified by a set of gates, which compute some boolean functions and which are connected in an acyclic manner by wires. Very often we are interested in a restricted version of a given model. For example, for circuits we might restrict the types of gates allowed in them or require that the circuits have a small depth. On the other hand, we can extend the power of a model, e.g. by permitting multiple tapes, or variations which allow for non-uniformity, non-determinism, or randomness.

To measure the complexity of a problem, we encode its instances as words over some finite alphabet Σ (Σ is usually taken to be $\{0,1\}$) and call the length of an encoding the *size of the instance*. For example, a graph could be encoded with its adjacency matrix, a number can be encoded in its binary representation, and so on. Once we fix the encoding, we can assume that the problem is a subset of Σ^*, the set of all words over the alphabet

Σ. A solution to the problem is an abstract device from the fixed model of computation, which for a given instance checks whether it belongs to the set of the problem. In the process of providing an answer, a solution to a problem uses different resources. For a TM we might be interested in the number of steps performed, the amount of space used, or the amount of randomness. For circuits we can ask what is their size. Ideally we want the resource to be used efficiently. The complexity of a solution to a problem is the function of the instance size which is the maximum over all instances of the given size of the amount of the resource used by the solution. The complexity of a problem is the complexity of the optimal solution to the problem.

Determining the complexity of a problem has two sides. On one hand, providing a solution sets an upper bound on the complexity. For example, the algorithm of [AKS02] sets a polynomial upper bound on the number of steps in which a deterministic TM decides whether a given number is prime. On the other hand, we might want to prove a lower bound on the complexity of a problem. Bridging the gap between a lower and an upper bound on the complexity of a problem is an ever occurring motif in many complexity-theoretic investigations. Naturally, proving lower bounds is easier in less powerful models of computation and obtaining upper bounds is easier in more powerful models.

A measure of how hard it is to prove lower bounds is the fact that at the moment lower bounds are known only for very restricted circuit models, e.g. circuits which have constant depth. The hardness of proving lower bounds is one of the reasons why restricted models of computation were introduced – the hope is that proving lower bounds in more restricted models of computation will provide techniques for proving lower bounds in more general models.

Another point of view on the central questions of computational complexity theory described above is about relating different complexity classes. Complexity classes classify problems according to their complexity in a fixed model of computation. For example, the class **P** is the set of problems decidable in polynomial time by a deterministic TM and **NP** is the set of problems acceptable by a non-deterministic TM in polynomial time. Similarly we have the classes **L** and **NL**, where instead of polynomial time we

have logarithmic space. In the boolean circuit model important examples are $\mathbf{AC}^k$ – polynomial size, depth $O(\log^k n)$, unbounded fan-in circuits with AND, OR, and NOT gates, and $\mathbf{NC}^k$ – polynomial size, depth $O(\log^k n)$, bounded fan-in circuits with AND, OR, and NOT gates. The important questions here are, given two complexity classes how they relate to each other – are they comparable, are they subsets of each other, is one a strict subset of another, etc. Such questions can be informally phrased as analyzing the relative computational power of different models of computation. For example, we have the following relations

$$\mathbf{AC}^0 \subset \mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{SL} \subseteq \mathbf{RL} \subseteq \mathbf{NL} \subseteq$$
$$\subseteq \mathbf{AC}^1 \subseteq \mathbf{NC}^2 \subseteq \mathbf{AC}^2 \subseteq \mathbf{NC}^3 \subseteq \mathbf{AC}^3 \subseteq \ldots \subseteq \mathbf{P} \subseteq \mathbf{NP},$$

where **SL** and **RL** are the classes of problems solvable by a correspondingly symmetric and randomized TM in logarithmic space. Notice that the only strict inclusion is at the bottom of this hierarchy of classes, although it is believed that all of the inclusions are strict, with the exception of the ones between **L** and **RL**. The second inclusion is due to Borodin [Bor77], the fourth is due to Aleliunas et al. [AKL$^+$79], and the fact that $\mathbf{NL} \subseteq \mathbf{AC}^1$ follows from the boolean matrix repeated squaring algorithm [Pap94]. The other inclusions follow from the definitions of the corresponding classes [Pap94]. At the time of writing of this dissertation it was proven by Reingold [Rei05] that actually $\mathbf{L} = \mathbf{SL}$.

An important concept is that of a complete problem. The important property of a problem P complete for a complexity class C under an appropriate reduction is that, if $P \in D$, where D is some complexity class, then $C \subseteq D$. Thus, in some sense, a complete problem captures the complexity of its class. Informally, if we can prove that P can be solved efficiently, then all problems in C have an efficient solution. Thus, checking if one complexity class is more powerful than another is reduced to providing a suitable upper bound on a particular problem. On the other hand, lower bounds can be used to show that two complexity classes are different. Thus, we can think of the central problems of computational complexity theory as analyzing the relative power of complexity classes.

As mentioned earlier, restricting a model of computation reduces its power, and thus makes proving lower bounds easier. Hence, separation in low level complexity classes

is usually easier. On the other hand, it has been noticed that changing the complexity measure also makes analogous questions easier. One example of this phenomenon is that the **NP** vs. **coNP** question seems to be out of reach of current techniques, whereas the analogous question for space has been resolved [Sze87, Imm88]. Another example, again related to time vs. space, is about the power of randomized computation. In the time context no general efficient derandomization of polynomial time randomized algorithms is known, although it is believed that such derandomization exists [IW97], whereas for logarithmic space randomized algorithms such derandomization is known [SZ95].

In this dissertation we study the computational power of constant-depth circuits and space-bounded computation. On one hand, we look at new techniques for proving lower bounds on the size of $\mathsf{MAJ} \circ \mathsf{MOD}_q \circ \mathsf{AND}$ circuits computing the MOD_v function (the notation is defined below) for relatively prime v and q. This question comes from natural extensions of considerations related to the inability of $\mathbf{AC}^0$ circuits to compute parity. On the other hand, we consider upper bounds on the space complexity of the undirected st-connectivity problem. This problem is **SL**-complete and questions about the complexity classes between **L** and **NL** are important and receive a lot of attention.

1.1 Constant-depth circuits

We denote vectors of scalars with bold font, i.e. x is a scalar and $\mathbf{x} = (x_1, \ldots, x_n)$ is a vector. A *boolean function* on n variables is a function from $\{0,1\}^n$ to $\{0,1\}$. Define the following boolean functions on n variables: $\mathsf{MOD}_q(\mathbf{x}) = 0$ iff q divides $\sum_i x_i$, $\mathsf{MAJ}_n(\mathbf{x}) = 1$ iff $\sum_i x_i > n/2$, $\mathsf{AND}_n(\mathbf{x}) = 1$ iff $\sum_i x_i = n$, and $\mathsf{OR}_n(\mathbf{x}) = 1$ iff $\sum_i x_i > 0$. For $x \in \{0,1\}$, let $\mathsf{NOT}(x) = 1 - x$. We call MOD_2 the *parity* function. Let $\mathbf{SYM}_n$ be the set of all symmetric boolean functions on n variables. Obviously $\mathsf{MAJ}_n, \mathsf{AND}_n, \mathsf{OR}_n \in \mathbf{SYM}_n$. We omit n from the subscripts of MAJ, AND, and OR, when its value is of no significance and can be deduced from the context.

A *boolean circuit* C is specified by some $n \in \mathbb{N}$ and a directed acyclic graph with exactly one vertex with out-degree zero. The vertices of C are called *gates* and its edges are called *wires*. The gates which have non-zero in-degree are labeled with boolean functions.

A gate with in-degree zero is called an *input gate* and is labeled with x_i, for some $i \in [n]$. The gate with out-degree zero is called the *output gate*. C defines a boolean function on n variables, which we also denote with C, in the intuitive manner. *The size of* C is the number of gates in it. A gate g of C is of *depth* d, if the length of a longest path from an input gate to g is d. *The depth of* C is the depth of the output gate. The in-degree/out-degree of a gate is called its *fan-in/fan-out*.

The following notation is standard. $\mathbf{AC}^0_k$ denotes the set of boolean functions computable by circuits of polynomial size, depth k, with AND, OR, and NOT gates; $\mathbf{AC}^0 = \bigcup_k \mathbf{AC}^0_k$. $\mathbf{ACC}^0_k[q]$ denotes the set of boolean functions computable by circuits of polynomial size, depth k, with AND, OR, NOT, and MOD_q gates; $\mathbf{ACC}^0[q] = \bigcup_k \mathbf{ACC}^0_k[q]$ and $\mathbf{ACC}^0 = \bigcup_q \mathbf{ACC}^0[q]$. $\mathbf{TC}^0_k$ denotes the set of boolean functions computable by circuits of depth k with AND, OR, NOT, and MAJ gates; $\mathbf{TC}^0 = \bigcup_k \mathbf{TC}^0_k$.

Let $\mathcal{C}_2$ and $\mathcal{C}_2$ be two classes of boolean circuits. We denote with $\mathcal{C}_1 \circ \mathcal{C}_2$ the class of circuits which have the outputs of circuits from $\mathcal{C}_2$ fed into the inputs of a circuit from $\mathcal{C}_1$. Thus, for example, $\mathsf{MAJ} \circ \mathsf{MOD}_q \circ \mathsf{AND}$ is the class of circuits having a MAJ gate at the output, MOD_q gates at depth two, and AND gates at depth one.

One of the questions that this dissertation investigates is how well $\mathsf{MAJ} \circ \mathsf{MOD}_q \circ \mathsf{AND}$ circuits compute the MOD_v function for relatively prime v and q. Despite the innocent look of the class of circuits we are considering, this question is quite hard. Proving lower bounds on the size of boolean circuits for specific functions is one of the central problems in computational complexity theory. The general question of proving lower bounds on the size of boolean circuits is notoriously difficult, because, for example, superpolynomial lower bounds on the size of boolean circuits computing a specific function from the complexity class **NP** would imply that $\mathbf{P} \neq \mathbf{NP}$. However, even much weaker (e.g. superlinear) lower bounds seem to remain out of reach of the current techniques. Therefore imposing various restrictions on the circuits and developing lower bound methods for restricted circuit models has received a lot of attention in the last few decades. The hope is to extend such techniques and develop new methods which are applicable towards stronger and stronger models.

1.1.1 Lower bounds for constant-depth circuits

The fundamental work of [Ajt83, FSS84, Yao85, Hås86] shows that $\mathsf{MOD}_2 \notin \mathbf{AC}^0$ and more specifically constant-depth circuits with AND, OR, and NOT gates need exponential size to compute MOD_2. These results also imply that $\mathsf{MAJ} \notin \mathbf{AC}^0$. A natural step after these results was to consider what happens, if we allow more powerful gates, e.g. MOD_q gates or MAJ gates, in our circuits. This led to the definition of the classes $\mathbf{ACC}^0$ and $\mathbf{TC}^0$ given above. A preliminary answer to this question was given by Smolensky [Smo87] who showed that $\mathsf{MOD}_v \notin \mathbf{ACC}^0[p^k]$, when v and p are different primes and k is fixed. The result of Smolensky implies that $\mathsf{MOD}_2 \notin \mathbf{AC}^0$. The power of $\mathbf{ACC}^0[q]$ circuits when q is not a prime power is much less understood. For example, although it is highly unlikely, $\mathbf{NP} \subseteq \mathbf{ACC}^0_3[6]$ has not been ruled out yet. A further motivation for considering lower bounds in constant-depth circuits is that they imply separation oracles for higher level complexity classes [FSS84, Yao85].

Depth-three circuits can be surprisingly powerful. Allender [All89] proved that $\mathbf{AC}^0$ is contained in the class of $\mathsf{MAJ} \circ \mathsf{MOD}_2 \circ \mathsf{AND}_{\log^{O(1)} n}$ circuits of quasipolynomial ($2^{\log^{O(1)} n}$) size. Yao [Yao90] proved that $\mathbf{ACC}^0$ is contained in the class of $\mathbf{TC}^0_2 \circ \mathsf{AND}_{\log^{O(1)} n}$ circuits of quasipolynomial size. It remains open, whether Allender's result can be extended to $\mathbf{ACC}^0$, i.e. it is not known, whether $\mathbf{ACC}^0$ is contained in the class of quasipolynomial size $\mathsf{MAJ} \circ \mathsf{MOD}_2 \circ \mathsf{AND}_{\log^{O(1)} n}$ circuits. This question was asked by Green in [Gre02] and is the main motivation for our investigation.

The result of Håstad and Goldmann [HG84] implies that a certain function in $\mathbf{ACC}^0$ requires exponential size $\mathsf{MAJ} \circ \mathsf{MOD}_2 \circ \mathsf{AND}_{O(\log n)}$ circuits. The result of Razborov and Wigderson [RW93] implies that a certain function in $\mathbf{ACC}^0$ requires $n^{\Omega(\log n)}$ size $\mathsf{MAJ} \circ \mathsf{MOD}_2 \circ \mathsf{AND}_n$ circuits. This result was recently extended to circuits with arbitrary $\mathbf{AC}^0$ circuits in place of the AND gates by Hansen and Miltersen [HM04]. [RW93] and [HM04] build on the results of [HG84]. However, the method of [HG84] applies to having arbitrary gates from $\mathbf{SYM}$ in the middle layer. Thus, in view of Yao's result [Yao90] and because $\mathsf{MAJ} \circ \mathbf{SYM} = \mathbf{TC}^0_2$, these results cannot be directly extended to obtaining exponential lower bounds on the size of $\mathsf{MAJ} \circ \mathsf{MOD}_2 \circ \mathsf{AND}_{\log^{O(1)} n}$ computing a function

in $\mathbf{ACC}^0$.

Other combinations of MAJ, MOD and AND gates in depth-3 circuits have also been considered (e.g. [BM89, Gro94, KP94, GT00]). For example, exponential lower bounds for functions in $\mathbf{ACC}^0$ were obtained in [Gro94, BM89] for $\mathsf{MAJ} \circ \mathsf{AND} \circ \mathsf{MOD}$ circuits, in [KP94] for $\mathsf{MAJ} \circ \mathsf{AND}_{O(1)} \circ \mathsf{MOD}$ circuits, and in [GT00] for $\mathsf{MOD}_q \circ \mathsf{AND}_{O(1)} \circ \mathsf{MOD}_2$ circuits.

1.1.2 Correlation and lower bounds

Depth-three circuits with MOD gates in the middle layer appear to be harder to analyze and, in particular, the power of $\mathsf{MAJ} \circ \mathsf{MOD} \circ \mathsf{AND}$ circuits remains less understood. Obtaining exponential lower bounds on the size of such circuits under various restrictions has received considerable attention [Gol95, CGT96, Gre99, AB01, Gre02, Bou05]. The starting point of all these results, including [HG84] which considers the more general $\mathsf{MAJ} \circ \mathbf{SYM} \circ \mathsf{AND}$ circuits, is the Discriminator Lemma of [HMP$^+$87]. We will need the following definition of correlation, which measures how close two boolean functions are.

Definition 1.1.1. *Let $g, f : \{0,1\}^n \to \{0,1\}$ be two boolean functions. The* correlation *$C(g,f)$ between g and f is*

$$C(g,f) = 2^{-n} \sum_{\mathbf{x} \in \{0,1\}^n} (-1)^{g(\mathbf{x})+f(\mathbf{x})}.$$

Notice that $C(g,f)$ is the probability that g and f are the same on a uniformly random input minus the probability that they are different. If the functions are the same, then the correlation is 1, and if they are complements of each other, then the correlation is -1. The correlation is 0 iff there are the same number of inputs on which the functions differ and on which they are the same. In this case, if we think of f as approximating g, then f does not perform better than a single unbiased coin toss.

Lemma 1.1.1 (Discriminator Lemma, [HMP$^+$87]). *Let $g : \{0,1\}^n \to \{0,1\}$ be a boolean function computed by a boolean circuit with a MAJ_m gate at the output which takes the results of the circuits $C_1, \ldots, C_m$ as inputs. Assume also that g is balanced, i.e. $|g^{-1}(0)| = |g^{-1}(1)|$. Then there exists $i \in [m]$ such that $|C(g, C_i)| \geq 1/m$.*

By this lemma, an upper bound on the absolute value of the correlation between a function g and an arbitrary circuit from a given class $\mathcal{C}$ implies a lower bound on the fan-in of the MAJ gate in $\mathsf{MAJ} \circ \mathcal{C}$ circuits computing g. In particular, proving that the absolute value of the correlation between MOD_v and functions from $\mathsf{MOD}_q \circ \mathsf{AND}_d$ is exponentially small, implies an exponential lower bound on the size of $\mathsf{MAJ} \circ \mathsf{MOD}_q \circ \mathsf{AND}_d$ circuits computing MOD_v.

Let $P(\mathbf{x})$ be a polynomial of degree d over $\mathbb{Z}_q$. Let f_P be a boolean function such that $f_P(\mathbf{x}) = 0$ iff $P(\mathbf{x}) = 0$. Notice that $f_P \in \mathsf{MOD}_q \circ \mathsf{AND}_d$. The correlation between a boolean function g and P is defined to be the correlation between g and f_P.

Smolensky's result [Smo87] implies that for different primes v and p, the absolute value of the correlation between MOD_v and polynomials of degree $\log^{O(1)} n$ over $\mathbb{Z}_q$, for $q = p^k$, is at most $n^{-1/2+o(1)}$. Unfortunately the technique of [Smo87] does not yield smaller bounds on the absolute value of the correlation, even for degree 2 or sparse polynomials, and cannot be applied, if q is not a prime power. By the results of [Ajt83, FSS84, Yao85, Hås86] we know that the absolute value of the correlation between MOD_2 and functions from $\mathbf{AC}^0$ is exponentially small. Cai, Green and Thierauf [CGT96] proved that the absolute value of the correlation between MOD_2 and symmetric polynomials of degree $\log^{O(1)} n$ over $\mathbb{Z}_q$, for q odd, is exponentially small. This was generalized by Green [Gre99] to an exponentially small upper bound on the absolute value of the correlation between MOD_v and symmetric polynomials of degree $\log^{O(1)} n$ over $\mathbb{Z}_q$, when v is a prime which does not divide q.

Extending these bounds to non-symmetric polynomials poses a significant challenge. The degree 1 case was solved by Goldmann [Gol95], who proved that the absolute value of the correlation between MOD_v and linear polynomials over $\mathbb{Z}_q$, when v has a prime factor which does not divide q, is at most $2^{-\Omega(n)}$. Alon and Beigel [AB01] showed that the absolute value of the correlation between MOD_2 and degree 2 polynomials over $\mathbb{Z}_q$, for odd q, is at most $2^{-(\log n)^{\varepsilon}}$ for some constant $\varepsilon < 1$, and for degree $O(1)$ polynomials the absolute value of the correlation is $o(1)$. Note that the bounds of [AB01] are weaker than the $n^{-1/2+o(1)}$ upper bounds implied by Smolensky's results [Smo87], but [Smo87]

is applicable only when q is a prime power. The first improvement over the bounds of [Smo87] and [AB01] for non-symmetric polynomials of degree more than 1 was achieved by Green [Gre02], who proved that the absolute value of the correlation between MOD_2 and degree 2 polynomials over $\mathbb{Z}_3$ is at most $2^{-\Omega(n)}$. The method used in [Gre02] relies very specifically on the degree being 2 and $q = 3$. Finally, Bourgain [Bou05] showed that for every $\varepsilon \in [0, 1)$ as long as the degree of the polynomial is at most $\varepsilon \log n/3q$, the correlation between MOD_v and the polynomial over $\mathbb{Z}_q$ is $2^{-\Omega(n^{1-\varepsilon})}$, if q is odd, and q and v are relatively prime. Bourgain's result was generalized by Green, Roy and Straubing [GRS05] to arbitrary (not necessarily odd) q.

1.1.3 Our results on depth-three circuits

One of the problems which this dissertation addresses is proving upper bounds on the absolute value of the correlation between MOD_2 and polynomials over $\mathbb{Z}_q$, q odd. We study this question because, as we saw in the previous section, it implies a lower bound on $\mathsf{MAJ} \circ \mathsf{MOD}_q \circ \mathsf{AND}$ circuits computing parity and, more generally, because obtaining techniques for dealing with modular gates is an important topic in computational complexity theory. We suggest a new approach to estimating the correlation, which is applicable to arbitrary odd q and improves previous bounds for several classes of polynomials.

We represent the correlation between parity and a polynomial over $\mathbb{Z}_q$ by an exponential sum. In addition, we use a linear transformation to evaluate the polynomial on a given input, i.e. first we apply a linear transformation to the given input and then the value of the polynomial is obtained as a function of the resulting vector. When bounding the exponential sum, this allows us to use linear algebraic properties over $\mathbb{Z}_2$ of a matrix obtained from the polynomial. In particular, we show a new expression for the exponential sum which involves a certain affine space over $\mathbb{Z}_2$ corresponding to the polynomial. Our technique gives a unified treatment and generalization of bounds which include the results of Goldmann [Gol95] on linear polynomials and Cai, Green, and Thierauf [CGT96] on symmetric polynomials. Furthermore, we obtain bounds on the exponential sums for classes of polynomials of large degree and with a large number of terms, which previous techniques did not apply to.

Previously, exponential sums have been used to represent correlation by Cai, Green and Thierauf [CGT96] and Green [Gre99] for symmetric polynomials, Green [Gre02] for degree 2 polynomials and $q = 3$, and Bourgain [Bou05] and Green, Roy, and Straubing [GRS05] for polynomials of degree almost $\log n/3q$. The benefit of working with exponential sums, instead of correlation directly, is that we can employ algebraic and analytic tools when bounding them. Despite the availability of such tools, bounding exponential sums is quite difficult. Studying similar sums is a major topic in mathematics [LN97] and we hope that employing techniques already developed for dealing with them will help to resolve the questions of interest to us.

We use $\equiv_v$ to denote equality modulo v. For two functions $g, f : \{0,1\}^n \to \mathbb{Z}$ and $v, q \in \mathbb{N}^+$ the exponential sum corresponding to g modulo v and f modulo q on boolean inputs is

$$E_{v,q}(g,f) = 2^{-n} \sum_{\mathbf{x}\in\{0,1\}^n} \omega_v^{g(\mathbf{x})} \omega_q^{f(\mathbf{x})}, \tag{1.1}$$

where ω_v and ω_q are correspondingly primitive v-th and q-th root of unity. These exponential sums are closely related to the correlation. For example, if $\chi(\mathbf{x}) = \sum_{i\in[n]} x_i$, P is a polynomial with integer coefficients, and f_P is the boolean function which is 0 iff $P(\mathbf{x}) \equiv_q 0$, then (Lemma 2.2.1)

$$C(\mathsf{MOD}_2, f_P) = \frac{2}{q} \sum_{t\in[q]} E_{2,q}(\chi, tP).$$

Thus a bound on $|E_{2,q}(\chi, tP)|$ for every $t \in [q]$ implies a bound on $C(\mathsf{MOD}_2, f_P)$. In a certain sense the converse of this is also true (Lemma 2.2.2). In general, we are interested in bounding $|E_{2,q}(\chi_{\mathbf{g}}, f)|$, where $\mathbf{g} \in \{0,1\}^n$ and $\chi_{\mathbf{g}}(\mathbf{x}) = \sum_{i\in[n]} g_i x_i$, i.e. we want to bound the exponential sums corresponding to parity over any subset of the input bits.

First, given a polynomial P with integer coefficients and using that q is odd we can construct another polynomial Q with integer coefficients of the same degree as P such that for every $\mathbf{x} \in \{0,1\}^n$,

$$P(\mathbf{x}) \equiv_q Q(\mathbf{y}),$$

where $\mathbf{y} \in \{-1,1\}^n$ and $y_i = (-1)^{x_i}$. Then, from Q we obtain a matrix $M \in \{0,1\}^{m\times n}$

such that

$$Q(\mathbf{y}) \equiv_q |\mathbf{z}|, \tag{1.2}$$

where $\mathbf{z} \in \{0,1\}^m$ and $\mathbf{x} \in \{0,1\}^n$ are such that $\mathbf{z} \equiv_2 M\mathbf{x}$, $x_i = 1$ iff $y_i = -1$, and $|\mathbf{z}|$ is the number of ones in $\mathbf{z}$. This situation can be reversed and given a matrix we can obtain a polynomial such that (1.2) holds. For every $\mathbf{g} \in \{0,1\}^n$ we prove (Lemma 2.4.1) that

$$E_{2,q}(\chi_{\mathbf{g}}, P) = 2^{-m} \sum_{\mathbf{y} \in K(M,\mathbf{g})} (\omega_q - \bar{\omega}_q)^{|\mathbf{y}|} (\omega_q - \bar{\omega}_q)^{m-|\mathbf{y}|}, \tag{1.3}$$

where $K(M,\mathbf{g})$ is the set of solutions to $M^T\mathbf{y} = \mathbf{g}$ over $\mathbb{Z}_2$. Since $K(M,\mathbf{g})$ is an affine space over $\mathbb{Z}_2$, this expression relates bounding the exponential sum to properties of this space, i.e. with linear algebraic properties over $\mathbb{Z}_2$ of M. In particular, using this expression we obtain bounds when $K(M,\mathbf{g})$ does not have too many elements, e.g. M is non-singular over $\mathbb{Z}_2$ (Theorem 2.7.1).

Generalize the notion of affine space in the following way. For matrices $M \in \{0,1\}^{m\times n}$, $M_1, \ldots, M_k$, $M_i \in \{0,1\}^{m_i \times n}$, and $\mathbf{g} \in \{0,1\}^n$ define

$$I(M) = \left\{\mathbf{z} \in \{0,1\}^n : \exists\, \mathbf{y} \in \{0,1\}^m \text{ s.t. } M^T\mathbf{y} \equiv_2 \mathbf{z}\right\},$$

$$J(M_1, \ldots, M_k; \mathbf{g}) = \left\{ (\mathbf{g}_1, \ldots, \mathbf{g}_k) \in I(M_1) \times \cdots \times I(M_k) : \sum_{i \in [k]} \mathbf{g}_i \equiv_2 \mathbf{g} \right\}.$$

$I(M)$ is the linear space over $\mathbb{Z}_2$ of vectors in the image of the linear transformation given by M^T. The affine space $K(M,\mathbf{g})$ is essentially $J(M_1, \ldots, M_m; \mathbf{g})$ where M_i is the i-th row of M. Define the weight of $J(M_1, \ldots, M_k; \mathbf{g})$ to be the smallest number of non-zero components of any of its elements. We obtain bounds on the exponential sum provided that $M_1, \ldots, M_k$ is a partition of the rows of M such that $J(M_1, \ldots, M_k; \mathbf{g})$ is small and its weight is large (Theorem 2.7.2). Furthermore, we obtain bounds independently of the weight of J, provided that for every $i \in [k]$ the polynomial Q_i corresponding to the matrix M_i is not constant modulo q on $\{-1,1\}$ inputs (Theorem 2.7.5). These bounds cover the linear polynomials case of Goldmann, but also include polynomials in which the degree is large. Simple examples of polynomials which these consideration can handle are when the matrix M is block-diagonal with many blocks $M_1, \ldots, M_k$ and either the weight of every vector in $I(M_i)$ is small, or every Q_i is not constant modulo q on $\{-1,1\}$ inputs.

Next, we show an even more general expression (Lemma 2.8.1) for the exponential sum (1.1) when $v = 2$ and $g = \chi_{\mathbf{g}}$, for some $\mathbf{g} \in \{0,1\}^n$. Given a function $f : \{0,1\}^n \to \mathbb{Z}$ this expression requires that we have $r \in \mathbb{N}^+$ and $A \in \mathbb{Z}^{m \times n}$, such that the vector $A\mathbf{x}$ modulo r determines the value of $f(\mathbf{x})$ modulo q. More precisely, there exists $h : \mathbb{Z}^m \to \mathbb{Z}$ such that for every $\mathbf{x} \in \{0,1\}^n$ and $\mathbf{y} \in \mathbb{Z}^m$, if $\mathbf{y} \equiv_r A\mathbf{x}$, then $f(\mathbf{x}) \equiv_q h(\mathbf{y})$. Notice that the considerations above give such a matrix with $r = v = 2$, if f is a polynomial. In particular (1.3) follows from this more general expression. On the other hand, we show that Zabek's Theorem (Theorem 2.8.3 and Lemma 2.8.4) gives an odd $r = poly(d)$ and a matrix with only one row, if f is a symmetric polynomial of degree at most d. This allows us to derive the result of Cai, Green, and Thierauf on symmetric polynomials (Theorem 2.8.5). We extend this result to functions of not too many generalized symmetric polynomials. A generalized symmetric polynomial T is specified by a symmetric polynomial S and a vector $\mathbf{a} \in \mathbb{Z}^n$ such that $T(\mathbf{x}) = S(a_1x_1, \cdots, a_nx_n)$; T is symmetric, if $\mathbf{a} = \mathbf{1}$. For a generalized symmetric polynomial of degree at most d a result of Voloch (Theorem 2.8.12) allows us to obtain an odd $r = poly(d)$ and a matrix with d rows. This gives us a bound analogous to the bound of Cai, Green, and Thierauf, when $f(\mathbf{x}) = g(T_1(\mathbf{x}), \ldots, T_k(\mathbf{x}))$, for any $k \leq n^\delta$, where δ depends on q, g is any function from $\mathbb{Z}_q$ to $\mathbb{Z}_q$, not necessarily a polynomial, and the T_i's are different generalized symmetric polynomials of degree at most d (Theorem 2.8.13).

Some of the results described above appear in [GT06].

1.2 Space-bounded computation

The other problem which this dissertation addresses is the space complexity of undirected st-connectivity. Before we present the problem and a history of algorithms for it, we define the relevant complexity classes.

A *deterministic Turing Machine* (DTM) is specified by 1) a constant number of infinite tapes, which are divided into cells containing 0, 1, and some special symbols, 2) a set of internal states, one of which is designated as a YES and one as a NO final state, 3) heads positioned over the tapes, and which can change the content of the cell they

are reading at the moment, and move to an adjacent cell, and 4) a program specifying the behavior of the DTM. The input to a DTM is given on a read-only tape. Given an input, a DTM performs some number of steps, according to its program and the content of the tapes, and stops in a final state determining its output. The tapes besides the input tape are called *worktapes*. If M is a DTM, we denote with $M(\mathbf{x})$ the boolean function computed by M on input $\mathbf{x}$. A TM is *non-deterministic* (NTM), if besides the input tape it has an additional one-way read-only tape, providing the non-deterministic choices of M. The output of a NTM depends both on the input and the non-deterministic choices. If M is a NTM we denote with $M(\mathbf{x}, \mathbf{y})$ the boolean function computed by M, where $\mathbf{x}$ is the input and $\mathbf{y}$ are the non-deterministic choices of M.

In the following, n denotes the size of an input.

Let P be a decision problem. P is *decided* by a DTM M, if for every $\mathbf{x}$, $M(\mathbf{x}) = 1$ iff $\mathbf{x} \in P$. P is *accepted* by a NTM M, if for every $\mathbf{x}$, if $\mathbf{x} \in P$, then there exists $\mathbf{y}$, such that $M(\mathbf{x}, \mathbf{y}) = 1$, and if $\mathbf{x} \notin P$, then for every $\mathbf{y}$, $M(\mathbf{x}, \mathbf{y}) = 0$. The *time* a TM takes on an input is the number of steps it needs to produce an answer, and the *space* is the largest number of cells used on the worktapes.

A decision problem P is in $\mathbf{TIME}(f(n))$, if there is a TM M, which decides P in time $O(f(n))$. A decision problem P is in $\mathbf{SPACE}(f(n))$, if there is a TM M, which decides P in space $O(f(n))$. Similarly we define $\mathbf{NTIME}(f(n))$ and $\mathbf{NSPACE}(f(n))$ for NTM. Let $\mathbf{P} = \bigcup_c \mathbf{TIME}(n^c)$ and $\mathbf{NP} = \bigcup_c \mathbf{NTIME}(n^c)$. Let $\mathbf{L} = \mathbf{SPACE}(\log n)$ and $\mathbf{NL} = \mathbf{NSPACE}(\log n)$.

A random access machine (RAM) is specified by 1) a constant number of registers, which contain integer numbers, 2) a program composed of instructions, which can perform arithmetic and branching using the content of the registers, and 3) a program counter. A parallel RAM (PRAM) is a RAM which has several processors executing its program, working in parallel and using the same registers. There are different versions of PRAM based on how we decide to resolve simultaneous accesses to a register. In EREW PRAM (exclusive read, exclusive write parallel RAM) concurrent access is forbidden, in CREW PRAM concurrent reads are allowed, and in CRCW PRAM both concurrent reads and

writes are allowed. We assume that concurrent writes are resolved arbitrarily.

The set of problems accepted by a RAM and a DTM in polynomial time is the same. The PRAM model can provide a significant speedup over a DTM necessary to solve a problem. Let $\mathbf{EREW}(t(n), p(n))$ be the set of problems decided by an EREW PRAM algorithm in time $O(t(n))$ and using $O(p(n))$ processors. Define analogously $\mathbf{CREW}(t(n), p(n))$ and $\mathbf{CRCW}(t(n), p(n))$ for CREW and CRCW PRAM. The *work* of a PRAM algorithm is defined as $t(n)p(n)$. Just like a TM is considered efficient, if it works in polynomial time, a PRAM algorithm is considered efficient, if it works in time $\log^{O(1)} n$ using a polynomial number of processors. The question of separating efficient sequential algorithms (represented by polynomial time DTMs) and efficient parallel algorithms (represented by polylogarithmic time PRAMs) is a big open question, similar to the **P** vs. **NP** question in this context. A PRAM algorithm is *optimal*, if its work is equal to the time necessary to solve the problem on a DTM.

Space, parallel time, and circuit depth are closely related. Informally we have the Parallel Computation Thesis, which says that parallel time and sequential space are polynomially related (see [KR90]). More precisely we have the following inclusions. Let $\mathbf{NC}(f(n), g(n))$ be the set of all problems solvable by circuits with AND, OR, and NOT gates, bounded fan-in, depth $O(f(n))$, and size $O(g(n))$. Define similarly $\mathbf{AC}(f(n), g(n))$ for unbounded fan-in circuits. Then,

$$\bigcup_c \mathbf{NC}(f, c^f) \subseteq \mathbf{SPACE}(f) \subseteq \mathbf{NSPACE}(f) \subseteq \bigcup_c \mathbf{AC}(f, c^f),$$

also

$$\mathbf{CRCW}(f, g) \subseteq \bigcup_c \mathbf{AC}(f, g^c) \text{ [SV84]},$$

and, if we consider uniform circuits, $\mathbf{AC}(f, g) \subseteq \mathbf{CRCW}(f, g)$. Finally

$$\mathbf{NC}(f, g) \subseteq \mathbf{EREW}(f, g) \text{ [HKP84]}.$$

Using that $\mathbf{AC}(f, g) \subseteq \bigcup_c \mathbf{NC}(f \log g, g^c)$, the above inclusions imply that

$$\bigcup_c \mathbf{CRCW}(f, n^c) \subseteq \mathbf{SPACE}(f \log n),$$

but leave open the problem of showing that particular problems from $\bigcup_c \mathbf{CRCW}(f, n^c)$ are in $\mathbf{SPACE}(o(f \log n))$, and of better sequential space-efficient general simulations of algorithms for problems in $\bigcup_c \mathbf{EREW}(f, n^c)$.

Of interest to us is also the class **SL**, defined as the set of problems solvable by a symmetric NTM in $O(\log n)$ space (in a symmetric NTM, for any two configurations c and d, if there is a computation leading from c to d, there is a computation which leads from d to c). This class is interesting to us because undirected st-connectivity is **SL**-complete [LP82]. As we mentioned earlier

$$\mathbf{L} \subseteq \mathbf{SL} \subseteq \mathbf{RL} \subseteq \mathbf{NL}.$$

Recently Reingold [Rei05] showed that $\mathbf{L} = \mathbf{SL}$.

1.2.1 Undirected st-connectivity

Let $G = (V, E)$ be a graph, $n = |V|$, $m = |E|$, and s and t be two vertices of G. The st-connectivity (STCONN) problem asks, whether there is a path from s to t. If G is undirected, we get the undirected st-connectivity (USTCONN) problem. The STCONN problem is one of the most basic graph problems. Applications of it range from image processing and VLSI design, to solving more complex graph problems such as ear decomposition, triconnectivity, and planarity. Furthermore, STCONN plays an important role in complexity theory because STCONN is **NL**-complete [Sav70] and USTCONN is **SL**-complete [LP82].

Linear space and time sequential algorithms for STCONN have been known for a long time [Tar72]. The advances in computer technology posed the problem of developing efficient parallel algorithms. Unfortunately, the existing sequential algorithms are implemented through depth- or breadth-first search and implementing those methods efficiently in parallel is difficult (see [Rei85, KR90]). Also the problem of constructing space-efficient algorithms gained theoretical interest, first because of the significance of the STCONN problem to defining complexity classes, and second because of the relation between sequential space and parallel time.

If we allow one-sided randomness, the result of Aleliunas et al. [AKL+79] shows that USTCONN can be solved in $O(\log n)$ space, i.e. it is in **RL**. It is widely believed that **RL** = **L** and so obtaining deterministic space-efficient algorithms for USTCONN became an important step in justifying this important conjecture. Also, it is hoped that techniques which show that USTCONN is in **L** will provide important clues about the proof of this conjecture. The starting point of deterministic space-efficient sequential algorithms is the $O(\log^2 n)$ space algorithm for STCONN of Savitch [Sav70]. For a long time this was the best result even for USTCONN. The space bound for undirected graphs was first improved by Nisan et al. [NSW92] to $O(\log^{3/2} n)$ and then to $O(\log^{4/3} n)$ by Armoni et al. [ATSWZ97]. Both of these results depend on the space-efficient construction of universal traversal sequences by Nisan [Nis90]. Very recently and independent of us, the space complexity of USTCONN was shown by Reingold [Rei05] to be $O(\log n)$. This result was obtained using a set of techniques different than ours.

Developing efficient parallel algorithms for USTCONN has a very rich history. The situation in this context is complicated further by the existence of different models of parallel computation. We will trace the history of USTCONN algorithms for the models from the PRAM family, defined above, which are generally accepted as having important theoretical role, although not being very realistic.

An early CREW PRAM algorithm for USTCONN running in $O(\log^2 n)$ time with $O(n^2/\log n)$ processors can be found in [HCS79]. In [SJ81] the work performed by this algorithm was improved slightly for sparse graphs allowing it to use $O(m + n \log n)$ processors. Further improvement of the work of the first algorithm was obtained in [CLC82] where the number of processors was reduced by a factor of $\log n$.

These results for CREW PRAM were followed by a sequence of results concerning CRCW PRAM. All of these algorithms achieved time $O(\log n)$, which is the best possible, including randomized, for the weaker CREW PRAM. The lower bound for CRCW PRAM algorithms is $\Omega(\log n/\log\log n)$, hence a small gap remains to be closed. The first of those algorithms appeared in [SV82] and uses $O(m + n)$ processors. A simpler algorithm with the same time and processor bounds was shown in [AS83]. A big move towards improving

the work of CRCW PRAM algorithms for USTCONN was made in [CV86] where an algorithm using $O((m+n)\alpha(m,n)/\log n)$ processors was developed, where α is a certain inverse of the Ackerman function and is extremely slow-growing.

The above results showed that $O(\log n)$ time was achievable on CRCW PRAM, but the question whether the same time bound holds on the weaker CREW PRAM model was still open, because simulating those algorithms on the weaker model increases their running time to $O(\log^2 n)$ [KR90]. For almost a decade there was no advance in the CREW PRAM algorithms for USTCONN. The breaking of the $O(\log^2 n)$ barrier in this model came with [JM91], which showed a deterministic algorithm solving USTCONN in $O(\log^{3/2} n)$ time with $O(m+n)$ processors.

Due to simulation results particular to those algorithms, the results of [NSW92] and [ATSWZ97] imply correspondingly an $O(\log^{3/2} n)$ and an $O(\log^{4/3} n)$ time EREW PRAM algorithm with polynomial number of processors. The result of [NSW92] concerning the time bound of EREW PRAM algorithms for USTCONN was superseded by [KNP92] and [JM92] which show algorithms running in time $O(\log^{3/2} n)$ with $O(m+n)$ processors. [JM92] actually solves the harder problem of finding the minimum spanning forest. An algorithm with better running time was presented in [CL95]. This algorithm works in time $O(\log n \log\log n)$ and uses $O(m+n)$ processors. Finally, [CHL01] showed that the optimal $O(\log n)$ time can be achieved on EREW PRAM with an algorithm using linear number of processors and which actually solves the minimum spanning forest problem.

Just like in the sequential-space context, having randomness is an advantage in developing efficient parallel algorithms for USTCONN. In the CRCW PRAM model a randomized algorithm running in optimal time and having optimal expected work appeared in [Gaz86]. Before the optimal time deterministic EREW PRAM algorithm of [CHL01], [KNP92] showed an optimal time randomized EREW PRAM algorithm with $O((m+n^{1+\varepsilon})/\log n)$ processors, for every $\varepsilon > 0$, based on a parallelization of the randomized algorithm of [AKL$^+$79]. [KNP92] also contains a randomized EREW PRAM algorithm for USTCONN running in $O(\log n \log\log n)$ time with $O(m+n)$ processors. The optimal time result of [KNP92] was followed by the randomized algorithms of [Rad94] and

[HZ96]. [Rad94] showed an algorithm which uses linear number of processors and [HZ96] achieved the optimal $O((m+n)/\log n)$ processors. The current best randomized EREW PRAM algorithm is the optimal time and work algorithm of [PR02], which furthermore solves the harder minimum spanning forest problem.

1.2.2 Our result: an $O(\log n \log\log n)$ space algorithm for USTCONN

We propose an $O(\log n \log\log n)$ space algorithm for USTCONN. The starting point of this algorithm is the **EREW**$(\log n \log\log n, m+n)$ algorithm of Chong and Lam [CL95]. We simulate the Chong–Lam algorithm with a sequential algorithm which requires linear space. We define a configuration as a mathematical structure, which captures the state of the sequential algorithm. We define also a sequence of configurations, such that every element of this sequence corresponds to the state of the sequential algorithm at certain points of its execution. The sequence of configurations trivially implies an $O(\log^2 n)$ space algorithm, which instead of storing all of its current state, recomputes parts of it when it needs them. This technique is standard for designing space-efficient algorithms. Finally, we modify the $O(\log^2 n)$ space algorithm into an algorithm which uses $O(\log n \log\log n)$ space.

The possibility of using parallel algorithms to define space-efficient sequential algorithms for USTCONN was suggested by Prof. Vijaya Ramachandran in 2000. She conjectured an $O(\log n \log\log n)$ space algorithm derived from [CL95] and an alternate simple $O(\log^{3/2} n)$ space algorithm derived from the algorithm of [JM91], by using the max-degree hooking scheme of [CL95]. She observed that the step needing derandomization in [NSW92] is not necessary in a tree-based hook and contract approach, because the trees automatically give rise to disjoint clusters of vertices. The max-degree hooking scheme employed by [CL95] gives the additional benefit that small trees have small neighborhoods. The main challenge was to implement the levels of recursion, so that they process small trees in $o(\log n)$ space.

In our algorithm the space of a level of recursion is between $\Omega(\log\log n)$ and $O(\log n)$, depending on the level. A key tool for our method are the exploration walks on trees defined in [Kou01b]. Exploration walks on trees are similar to the Euler tour

technique used by [TV85] in the parallel context. These walks play the role of the edge-list plugging technique and pointer jumping employed by the Chong–Lam algorithm, because they allow us to traverse trees efficiently.

This result appeared in [Tri05].

Chapter 2

Constant-depth circuits

Motivated by the ongoing quest for techniques for proving lower bounds for circuits which use MOD gates, in this chapter we describe our approach to estimating the correlation between parity and polynomials over $\mathbb{Z}_q$, q odd, using exponential sums. Before presenting our main results, in section 2.3 we illustrate the benefit, i.e. the availability of algebraic techniques, of working with exponential sums by presenting the results of Green and Bourgain. Although from the point of view of applying bounds on exponential sums to circuit lower bounds we want q to be odd, in section 2.5 we show that evaluating those sums for q even has implications for the weight distribution of binary affine codes, which in our opinion is one more indication of the importance of considering exponential sums. Finally, we show that representing the evaluation of a function using a linear transformation opens up a variety of possibilities for bounding the exponential sums. In particular, in section 2.4.1 we obtain a new expression for the exponential sum which involves an affine space over $\mathbb{Z}_2$ corresponding to the polynomial. Bounds on the new expression include the result of Goldmann, among many others shown in section 2.7. Furthermore in section 2.8, we are able to generalize the result of Cai, Green, and Thierauf on symmetric polynomials to functions of small number of generalized symmetric polynomials.

2.1 Notation

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$, the set of natural numbers, $\mathbb{N}^+ = \mathbb{N} - \{0\}$, the set of positive natural numbers, $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the ring of the integers, and $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ be the

field of rational, real, and complex numbers, correspondingly. Let $[0,1) = \{a \in \mathbb{R} : 0 \leq a < 1\}$. For $n \in \mathbb{N}^+$, let $[n] = \{1, \ldots, n\}$.

For a ring R, let $\mathcal{P}(R,n)$ be the set of polynomials over R on n independent variables and $\mathcal{P}(R,n,d)$ be the set of polynomials of degree at most d. Similarly define $\mathcal{P}^{sym}(R,n)$ and $\mathcal{P}^{sym}(R,n,d)$ for symmetric polynomials.

Let I be some finite index set and $a_i \in \mathbb{C}$ for every $i \in I$, then let

$$\widehat{\sum_{i \in I}} a_i = \frac{1}{|I|} \sum_{i \in I} a_i,$$

be the average value of the a_i's.

For $q \in \mathbb{N}^+$, let $\mathbb{Z}_q$ be the ring of integers modulo q and $\mathbb{Z}_q^*$ be the set of invertible elements of $\mathbb{Z}_q$, i.e. those which are relatively prime to q. We identify $\mathbb{Z}_q$ with $[q]$, when we want to think of it simply as a subset of $\mathbb{Z}$. For $x, y \in \mathbb{Z}$, $x \equiv_q y$ iff q divides $x - y$. Let

$$\varphi_q = 2\pi/q,$$

and

$$\omega_q = e^{2\pi \mathrm{i}/q} = \cos\varphi_q + \mathrm{i}\sin\varphi_q.$$

the q-th primitive root of unity and let $\bar{\omega}_q$ be the conjugate of ω_q, i.e. $\bar{\omega}_q = \omega_q^{-1}$. Let

$$\gamma_q = \max_{t \in [q-1]} \sin t\pi/q.$$

Notice that $\gamma_q \geq 1/\sqrt{2}$ and, if $q \geq 3$ is odd,

$$\begin{array}{l} \gamma_q = \sin(1 - 1/q)\pi/2 < 1, \\ \gamma_q = \max_{t \in [q]} |\sin t\varphi_q|, \text{ and } \max_{t \in [q-1]} |\cos t\varphi_q| < \gamma_q. \end{array} \tag{2.1}$$

We have that [LN97]

$$\widehat{\sum_{s \in [q]}} \omega_q^{sx} = \begin{cases} 1, & x \equiv_q 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

For $x \in \mathbb{Z}$ define

$$\delta_q(x) = \begin{cases} 1, & x \equiv_q 0, \\ -1, & \text{otherwise.} \end{cases}$$

It is immediate from (2.2) that

$$\delta_q(x) = 2\widehat{\sum_{s\in[q]}} \omega_q^{sx} - 1. \tag{2.3}$$

We denote vectors with bold font. If $f(y_1, \ldots, y_k)$ is a function from scalars to scalars, then for vectors $\mathbf{x}_1, \ldots, \mathbf{x}_k$, $\mathbf{x}_i = (x_{i1}, \ldots, x_{in})$,

$$f(\mathbf{x}_1, \ldots, \mathbf{x}_k) = (f(x_{11}, x_{21}, \ldots, x_{k1}), \ldots, f(x_{1n}, \ldots, x_{kn})).$$

Similarly if $R(y_1, \ldots, y_k)$ is a relation between scalars, then for vectors $\mathbf{x}_1, \ldots, \mathbf{x}_k$

$$R(\mathbf{x}_1, \ldots, \mathbf{x}_k) \text{ iff } R(x_{1i}, \ldots, x_{ki}) \text{ for every } i \in [n].$$

In particular

$$a^{\mathbf{x}} = (a^{x_1}, \ldots, a^{x_n}),$$
$$\mathbf{x} \cdot \mathbf{y} = (x_1 y_1, \ldots, x_n y_n),$$
$$\mathbf{x} \equiv_q \mathbf{y} \text{ iff } x_i \equiv_q y_i \text{ for every } i \in [n],$$
$$\mathbf{x} \leq \mathbf{y} \text{ iff } x_i \leq y_i \text{ for every } i \in [n].$$

For a function $f : \{-1, 1\}^n \to \mathbb{Z}$, let $\bar{f} : \mathbb{Z}^n \to \mathbb{Z}$ be

$$\bar{f}(\mathbf{x}) = f((-1)^{\mathbf{x}}).$$

When vectors are used in multiplication they are assumed to be in a column form and $\mathbf{x}^T$ is the row vector corresponding to a column vector $\mathbf{x}$. Similarly M^T is the transpose of a matrix M. For two vectors $\mathbf{x}$ and $\mathbf{y}$, $\mathbf{x}^T\mathbf{y}$ is the usual inner product of the two vectors, that is

$$\mathbf{x}^T\mathbf{y} = \sum_{i\in[n]} x_i y_i.$$

For a matrix M and a vector $\mathbf{x}$, $M\mathbf{x}$ is the product of M and $\mathbf{x}$. We denote with $\mathbf{0}_{m\times n}$ and $\mathbf{1}_{m\times n}$ the all 0's and the all 1's matrix of dimension $m \times n$ (we omit n if it is equal to 1). For $M \in \mathbb{Z}^{m\times n}$ and q prime, $\mathrm{rk}_q(M)$ denotes the rank of M over $\mathbb{Z}_q$.

We will often use the following simple consequence of (2.2). For $\mathbf{x} \in \mathbb{Z}^n$,

$$\widehat{\sum_{\mathbf{s}\in[q]^n}} \omega_q^{\mathbf{s}^T\mathbf{x}} = \begin{cases} 1, & \mathbf{x} \equiv_q \mathbf{0}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

Let $v, q \in \mathbb{N}^+$ and $g, f : \mathbb{Z}^n \to \mathbb{Z}$. Define the correlation between g modulo v and f modulo q on binary inputs to be

$$C_{v,q}(g, f) = \widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} \delta_v(g(\mathbf{x}))\delta_q(f(\mathbf{x})).$$

Define the exponential sums

$$E_{v,q}(g, f) = \widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} \omega_v^{g(\mathbf{x})}\omega_q^{f(\mathbf{x})},$$
$$D_{v,q}(g, f) = \widehat{\sum_{\mathbf{x}\in[v]^n}} \omega_v^{g(\mathbf{x})}\omega_q^{f(\mathbf{x})}.$$

Abusing notation somewhat, for $\mathbf{g} \in \mathbb{Z}^n$ we denote with $\mathbf{g}$ also the function $\chi_{\mathbf{g}}(\mathbf{x}) = \mathbf{g}^T\mathbf{x}$.

2.2 Correlation and exponential sums

We wish to estimate how well $\mathsf{MOD}_q \circ \mathsf{AND}$ circuits approximate parity. Let $f : \{0,1\}^n \to \{0,1\}$ be the function computed by a $\mathsf{MOD}_q \circ \mathsf{AND}$ circuit, and let P_f be the polynomial corresponding to the circuit. Then $(-1)^{f(\mathbf{x})} = \delta_q(P_f(\mathbf{x}))$ for every $\mathbf{x} \in \{0,1\}^n$, and with our notation, the correlation between parity and f is equal to $C_{2,q}(\mathbf{1}, P_f)$. In general, our methods apply to estimating the correlation for parity over arbitrary subset of the input bits. Thus, we are interested in estimating $C_{2,q}(\mathbf{g}, P)$ for a vector $\mathbf{g} \in \{0,1\}^n$ and a polynomial $P \in \mathcal{P}(\mathbb{Z}, n)$.

Following [CGT96, Gre99, Gre02], we use exponential sums to represent the correlation $C_{2,q}(\mathbf{1}, f)$ between parity and an integer valued function on boolean inputs. This representation also applies to parity taken over arbitrary subsets of the input variables. As we will see later, the benefit in working with exponential sums, instead of correlation directly, is that we can employ analytic and algebraic techniques in bounding them.

Lemma 2.2.1. *Let $q \in \mathbb{N}^+$, $f : \{0,1\}^n \to \mathbb{Z}$, and $\mathbf{g} \in \{0,1\}^n$. Then*

$$C_{2,q}(\mathbf{g}, f) = -\nu + 2\widehat{\sum_{t\in[q]}} E_{2,q}(\mathbf{g}, tf),$$

where ν is 0, if $\mathbf{g} \neq \mathbf{0}$, and 1, otherwise.

Proof. The equality follows from (2.3), i.e. that $\delta_2(z) = (-1)^z$ and $\delta_q(z) = 2\widehat{\sum}_{t\in[q]} \omega_q^{tz} - 1$ for $z \in \mathbb{Z}$, and since by (2.4)

$$\widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} (-1)^{\mathbf{g}^T\mathbf{x}} = \begin{cases} 1, & \mathbf{g} \equiv_2 \mathbf{0}, \\ 0, & \text{otherwise.} \end{cases}$$

□

Notice that, if $\mathbf{g} \neq \mathbf{0}$, by the triangle inequality applied to the expression in Lemma 2.2.1, there exists $t \in [q]$ such that $|C_{2,q}(\mathbf{g}, f)| \leq 2|E_{2,q}(\mathbf{g}, tf)|$. Hence, if $\mathbf{g} \neq \mathbf{0}$ and we can obtain an exponentially small bound on $|E_{2,q}(\mathbf{g}, tf)|$ for every $t \in [q]$, then we have an exponentially small bound on $|C_{2,q}(\mathbf{g}, f)|$. Let us see that the converse of this is also true in the following sense: if $|C_{2,q}(\mathbf{g}, f + c)|$ is exponentially small for every $c \in [q]$, then $|E_{2,q}(\mathbf{g}, tf)|$ is exponentially small as well for every $t \in [q]$. To prove this, we use that $E_{2,q}(\mathbf{g}, t(f + c)) = \omega_q^{tc} E_{2,q}(\mathbf{g}, tf)$ and the following lemma.

Lemma 2.2.2. *Let $q \in \mathbb{N}^+$, $a_1, a_1, \ldots, a_q \in \mathbb{C}$ and $b_j = \sum_{t\in[q]} a_t \omega_q^{tj}$. Then*

- *if $\varepsilon \in \mathbb{R}$ is such that $|a_t| \leq \varepsilon$ for every $t \in [q]$, then $|b_j| \leq q\varepsilon$ for every $j \in [q]$,*
- *if $\varepsilon \in \mathbb{R}$ is such that $|b_j| \leq \varepsilon$ for every $j \in [q]$, then $|a_t| \leq \varepsilon$ for every $t \in [q]$.*

Proof. The first point is immediate by the triangle inequality. To see the second point notice that using (2.2) we get $a_i = \widehat{\sum}_{t\in[q]} b_t \bar{\omega}_q^{ti}$. Now apply triangle inequality again. □

2.3 Previous results

As we mentioned in the introduction, the history of bounding the correlation $|C_{v,q}(\mathbf{1}, P)|$ for $P \in \mathcal{P}(\mathbb{Z}, n, d)$ is quite rich and we have the results of [Smo87, CGT96, Gre99, Gol95, AB01, Gre02, Bou05, GRS05] which give different bounds under a variety of restrictions

on v, q, and P. In this section we give the proofs of two results on bounding the correlation when $v = 2$ using exponential sums. The first is the result of Green [Gre02] for $q = 3$ and degree 2 and the second is the result of Bourgain [Bou05] for $q \geq 3$ odd and degree up to almost $\log n/(3q)$. We do not use these results and their proofs are given here just as an illustration of the power of the method of bounding the correlation through exponential sums.

Recall that for a function $f : \{0,1\}^n \to \mathbb{Z}$, $\bar{f} : \mathbb{Z}^n \to \mathbb{Z}$ is such that $\bar{f}(\mathbf{x}) = f((-1)^{\mathbf{x}})$.

2.3.1 Green's result

Lemma 2.3.1 ([Gre02]). *Let $a, b, c \in \mathbb{Z}$, such that $c \not\equiv_3 0$. Let $Q(x, y) = ax + by + cxy$. Then*

$$E_{2,3}(\mathbf{1}, \bar{Q}) = \frac{1}{4}(\omega_3^c - \bar{\omega}_3^c)\left(\omega_3^{(a-b)^2} + \bar{\omega}_3^{(a+b)^2}\right).$$

Proof. Expanding the left hand side of the expression in the lemma and cancelling $1/4$ we obtain

$$\omega_3^{a+b+c} + \omega_3^{-a-b+c} - \omega_3^{a-b-c} - \omega_3^{-a+b-c}.$$

Let $e = a + b$ and $f = a - b$. Our goal is to prove that

$$\omega_3^{e+c} + \omega_3^{-e+c} - \omega_3^{f-c} - \omega_3^{-f-c} = \omega_3^{-e^2+c} - \omega_3^{-e^2-c} + \omega_3^{f^2+c} - \omega_3^{f^2-c}.$$

Since for every $z \in \mathbb{Z}$, by inspection,

$$\omega_3^z + \bar{\omega}_3^z = \omega_3^{z^2} + \bar{\omega}_3^{z^2},$$

the left-hand side above could be rewritten as

$$\omega_3^{e^2+c} + \omega_3^{-e^2+c} - \omega_3^{f^2-c} - \omega_3^{-f^2-c}.$$

After cancellation our goal becomes

$$\omega_3^{e^2+c} - \omega_3^{-f^2-c} = -\omega_3^{-e^2-c} + \omega_3^{f^2+c}$$

and after rearrangement

$$\omega_3^{e^2+c} + \omega_3^{-e^2-c} = \omega_3^{f^2+c} + \omega_3^{-f^2-c}$$

which holds since for every $z \in \mathbb{Z}$, by (2.2)

$$\omega_3^{z^2+c} + \omega_3^{-z^2-c} = -1.$$

□

Theorem 2.3.2 (Green, [Gre02]). *For every $Q \in \mathcal{P}(\mathbb{Z}, n, 2)$,*

$$|E_{2,3}(\mathbf{1}, \bar{Q})| \leq \gamma_3^{\lceil n/2 \rceil}.$$

Proof. First, if modulo 3, Q is linear, then the theorem follows from Goldmann's result [Gol95] (see also the comment after Theorem 2.7.2).

Assume now that modulo 3, Q has a degree 2 term. Then

$$Q(y_1, \ldots, y_{n-2}, z_1, z_2) = R(\mathbf{y}) + A(\mathbf{y})z_1 + B(\mathbf{y})z_2 + cz_1z_2,$$

where R has degree at most 2, A and B are linear, and $c \not\equiv_3 0$. Then

$$\widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} (-1)^{\mathbf{1}^T\mathbf{x}} \omega_3^{\bar{Q}(\mathbf{x})} = \widehat{\sum_{\mathbf{x}\in\{0,1\}^{n-2}}} (-1)^{\mathbf{1}^T x} \omega_3^{\bar{R}(\mathbf{x})} E_{2,3}(\mathbf{1}, \bar{S}_{\mathbf{x}}),$$

where for every $\mathbf{x} \in \{0,1\}^{n-2}$,

$$S_{\mathbf{x}}(z_1, z_2) = \bar{A}(\mathbf{x})z_1 + \bar{B}(\mathbf{x})z_2 + cz_1z_2.$$

Since $c \not\equiv_3 0$, by Lemma 2.3.1 we know that

$$E_{2,3}(\mathbf{1}, \bar{S}_{\mathbf{x}}) = \frac{1}{4}(\omega_3^c - \bar{\omega}_3^c)\left(\omega_3^{(\bar{A}(\mathbf{x})-\bar{B}(\mathbf{x}))^2} + \bar{\omega}_3^{(\bar{A}(\mathbf{x})+\bar{B}(\mathbf{x}))^2}\right).$$

Notice that, since A and B are linear on $n-2$ variables, $(A-B)^2$ and $(A+B)^2$ are of degree at most 2 on $n-2$ variables. Thus, there exists T of degree at most 2 on $n-2$ variables (T is either $(A-B)^2$ or $-(A+B)^2$) such that if $U = R + T$, then

$$|E_{2,3}(\mathbf{1}, \bar{Q})| \leq \frac{|\omega_3^c - \bar{\omega}_3^c|}{2} |E_{2,3}(\mathbf{1}, \bar{U})|.$$

Since $c \not\equiv_3 0$, $|\omega_3^c - \bar{\omega}_3^c|/2 = \gamma_3$.

Since U is of degree at most 2 on $n-2$ variables, we managed to extract γ_3 out and reduce the number of variables by 2 without decreasing the exponential sum and keeping

the degree at most 2. We can now extract γ_3 again, if modulo 3 the new polynomial has a degree 2 term and so on. Let's say that we did the extraction of γ_3 k times. Since on every step we reduce the number of variables by 2, $k \leq \lfloor n/2 \rfloor$. At the end we are left with a linear polynomial in $n-2k$ variables whose exponential sum, as we mentioned at the beginning is at most γ_3^{n-2k}. Hence

$$|E_{2,3}(\mathbf{1}, \bar{Q})| \leq \gamma_3^k \gamma_3^{n-2k} = \gamma_3^{n-k} \leq \gamma_3^{\lceil n/2 \rceil}.$$

□

2.3.2 Bourgain's result

Theorem 2.3.3 (Bourgain, [Bou05]). *Let $q \geq 3$ be odd and $d \in \mathbb{N}^+$. Then for every $Q \in \mathcal{P}(\mathbb{Z}, n, d)$,*

$$|E_{2,q}(\mathbf{1}, \bar{Q})| \leq d\gamma_q^{8^{-qd}n}.$$

Notice that from the theorem follows immediately that for every $\varepsilon \in [0,1)$, if Q has degree at most $(\varepsilon \log n)/(3q)$, then

$$|E_{2,q}(\mathbf{1}, \bar{Q})| \leq 2^{-\Omega(n^{1-\varepsilon})},$$

where the constant in Ω depends only on q.

Proof. By induction on $d > 0$ we will prove that, if Q has degree at most d, then

$$|E_{2,q}(\mathbf{1}, \bar{Q})| \leq \sum_{i \in [d]} \gamma_q^{8^{-qi}n}.$$

The theorem follows from this immediately.

If $d = 1$, the statement follows from Goldmann's result [Gol95] (see also the comment after Theorem 2.7.2), so we can assume that $d > 1$.

We have that

$$\begin{aligned} E_{2,q}(\mathbf{1}, \bar{Q})^q &= \widehat{\sum_{\mathbf{x}_1, \ldots, \mathbf{x}_q \in \{0,1\}^n}} (-1)^{\sum_{j \in [q]} \mathbf{1}^T \mathbf{x}_j} \omega_q^{\sum_{j \in [q]} \bar{Q}(\mathbf{x}_j)} \\ &= \widehat{\sum_{\mathbf{y} \in \{0,1\}^n}} \widehat{\sum_{\mathbf{x}_1, \ldots, \mathbf{x}_q \in \{0,1\}^n}} (-1)^{\sum_{j \in [q]} \mathbf{1}^T (\mathbf{x}_j + \mathbf{y})} \omega_q^{\sum_{j \in [q]} \bar{Q}(\mathbf{x}_j + \mathbf{y})}, \end{aligned}$$

where we use that the inner summation is the same for every $\mathbf{y} \in \{0,1\}^n$. By triangle inequality and using that q is odd

$$|E_{2,q}(\mathbf{1}, \bar{Q})|^q \leq \widehat{\sum_{\mathbf{x}_1,\ldots,\mathbf{x}_q \in \{0,1\}^n}} \left| \widehat{\sum_{\mathbf{y} \in \{0,1\}^n}} (-1)^{\mathbf{1}^T \mathbf{y}} \omega_q^{\sum_{j \in [q]} \bar{Q}(\mathbf{x}_j + \mathbf{y})} \right|.$$

For a given $\mathbf{x}_1, \ldots, \mathbf{x}_q \in \{0,1\}^n$, $\mathbf{x}_j = (x_{j1}, \ldots, x_{jn})$, let

$$A = \{i \in [n] : \forall j \in [q]\, x_{ji} = 0\}.$$

Then

$$\begin{aligned} |E_{2,q}(\mathbf{1}, \bar{Q})|^q &\leq \widehat{\sum_{\mathbf{x}_1,\ldots,\mathbf{x}_q \in \{0,1\}^n}} \widehat{\sum_{y_i \in \{0,1\}: i \notin A}} \left| \widehat{\sum_{y_i \in \{0,1\}: i \in A}} (-1)^{\sum_{i \in A} y_i} \omega_q^{\sum_{j \in [q]} \bar{Q}(\mathbf{x}_j + \mathbf{y})} \right| \\ &= \widehat{\sum_{\mathbf{x}_1,\ldots,\mathbf{x}_n \in \{0,1\}^n}} \widehat{\sum_{y_i \in \{0,1\}: i \notin A}} |E_{2,q}(\mathbf{1}_{|A|}, \bar{R}_A)|, \end{aligned}$$

where for given A and y_i, $i \notin A$, R_A is a polynomial on $|A|$ variables, indexed by the elements of A, and $R_A(\mathbf{u}) = \sum_{j \in [q]} Q((-1)^{\mathbf{x}_j} \cdot \mathbf{z})$, where

$$z_i = \begin{cases} u_i, & i \in A, \\ (-1)^{y_i}, & \text{otherwise.} \end{cases}$$

Let us see now that R_A has degree at most $d-1$. Because R_A is a polynomial on u_i, $i \in A$, a typical term of R_A of degree k has a coefficient of the form

$$c \sum_{j \in [q]} \prod_{i \in S} (-1)^{x_{ji} + y_i} \prod_{i \in T} (-1)^{x_{ji}},$$

for some $c \in \mathbb{Z}$, $S \subseteq [n] - A$, $T \subseteq A$, and $|S| + |T| = k$, where $c \prod_{i \in S \cup T} z_i$ is a term of $Q(\mathbf{z})$. Since Q has degree at most d, in a term of R_A of degree d, we must have $S = \emptyset$ and $|T| = d$. Since for every $i \in A$ and $j \in [q]$, we have that $x_{ji} = 0$, such term must be 0 modulo q. Therefore R_A has degree at most $d-1$.

For every $i \in [n]$, we have that

$$\Pr_{\mathbf{x}_1,\ldots,\mathbf{x}_q \in \{0,1\}^n} [\forall j \in [q]\, x_{ji} = 0] = 2^{-q}.$$

Therefore

$$\widehat{\sum_{\mathbf{x}_1,\ldots,\mathbf{x}_q \in \{0,1\}^n}} |A| = n 2^{-q},$$

and by Chernoff bound

$$\Pr_{\mathbf{x}_1,\ldots,\mathbf{x}_q\in\{0,1\}^n}[|A| < (1/2)n2^{-q}] \leq 2^{-(1/8)n2^{-q}}.$$

Hence there exists a polynomial R on at least $(1/2)n2^{-q}$ variables and of degree $d-1$ such that

$$|E_{2,q}(\mathbf{1},\bar{Q})| \leq \left(2^{-n2^{-q-3}} + |E_{2,q}(\mathbf{1},\bar{R})|\right)^{1/q},$$

and therefore the inductive hypothesis implies that

$$\begin{aligned}|E_{2,q}(\mathbf{1},\bar{Q})| &\leq \left(2^{-n2^{-q-3}} + \sum_{i\in[d-1]} \gamma_q^{8^{-qi}n2^{-q-1}}\right)^{1/q} \\ &\leq \gamma_q^{n2^{-q-3}/q} + \sum_{i\in[d-1]} \gamma_q^{8^{-qi}n2^{-q-3}/q} \leq \sum_{i\in[d]} \gamma_q^{8^{-qi}n},\end{aligned}$$

using that $q \geq 3$, $\gamma_q \geq 1/\sqrt{2}$, and that for $a, b \geq 0$, $(a+b)^{1/q} \leq a^{1/q} + b^{1/q}$. □

2.4 Polynomial evaluation vs. linear transformations

With this section we begin the exposition of our approach to bounding exponential sums using a linear transformation for the evaluation of the given polynomial. Let q be odd and $P \in \mathcal{P}(\mathbb{Z}, n)$. First, let us construct $Q \in \mathcal{P}(\mathbb{Z}, n)$ with the same degree as P such that for $\mathbf{x} \in \{0,1\}^n$,

$$P(\mathbf{x}) \equiv_q \bar{Q}(\mathbf{x}) = Q((-1)^{\mathbf{x}}).$$

Since q is odd, there exists a unique integer $\rho \in [q-1]$ such that $2\rho \equiv_q 1$. For $z \in \mathbb{Z}$, let

$$l(z) = \rho(1-z).$$

Notice that for $\mathbf{x} \in \{0,1\}^n$

$$\mathbf{x} \equiv_q l((-1)^{\mathbf{x}}). \tag{2.5}$$

Define $Q(\mathbf{y}) = P(l(\mathbf{y}))$. Since l considered as a univariate polynomial is linear with integer coefficients, Q is a polynomial with integer coefficients of the same degree as P. Also, by (2.5), for every $\mathbf{x} \in \{0,1\}^n$ we have $P(\mathbf{x}) \equiv_q \bar{Q}(\mathbf{x})$. Thus

$$C_{2,q}(\mathbf{g}, P) = C_{2,q}(\mathbf{g}, \bar{Q}).$$

Our next goal is to express the evaluation of $\bar{Q}$ using a linear transformation over $\mathbb{Z}_2$. Since Q is multilinear with integer coefficients, we can write it as $\sum_{I\subseteq[n]} c_I y_I$, where $c_I \in \mathbb{Z}$ and $y_I = \prod_{i\in I} y_i$ $(y_\emptyset = 1)$. Let $M \in \{0,1\}^{m\times n}$ be the matrix whose rows are the incidence vectors of the subsets $I \subseteq [n]$, each repeated $(c_I \bmod q)$ times. Notice that the degree of Q, and therefore the degree of P, is at most d iff M has at most d 1's per row. We have that

$$\bar{Q}(\mathbf{x}) \equiv_q \mathbf{1}^T(-1)^{M\mathbf{x}}. \tag{2.6}$$

Conversely given $M \in \{0,1\}^{m\times n}$ we can obtain a polynomial Q such that (2.6) holds – the coefficient in front of a term of Q depends on how many times the corresponding row of M is repeated. Thus to some extent we can identify the evaluation of polynomials on $\{-1,1\}$ inputs to certain linear transformations in $\mathbb{Z}_2$. The advantage we gain from the linear transformation point of view is that we can look at linear algebraic properties of the corresponding matrix. This point will be seen in the next section.

Given a polynomial Q and matrix M we say that M is a matrix of Q modulo q (or that Q is the polynomial of M modulo q), if (2.6) holds. A polynomial has many matrices, but modulo q a matrix has exactly one polynomial. Call the matrix obtained from Q in the way described above the *defining matrix* of Q modulo q, i.e. in the defining matrix of Q the row corresponding to the term y_I is repeated $(c_I \bmod q)$ times.

2.4.1 Main lemma

Let $P \in \mathcal{P}(\mathbb{Z}, n)$. As discussed in section 2.2, for $\mathbf{g} \neq \mathbf{0}$, $|C_{2,q}(\mathbf{g}, P)|$ is exponentially small, if $|E_{2,q}(\mathbf{g}, tP)|$ is exponentially small for every $t \in [q]$. Thus, we will be concerned with giving bounds on $|E_{2,q}(\mathbf{g}, tP)|$.

Definition 2.4.1. *For $M \in \{0,1\}^{m\times n}$ and $\mathbf{g} \in \{0,1\}^n$, define*

$$I(M) = \left\{\mathbf{z} \in \{0,1\}^n : \exists\, \mathbf{y} \in \{0,1\}^m \text{ s.t. } M^T\mathbf{y} \equiv_2 \mathbf{z}\right\},$$
$$K(M,\mathbf{g}) = \left\{\mathbf{y} \in \{0,1\}^m : M^T\mathbf{y} \equiv_2 \mathbf{g}\right\}.$$

As a subset of $\mathbb{Z}_2^n$, $I(M)$ is the linear space which is the image of the linear transformation specified by M^T. $K(M,\mathbf{g})$ is the affine space of $\mathbb{Z}_2^m$ which is the set

of solutions of $M^T\mathbf{y} = \mathbf{g}$ over $\mathbb{Z}_2$. In particular, $K(M, \mathbf{0})$ is the kernel of the linear transformation specified by M^T.

The following lemma is our main technical tool for obtaining bounds on exponential sums based on linear algebraic properties of matrices of polynomials. We derive this lemma from Lemma 2.8.1 at the end of section 2.8, but include a direct proof here.

Lemma 2.4.1. *Let $Q \in \mathcal{P}(\mathbb{Z}, n)$, $M \in \{0,1\}^{m\times n}$ a matrix of Q modulo q, and $\mathbf{g} \in \{0,1\}^n$. Then for every $t \in [q]$*

$$\begin{aligned} E_{2,q}(\mathbf{g}, t\bar{Q}) &= 2^{-m} \sum_{\mathbf{y}\in K(M,\mathbf{g})} (\omega_q^t - \omega_q^t)^{|\mathbf{y}|}(\omega_q^t + \omega_q^t)^{m-|\mathbf{y}|} \\ &= 2^{-m} \sum_{\mathbf{y}\in K(M,\mathbf{g})} (\mathrm{i}\sin t\varphi_q)^{|y|}(\cos t\varphi_q)^{m-|y|}, \end{aligned}$$

where $|\mathbf{y}|$ is the number of 1's in $\mathbf{y}$.

Proof. Notice first that for $\mathbf{u} \in \mathbb{Z}^m$

$$\begin{aligned} \omega_q^{t\mathbf{1}^T(-1)^{\mathbf{u}}} &= 2^{-m}\prod_{i=1}^{m}((\omega_q^t + \bar{\omega}_q^t) + (-1)^{u_i}(\omega_q^t - \bar{\omega}_q^t)) \\ &= \widehat{\sum_{\mathbf{y}\in\{0,1\}^m}} (-1)^{\mathbf{y}^T\mathbf{u}}(\omega_q^t - \bar{\omega}_q^t)^{|\mathbf{y}|}(\omega_q^t + \bar{\omega}_q^t)^{m-|\mathbf{y}|}. \end{aligned}$$

Using that, by definition, $\bar{Q}(\mathbf{x}) \equiv_q \mathbf{1}^T(-1)^{M\mathbf{x}}$,

$$E_{2,q}(\mathbf{g}, t\bar{Q}) = \widehat{\sum_{\mathbf{y}\in\{0,1\}^m}} \left(\widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} (-1)^{(\mathbf{g}+M^T\mathbf{y})^T\mathbf{x}} \right) (\omega_q^t - \bar{\omega}_q^t)^{|\mathbf{y}|}(\omega_q^t + \bar{\omega}_q^t)^{m-|\mathbf{y}|}.$$

Now the first equality in the lemma follows from (2.4). The second follows by the definition of ω_q. □

As an immediate consequence of Lemma 2.4.1 we have that, if $K(M, \mathbf{g}) = \emptyset$, then $E_{2,q}(\mathbf{g}, t\bar{Q}) = 0$ for every $t \in [q]$. Hence if $\mathbf{0} \neq \mathbf{g} \notin I(M)$, then $C_{2,q}(\mathbf{g}, \bar{Q}) = 0$. Thus we get the following interesting statement.

Theorem 2.4.2. *Let $P, Q \in \mathcal{P}(\mathbb{Z}, n)$ such that $P(\mathbf{x}) \equiv_q \bar{Q}(\mathbf{x})$, for $\mathbf{x} \in \{0,1\}^n$. Let M be a matrix of Q modulo q and $\mathbf{0} \neq \mathbf{g} \in \{0,1\}^n$. If $\mathbf{g} \notin I(M)$, i.e. the rows of the matrix M do not span the vector $\mathbf{g}$ over $\mathbb{Z}_2$, then the correlation $C_{2,q}(\mathbf{g}, P)$ is 0*

This theorem extends the fact that if a polynomial P does not depend on all the variables over which we take parity, then the correlation between parity and P is zero.

2.5 The weight distribution modulo 4 of binary affine codes

In this section we digress somewhat from our goal on bounding the exponential sums $E_{2,q}$ for q odd and consider what happens to those sums when q is even. Of course, in this case we cannot hope to prove that in general the exponential sums are small, so our effort will go in a somewhat different direction. To motivate the discussion, for $M \in \{0,1\}^{m\times n}$ and $\mathbf{g} \in \{0,1\}^n$ define

$$W(x,y;M,\mathbf{g}) = \sum_{\mathbf{y}\in K(M,\mathbf{g})} x^{|y|}y^{m-|y|}.$$

Since $K(M,\mathbf{g})$ is the binary affine space which is the set of solution to $M^T\mathbf{y} = \mathbf{g}$ in $\mathbb{Z}_2^m$, $W(x,y;M,\mathbf{g})$ is the weight enumerator of this space. By Lemma 2.4.1 we have that

$$W(\mathrm{i}\sin\varphi_q, \cos\varphi_q; M, \mathbf{g}) = E_{2,q}(\mathbf{g}, \bar{Q}),$$

where Q is the polynomial of M modulo q. Let

$$W_q(M,\mathbf{g}) = W(\mathrm{i}\sin\varphi_q, \cos\varphi_q; M, \mathbf{g}).$$

We have that

$$\begin{aligned}
W_2(M,\mathbf{g}) &= \sum_{\mathbf{y}\in K(M,\mathbf{g})} 0^{|y|}(-1)^{m-|y|},\\
W_4(M,\mathbf{g}) &= \sum_{\mathbf{y}\in K(M,\mathbf{g})} \mathrm{i}^{|y|}0^{m-|y|},\\
W_8(M,\mathbf{g}) &= 2^{-m/2}\sum_{\mathbf{y}\in K(M,\mathbf{g})} \mathrm{i}^{|y|}.
\end{aligned}$$

Thus $W_2(M,\mathbf{g})$ is $(-1)^m$, if $\mathbf{g} = \mathbf{0}$, and 0, otherwise; $W_4(M,\mathbf{g})$ is i^m, if $M^T\mathbf{1} \equiv_2 \mathbf{g}$, and 0, otherwise. To see what is the significance of $W_8(M,\mathbf{g})$, define

$$L_i = \#\{\mathbf{y} \in K(M,\mathbf{g}) : |\mathbf{y}| \equiv_4 i\}, \text{ for } i \in [4].$$

L_i counts the number of elements of the affine space $K(M, \mathbf{g})$ which have weight i modulo 4. Then

$$W_8(M, \mathbf{g}) = 2^{-m/2}((L_1 - L_3) + \mathrm{i}(L_4 - L_2)).$$

The purpose of this sections is to derive relations between L_i, $i \in [4]$, which constrain the possible values for $W_8(M, \mathbf{g})$. From a code-theoretic point of view $K(M, \mathbf{g})$ is a binary affine code and L_i is the number of codewords of this code of weight i modulo 4. Investigations on the weight distribution of binary affine codes is an important subject and the considerations in this section constitute a contribution in this direction.

For $\mathbf{x} \in \{0,1\}^n$, by definition, $|\mathbf{x}|$ is the number of 1's in $\mathbf{x}$. For $\mathbf{x} \in \mathbb{Z}^n$ define

$$|\mathbf{x}| = \#\{i \in [n] : x_i \equiv_2 1\}.$$

For $A \in \{0,1\}^{m \times r}$, $\mathrm{rk}_2(A) = r$, and $\mathbf{b} \in \{0,1\}^m$, define

$$I(A, \mathbf{b}) = \{(\mathbf{b} + A\mathbf{x}) \bmod 2 : \mathbf{x} \in \{0,1\}^r\} \subseteq \{0,1\}^m.$$

Notice that over $\mathbb{Z}_2$, $I(A, \mathbf{b})$ is an affine space. For $i \in [4]$, define

$$L_i = \#\{\mathbf{x} \in I(A, \mathbf{b}) : |\mathbf{x}| \equiv_4 i\}.$$

The connection between our earlier notation, $K(M, \mathbf{g})$, for affine spaces and the new one is the following. Assume that $\mathrm{rk}_2(M) = n$. Take $r = m - n$. Let $N \in \{0,1\}^{n \times n}$ be the first n rows of M and $D \in \{0,1\}^{r \times n}$ are its last r rows. Assume that $\mathrm{rk}_2(N) = n$. Let

$$A = \begin{bmatrix} \bar{N}^T D^T \\ I_r \end{bmatrix} \in \{0,1\}^{m \times r}, \ \mathbf{b} = \begin{bmatrix} \bar{N}^T \mathbf{g} \\ \mathbf{0} \end{bmatrix} \in \{0,1\}^m,$$

where $\bar{N} \in \{0,1\}^{n \times n}$ is such that $N\bar{N} \equiv_2 I_n$. Then $K(M, \mathbf{g}) = I(A, \mathbf{b})$.

We will prove the following relations between L_i.

(i) $L_1 + L_2 + L_3 + L_4 = 2^r$,

(ii) $|(L_2 + L_4) - (L_1 + L_3)| \in \{0, 2^r\}$,

(iii) there exists an integer $0 \leq k \leq \min\{r, m - r\}$, such that $|L_2 - L_4|, |L_1 - L_3| \in \{0, 2^{\lfloor (r+k)/2 \rfloor}\}$.

(i) is trivial because $|I(A,\mathbf{b})| = 2^r$.

To see (ii) let $d = \mathbf{1}^T\mathbf{b}$ and $\mathbf{c} = A^T\mathbf{1}$. Then

$$\begin{aligned}(L_2 + L_4) - (L_1 + L_3) &= \sum_{\mathbf{y}\in I(A,\mathbf{b})} (-1)^{\mathbf{1}^T\mathbf{y}} = \sum_{\mathbf{x}\in\{0,1\}^r} (-1)^{\mathbf{1}^T(\mathbf{b}+A\mathbf{x})} \\ &= (-1)^d \sum_{\mathbf{x}\in\{0,1\}^r} (-1)^{\mathbf{c}^T\mathbf{x}} = (-1)^d(1-1)^{|\mathbf{c}|}(1+1)^{r-|\mathbf{c}|} \\ &= (-1)^d 0^{|\mathbf{c}|} 2^{r-|\mathbf{c}|},\end{aligned}$$

where $0^0 = 1$. Hence

$$(L_2 + L_4) - (L_1 + L_3) = \begin{cases} 0, & |\mathbf{c}| \neq 0, \\ (-1)^d 2^r, & \text{otherwise.} \end{cases}$$

To see (iii) we do the following. First notice that for every $z \in \mathbb{Z}$, $(z^2 \bmod 4) \in \{0,1\}$, and $z^2 \equiv_4 0$ iff $z \equiv_2 0$. Hence for every $\mathbf{x} \in \mathbb{Z}^r$, $|\mathbf{x}| \equiv_4 \mathbf{x}^T\mathbf{x}$. Therefore

$$|\mathbf{b} + A\mathbf{x}| \equiv_4 \mathbf{x}^T(A^TA)\mathbf{x} + 2(\mathbf{b}^TA)\mathbf{x} + \mathbf{b}^T\mathbf{b}.$$

Let $B = A^TA$, $\mathbf{c} = 2\mathbf{b}^TA$, and $d = \mathbf{b}^T\mathbf{b}$. B is a symmetric matrix and $\mathbf{x}^TB\mathbf{x}$ defines a quadratic form $Q(\mathbf{x}) = \sum_{i\leq j} c_{ij}x_ix_j$. Since B is symmetric, for $i \neq j$ we have that

$$c_{ij} = b_{ij} + b_{ji} = 2b_{ij} \equiv_2 0.$$

We say that the quadratic form Q is in a *canonical form*, if there is $S \subseteq [r]$ such that

1. if $i \in S$, then $i + 1 \notin S$,
2. $Q(\mathbf{x}) \equiv_4 \sum_{i\in[r]} c_{ii}x_i^2 + 2\sum_{i\in S} x_ix_{i+1}$, and
3. if $i \in S$, then $c_{ii} \equiv_2 c_{i+1,i+1} \equiv_2 0$.

Equivalently, B is in a canonical form, if modulo 4 it has a block-diagonal form with blocks $B_1, \ldots, B_t$ such that each block is either 1×1 or 2×2, and if B_i is 2×2, then $(B_i)_{12} \equiv_4 (B_i)_{21} \equiv_4 1$ and $(B_i)_{11} \equiv_2 (B_i)_{22} \equiv_2 0$.

Lemma 2.5.1. *There is a non-singular over $\mathbb{Z}_2$ matrix $H \in \mathbb{Z}^{r\times r}$ such that H^TBH is in a canonical form.*

Proof. Call a non-singular over $\mathbb{Z}_2$ matrix H good, if $H\mathbf{e}_1 = \mathbf{e}_1$, where $\mathbf{e}_1$ is the first standard basis vector. By induction on r we will prove that there exists a good matrix H which brings Q into a form for which only the first two properties of a canonical form are fulfilled. Call such form *almost canonical*.

Note that the lemma is about coming up with a good linear transformation of the variables of Q such that Q is brought into a canonical form. The property that $H\mathbf{e}_1 = \mathbf{e}_1$ is equivalent to the property that the linear transformation $\mathbf{x} \leftarrow H\mathbf{x}$ does not transform x_1. The proof proceeds by finding a good linear transformation which brings Q into a new form and afterwards assuming that Q was originally in the new form. This is justified because the set of good matrices is closed under multiplication.

For $r = 1, 2$ the statement is trivial. Assume it holds for $r - 1 \geq 2$. We will prove it for r.

We have that $Q(\mathbf{x}) = \sum_{j \in [r]} c_{1j} x_1 x_j + Q_1(x_2, \ldots, x_r)$. First notice that if $c_{1j} \equiv_4 0$, for $j > 1$, then the statement follows by applying the inductive hypothesis to Q_1. Otherwise there is $j > 1$ such that $c_{1j} \equiv_4 2$. By renaming the variables we can assume that $j = 2$. Then the linear transformation $x_2 \leftarrow \sum_{j>1}(c_{1j}/2)x_j$ is good and afterwards $Q(\mathbf{x}) \equiv_4 c_{11}x_1^2 + 2x_1x_2 + Q_2(x_2, \ldots, x_r)$.

Apply the inductive hypothesis to Q_2. Then $Q(\mathbf{x}) \equiv_4 c_{11}x_1^2 + 2x_1x_2 + Q_3(x_2, \ldots, x_r)$ and Q_3 is in an almost canonical form. Notice that since the linear transformation which brought Q_2 to its almost canonical form is good, the property that $c_{12} \equiv_4 2$ and $c_{1j} \equiv_4 0$, for every $j > 2$, is maintained. Since Q_3 is in an almost canonical form, modulo 4 Q has the form $c_{11}x_1^2 + 2x_1x_2 + c_{22}x_2^2 + c_{23}x_2x_3 + c_{33}x_3^2 + Q_4(x_3, \ldots, x_r)$.

If $c_{23} \equiv_4 0$, we are done. Otherwise modulo 4 Q_4 does not depend on x_3 (by property 1 of a canonical form). So $Q(\mathbf{x}) \equiv_4 c_{11}x_1^2 + 2x_1x_2 + c_{22}x_2^2 + 2x_2x_3 + c_{33}x_3^2 + Q_4(x_4, \ldots, x_r)$ and Q_4 is in an almost canonical form. Consider c_{33}. If $c_{33} \equiv_2 0$, then $x_3 \leftarrow x_1 + x_3$ eliminates $2x_1x_2$. If $c_{33} \equiv_2 1$, then $x_3 \leftarrow x_2 + x_3$ eliminates $2x_2x_3$. In both cases $c_{11}x_1^2 + 2x_1x_2 + c_{22}x_2^2 + 2x_2x_3 + c_{33}x_3^2$ is brought into an almost canonical form and we are done.

Everything that remains to be done is to convert from an almost canonical form

to a canonical form. To this end consider $ax^2 + 2xy + by^2$, for $a, b \in \mathbb{Z}_4$. If $a \in \{1, 3\}$, then $x \leftarrow x + y$, eliminates $2xy$. Similarly if $b \in \{1, 3\}$, then $y \leftarrow x + y$ eliminates $2xy$. In any case we can bring $ax^2 + 2xy + by^2$ into a canonical form. Therefore, if Q is an almost canonical form, we can bring $c_{ii}x_i^2 + 2x_ix_{i+1} + c_{i+1,i+1}x_{i+1}^2$, for every $i \in S$, into a canonical form. □

By Lemma 2.5.1, we can bring B into a canonical form with a non-singular over $\mathbb{Z}_2$ matrix H. Let $\bar{H} \in \{0,1\}^{r\times r}$ be such that $H\bar{H} \equiv_2 I_r$ We have that

$$\begin{aligned} I(A, \mathbf{b}) &= \{(\mathbf{b} + A\mathbf{x}) \bmod 2 : \mathbf{x} \in \{0,1\}^r\} = \{(\mathbf{b} + (AH)(\bar{H}\mathbf{x})) \bmod 2 : \mathbf{x} \in \{0,1\}^r\} \\ &= \{(\mathbf{b} + (AH)\mathbf{x}) \bmod 2 : \mathbf{x} \in \{0,1\}^r\} = I(AH \bmod 2, \mathbf{b}), \end{aligned}$$

where in the equality before the last we use that $\bar{H}\mathbf{x} \bmod 2$ is a permutation of $\{0,1\}^r$. Since $\mathrm{rk}_2(AH) = \mathrm{rk}_2(A) = r$, we can assume that B is in a canonical form.

For $\mathbf{x} \in \{0,1\}^r$, we have that

$$\begin{aligned} |\mathbf{b} + A\mathbf{x}| &\equiv_4 \mathbf{x}^T B\mathbf{x} + \mathbf{c}^T\mathbf{x} + d \\ &\equiv_4 \sum_{i\in[r]} c_{ii}x_i^2 + 2\sum_{i\in S} x_ix_{i+1} + \sum_{i\in[r]} c_ix_i + d \\ &\equiv_4 \sum_{i\in[r]} e_ix_i + 2\sum_{i\in S} x_ix_{i+1} + d, \end{aligned}$$

where $e_i = c_{ii} + c_i$. Then

$$\begin{aligned} (L_4 - L_2) + \mathrm{i}(L_1 - L_3) &= \sum_{\mathbf{y}\in I(A,\mathbf{b})} \mathrm{i}^{|\mathbf{y}|} = \sum_{\mathbf{x}\in\{0,1\}^r} \mathrm{i}^{|\mathbf{b}+A\mathbf{x}|} \\ &= \mathrm{i}^d \prod_{i\in S}(1 + \mathrm{i}^{e_i} + \mathrm{i}^{e_{i+1}} - \mathrm{i}^{e_i+e_{i+1}}) \prod_{i\in[r]-S}(1 + \mathrm{i}^{e_i}). \end{aligned}$$

The possible values of $1 + \mathrm{i}^x + \mathrm{i}^y - \mathrm{i}^{x+y}$ are

$$\begin{bmatrix} 2+2\,\mathrm{i} & 2\,\mathrm{i} & 0 & 2 \\ 2\,\mathrm{i} & -2 & -2\,\mathrm{i} & 2 \\ 0 & -2\,\mathrm{i} & 2-2\,\mathrm{i} & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix}$$

where the rows (columns) of the matrix are indexed by $x \in [4]$ ($y \in [4]$). Similarly the possible values of $1 + \mathrm{i}^x$ are

$$\begin{bmatrix} 1+\mathrm{i} & 0 & 1-\mathrm{i} & 2 \end{bmatrix}$$

indexed by $x \in [4]$.

For $i, j \in [4]$ let

$$s_{ij} = \#\{k \in S : ((e_k, e_{k+1}) \bmod 4) \in \{(i,j),(j,i)\}\},$$
$$s_i = \#\{k \in [r] - S : e_k \equiv_4 i\}.$$

Since $\mathbf{c} \equiv_2 \mathbf{0}$ and B is in a canonical form (more specifically, B has property 3), we have that $s_{ij} = 0$, if i or j is odd. Hence

$$(L_4 - L_2) + \mathrm{i}(L_1 - L_3) = 0^{s_2}(-1)^{s_{22}}\mathrm{i}^d 2^{|S|+s_4}(1+\mathrm{i})^{s_1}(1-\mathrm{i})^{s_3}.$$

If $s_2 \neq 0$, then $0^{s_2} = 0$ and (iii) follows trivially. So we will assume that $s_2 = 0$. Since we are interested only in the absolute value of the real and imaginary parts of the complex number above, we can concentrate on $2^{|S|+s_4}(1+\mathrm{i})^{s_1}(1-\mathrm{i})^{s_3}$ (the other terms only change the sign and/or swap the real and imaginary parts). We have that

$$(1+\mathrm{i})^x(1-\mathrm{i})^y = 2^{(x+y)/2}(\cos\frac{\pi}{4}(x-y) + \mathrm{i}\sin\frac{\pi}{4}(x-y)).$$

Hence

$$2^{|S|+s_4}(1+\mathrm{i})^{s_1}(1-\mathrm{i})^{s_3} = 2^{|S|+s_4+(s_1+s_3)/2}(\cos\psi + \mathrm{i}\sin\psi),$$

where $\psi = \frac{\pi}{4}(s_1 - s_3)$. Since $s_2 = 0$, $s_1 + s_3 = r - 2|S| - s_4$, and hence

$$|S| + s_4 + (s_1+s_3)/2 = (r+s_4)/2.$$

Now using that $|\sin\psi|, |\cos\psi| \in \{0, 1, 2^{-1/2}\}$, we obtain that

$$|L_2 - L_4|, |L_1 - L_3| \in \{0, 2^{(r+s_4)/2}, 2^{(r+s_4-1)/2}\}.$$

Therefore, since $L_2 - L_4$ and $L_1 - L_3$ are integers, (iii) is valid with $k = s_4$. Since $s_4 \leq r$ is obvious, we have to check only that $s_4 \leq m - r$. For this we will prove the following lemma.

Lemma 2.5.2. *Let $S_1, S_2 \subseteq \mathbb{Z}_2^m$ and both are linearly independent. If for every $\mathbf{x} \in S_1$ and $\mathbf{y} \in S_2$, $\mathbf{x}^T\mathbf{y} = 0$, then $|S_1| + |S_2| \leq m$*

Proof. By the assumption of the lemma $S_2 \subseteq \text{span}(S_1)^\perp$. Since the vectors in S_2 are linearly independent, $|S_2| \leq \dim \text{span}(S_1)^\perp = m - |S_1|$, where in the equality we use that the vectors in S_1 are linearly independent. Hence

$$|S_1| + |S_2| \leq |S_1| + m - |S_1| \leq m.$$

□

Let $S_1 = \{A_{\cdot i} : i \notin S, b_{ii} \equiv_2 0\}$ and $S_2 = \{A_{\cdot i} : i \in [r]\}$, where $A_{\cdot i}$ is the i-th column of A. Notice that for $i, j \in [r]$, $A_{\cdot i}^T A_{\cdot j} = b_{ij}$, and therefore the vectors in S_1 and S_2 fulfill the conditions of Lemma 2.5.2, from which follows that $(s_4 + s_2) + r = |S_1| + |S_2| \leq m$. Hence $s_4 + s_2 \leq m - r$ and in particular $s_4 \leq m - r$.

Let us see now that s_4 can possibly take any value in $\{0, \ldots, r\}$. Take $M \in \{0,1\}^{r \times r}$ of rank r over $\mathbb{Z}_2$. Let $M' \in \{0,1\}^{4r \times r}$ be M with every 0 substituted with a column of four 0's, and every 1 with a column of four 1's. Fix $s \in \{0, \ldots, r\}$ and define

$$A = \left[\begin{array}{c|c} M' & \\ \hline \mathbf{0} & I_{r-s} \end{array}\right].$$

Obviously A has rank r over $\mathbb{Z}_2$. Also

$$B \equiv_4 \left[\begin{array}{c|c} \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & I_{r-s} \end{array}\right],$$

and hence $s_4 = s$.

2.6 Examples on bounding the exponential sum

In this section we give some examples on bounding the exponential sums $E = |E_{2,q}(\mathbf{1}, \bar{Q})|$ for q odd and $Q \in \mathcal{P}(\mathbb{Z}, n)$. The point on one hand is to show how those sums can be bounded in some simple cases and on the other to rule out some "intuitive conjectures", which turn out to be false, on bounding the exponential sums. Let M be a matrix of Q modulo q.

Since our techniques rely on Lemma 2.4.1, it seems plausible to conjecture that the bounds on E depend on linear algebraic properties of M such as its rank over $\mathbb{Z}_2$. If $Q(y) = y_1$ and $n > 1$, then $E = 0$. Notice that the rank over $\mathbb{Z}_2$ of the defining matrix

of Q modulo q is 1. Hence one might conjecture that, if the rank of M is small then E is also small. As Example 1 shows this is not true – there is a rank one matrix such that E is large. On the other hand, we will see later (Theorem 2.7.1) that E is small, if M is non-singular over $\mathbb{Z}_2$, and in particular has full-rank. Therefore, one might suspect that large rank means small E. Example 2 show that this proposition is also false. Thus the dependence of E on the rank of M is unclear. In Theorem 2.7.5 we weaken the condition of Theorem 2.7.1 that M is non-singular over $\mathbb{Z}_2$. Theorem 2.7.5 becomes cleaner, if in addition q is prime and it is interesting whether this condition on q can be removed. As Example 3 shows it cannot.

So far many of our bounds are obtained by more or less using triangle inequality in Lemma 2.4.1. It is important to see how much we lose in this and in particular is it possible that a bound obtained by applying triangle inequality to Lemma 2.4.1 is small, when E is small. As expected, Example 4 shows that this is not true and when bounding E we have to use more of its properties. Since the terms of the expression in Lemma 2.4.1 contain powers of i and, furthermore, $\sin t\varphi_q$ and $\cos t\varphi_q$ can themselves be negative, one might suspect that the cancellation that happens between different terms is the reason why triangle inequality does not give a bound close to E. As Example 5 shows, it is possible to have all terms of the expression in Lemma 2.4.1 to be real and non-negative and still E to be small.

There is one tantalizing conjecture whose validity we have not been able to verify. As can be seen easily, if M has at most d ones in each of its rows (corresponding to the fact that we are interested in polynomials of degree at most d), then every element of $K(M, \mathbf{1})$ has weight at least n/d. We are interested in giving a bound on E of the type γ^e, for some $\gamma < 1$ which depends on q and some e which depends on n and d. From Green's result (Theorem 2.3.2), see also the discussion following Theorem 2.7.2, we can conjecture that e is something like n/d and so the possibility arises that e is actually related to the smallest weight of an element of $K(M, \mathbf{1})$. So far we have not been able to give a counterexample to this. As the results in sections 2.8.1 and 2.8.2 show the exponent might actually be n/d^s, for some s depending on q. Of course, if it turns out that the

best exponent we can get is $n/2^{\Omega(d)}$, then Bourgain's result (Theorem 2.3.3) will be best possible.

Example 1 Let $Q(\mathbf{y}) = y_1 y_2 \dots y_n - 1$. Since q is odd,

$$\bar{Q}(\mathbf{x}) \equiv_q 0 \text{ iff } \mathbf{1}^T\mathbf{x} \equiv_2 0$$

and $C_{2,q}(\mathbf{1}, \bar{Q}) = 1$. Let M be the matrix with q rows which has $\mathbf{1}$ as its first row and $\mathbf{0}$ in the rest. Then M is the defining matrix of Q modulo q and has rank 1 over $\mathbb{Z}_2$. Also, for $t \in [q]$,

$$E_{2,q}(\mathbf{1}, t\bar{Q}) = 2^{-n}\bar{\omega}_q^t \sum_{\mathbf{x}\in\{0,1\}^n} (-1)^{\mathbf{1}^T\mathbf{x}}\omega_q^{t(-1)^{\mathbf{1}^T\mathbf{x}}} = (1 - \bar{\omega}_q^{2t})/2$$

and so $|E_{2,q}(\mathbf{1}, t\bar{Q})| = |\sin t\varphi_q|$.

Example 2 Let Q be the polynomial which has all monomials as terms with coefficient 1 and $M \in \{0,1\}^{2^n\times n}$ be the matrix with rows all elements of $\{0,1\}^n$. Then M is the defining matrix of Q modulo q, M has rank n over $\mathbb{Z}_2$, and we have that

$$\bar{Q}(\mathbf{x}) = \mathbf{1}^T(-1)^{M\mathbf{x}} = \begin{cases} 2^n, & \mathbf{x} = \mathbf{0}, \\ 0, & \text{otherwise.} \end{cases}$$

Assume that $\mathbf{g} \neq \mathbf{0}$ and $g_1 = 1$. Then

$$C_{2,q}(\mathbf{g}, \bar{Q}) = 2^{-n}(-1-1) = -2^{n+1}.$$

and

$$E_{2,q}(\mathbf{g}, t\bar{Q}) = 2^{-n}(\omega_q^{t2^n} - 1).$$

For $g = \mathbf{0}$ we obtain

$$C_{2,q}(\mathbf{0}, \bar{Q}) = 2^{-n}(2^n - 1 - 1) = 1 - 2^{-n+1}$$

and

$$E_{2,q}(\mathbf{0}, t\bar{Q}) = 2^{-n}((2^n - 1) + \omega_q^{t2^n}) = 1 - 2^{-n}(1 - \omega_q^{t2^n}).$$

$\bar{Q}$ is almost constant – it is non-zero only for $\mathbf{x} = \mathbf{0}$. If on the other hand we wanted a polynomial Q such that $\bar{Q}$ is constant modulo q, then it is not hard to see using

Fourier analysis on the boolean cube over $\mathbb{Z}_q$ (this is possible because q is odd), that all coefficients of non-constant terms of Q must be 0 modulo q.

Example 3 Let $Q(\mathbf{y}) = \prod_{i=1}^{n} y_i + 3\sum_{i=2}^{n} y_i - 1$ and M be the matrix with $3n+q$ rows such that its first row is $\mathbf{1}$, the next $q-1$ are $\mathbf{0}$, and the rest are I_n repeated three times. M is the defining matrix of Q modulo q and it has rank n over $\mathbb{Z}_2$. We will see that $C_{2,15}(\mathbf{1}, \bar{Q})$ is close to $1/5$.

Since $5Q(\mathbf{y}) \equiv_{15} 5\prod_{i=1}^{n} y_i - 5$, we have that

$$\begin{aligned} E_{2,15}(\mathbf{1}, 5\bar{Q}) &= \bar{\omega}_{15}^{5} \widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} (-1)^{\mathbf{1}^T\mathbf{x}} \omega_{15}^{5\mathbf{1}^T(-1)^{\mathbf{x}}} \\ &= \bar{\omega}_3 \widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} (-1)^{\mathbf{1}^T\mathbf{x}} \omega_3^{\mathbf{1}^T(-1)^{\mathbf{x}}} = (1-\omega_3)/2. \end{aligned}$$

Similarly $E_{2,15}(\mathbf{1}, 10\bar{Q}) = (1-\bar{\omega}_3)/2$

Consider $t \in [15] - \{5, 10, 15\}$. Then $tQ(\mathbf{y}) = Q_1(\mathbf{y}) + \sum_{i=2}^{n} Q_i(\mathbf{y}) - t$ where $Q_1(\mathbf{y}) = t\prod_{i=1}^{n} y_i - t$ and $Q_i(\mathbf{y}) = 3ty_i$, $i \geq 2$. Notice that each Q_i, $i \geq 1$, is non-trivial modulo 15 and its defining matrix modulo 15 has rank 1 over $\mathbb{Z}_2$, and so by Theorem 2.7.5,

$$|E_{2,15}(\mathbf{1}, t\bar{Q})| \leq 2^{-\Omega(n)}.$$

From Lemma 2.2.1 and (2.2) follows that

$$\begin{aligned} C_{2,15}(\mathbf{1}, \bar{Q}) &= \frac{1}{15}((1-\omega_3) + (1-\bar{\omega}_3)) + \frac{2}{15} \sum_{t\in[15]-\{5,10,15\}} \bar{\omega}_{15}^{t} E_{2,15}(\mathbf{1}, t\bar{Q}) \\ &= \frac{1}{5} + \frac{2}{15} \sum_{t\in[15]-\{5,10,15\}} \bar{\omega}_{15}^{t} E_{2,15}(\mathbf{1}, t\bar{Q}). \end{aligned}$$

and therefore

$$|C_{2,15}(\mathbf{1}, \bar{Q}) - 1/5| \leq 2^{-\Omega(n)}.$$

Example 4 Let $Q(\mathbf{y}) = 3\sum_{i\in[n]} y_i$ and M be the matrix with $3n$ rows which is I_n repeated 3 times. M is the defining matrix of Q modulo q. From Theorem 2.7.2 follows that $|E_{2,q}(\mathbf{1}, t\bar{Q})| \leq 2^{-\Omega(n)}$, for any $t \in [q-1]$. Our goal is to show that for some values of q and t

$$S = \sum_{y\in K(M,\mathbf{1})} |\sin t\varphi_q|^{|y|} |\cos t\varphi_q|^{3n-|y|}$$

is exponentially large in n. By Lemma 2.4.1, S is obtained by applying triangle inequality to $|E_{2,q}(\mathbf{1}, t\bar{Q})|$, thus $|E_{2,q}(\mathbf{1}, t\bar{Q})| \leq S$.

For every $\lambda, \mu \in \mathbb{C}$, we have that

$$\begin{aligned}\sum_{\mathbf{y} \in K(M,\mathbf{1})} \lambda^{|\mathbf{y}|} \mu^{3n-|\mathbf{y}|} &= \sum_{\mathbf{x} \in \{0,1\}^n} 3^{n-|\mathbf{x}|} \lambda^{2|\mathbf{x}|+n} \mu^{3n-(2|\mathbf{x}|+n)} \\ &= \lambda^n \sum_{\mathbf{x} \in \{0,1\}^n} \left(\lambda^2\right)^{|\mathbf{x}|} \left(3\mu^2\right)^{n-|\mathbf{x}|} \\ &= \lambda^n \left(\lambda^2 + 3\mu^2\right)^n = (\lambda(\lambda^2 + 3\mu^2))^n,\end{aligned}$$

where the first equality follows from the fact that the parity of three bits is 1 iff either exactly one or exactly three of them are 1, and if exactly one of them is 1 we have three choices for which bit to set to 1. The other equalities are simple arithmetic.

As can be seen from the graph of $f(\varphi) = |\sin\varphi|(\sin^2\varphi + 3\cos^2\varphi)$ for $\varphi \in (0, 2\pi)$, there exists $\varphi \in (0, 2\pi)$ such that $f(\varphi) > 1$. Hence for sufficiently large q we can always find t such that $f(2\pi t/q)) > 1$. Since $S = (f(2\pi t/q))^n$, it is exponentially large for such values of q and t.

Example 5 Fix $n \equiv_4 0$ and take the first n rows of M to be I_n and the rest of its rows to be $[\mathbf{1}_{(n-4)\times 4} \,|\, I_{n-4}]$. Then $m = 2n - 4$. $K(M, \mathbf{1})$ contains $\frac{1}{2}2^{n-4} = 2^{n-5}$ elements of weight $n - 4$ and 2^{n-5} elements of weight n. Since $n \equiv_4 0$, it follows that all elements of $K(M, \mathbf{1})$ have weight 0 modulo 4. Let Q be the polynomial of M. From Lemma 2.4.1 follows that

$$\begin{aligned}E_{2,q}(\mathbf{1}, \bar{Q}) &= 2^{n-5}(\sin^{n-4}\varphi_q \cos^n\varphi_q + \sin^n\varphi_q \cos^{n-4}\varphi_q) \\ &= \frac{1}{2}(2\sin\varphi_q\cos\varphi_q)^{n-4}(\cos^4\varphi_q + \sin^4\varphi_q) \\ &= \frac{1}{2}(\sin^{n-4} 2\varphi_q)(2 - \sin^2 2\varphi_q) = \frac{2-\gamma^2}{2}\gamma^{n-4},\end{aligned}$$

where $\gamma = \sin 2\varphi_q$.

2.7 Bounds based on linear algebraic properties of matrices

For q odd, Lemma 2.4.1 allows us to estimate the correlation between a polynomial P and parity, based on linear algebraic properties over $\mathbb{Z}_2$ of a matrix $M \in \{0,1\}^{m\times n}$. Recall

(section 2.4) that to obtain M, first we obtain a polynomial Q such that $\bar{Q}(\mathbf{x}) \equiv_q P(\mathbf{x})$, for $\mathbf{x} \in \{0,1\}^n$, and then M is taken to be a matrix of Q modulo q. Our methods can be used to estimate the correlation between polynomials and parity over an arbitrary subset of the variables: parity is taken over the coordinates that are 1 in the vector $\mathbf{g}$; taking parity of all the variables corresponds to using $\mathbf{g} = \mathbf{1}$.

In this section $q \geq 3$ is an odd integer, $Q \in \mathcal{P}(\mathbb{Z}, n)$, $M \in \{0,1\}^{m \times n}$ a matrix of Q modulo q, and $\mathbf{g}$ a vector in $\{0,1\}^n$. Recall that, by definition,

$$\bar{Q}(\mathbf{x}) = Q((-1)^{\mathbf{x}}) \equiv_q \mathbf{1}^T (-1)^{M\mathbf{x}}, \text{ for } \mathbf{x} \in \{0,1\}^n,$$

and, by (2.1),

$$\max_{t \in [q-1]} \{|\sin t\varphi_q|, |\cos t\varphi_q|\} \leq \gamma_q < 1.$$

By Lemma 2.4.1, if $\mathbf{g} \notin I(M)$, then $E_{2,q}(\mathbf{g}, t\bar{Q}) = 0$, $t \in [q]$. Estimating $E_{2,q}(\mathbf{g}, t\bar{Q})$ when $\mathbf{g} \in I(M)$ is a challenging task in general. Of course when $t \equiv_q 0$, the exponential sum is either 1 or 0, depending on whether $\mathbf{g} = \mathbf{0}$. We show that $|E_{2,q}(\mathbf{g}, t\bar{Q})|$, $t \in [q-1]$, is exponentially small, if M is suitably restricted. While our estimates on $|E_{2,q}(\mathbf{g}, t\bar{Q})|$ apply to arbitrary $\mathbf{g}$, and give good bounds even for $\mathbf{g} = \mathbf{0}$, from the point of view of estimating $|C_{2,q}(\mathbf{g}, \bar{Q})|$, by Lemma 2.2.1 those bounds are interesting only for $\mathbf{g} \neq \mathbf{0}$.

An important example to keep in mind in the following is when the variables of Q are partitioned into disjoint sets $S_1, \ldots, S_k$ and Q is a sum of polynomials $Q_1, \ldots, Q_k$ such that Q_i is a polynomial only on the variables in S_i. The extreme case of this is when $S_i = \{i\}$ and $Q_i(y_i) = y_i$, i.e. $Q(\mathbf{y}) = \sum_{i \in [n]} y_i$. In this case, M is obviously the identity matrix. With the same S_i, but having $Q_i(y_i) = c_i y_i$, for some $c_i \in \mathbb{Z}$, we obtain that Q is a linear polynomial and M can be taken to be the identity matrix with each of its rows repeated certain number of times. In the most general case of this example S_i is not necessary of size 1 for each $i \in [k]$ and then M has a block-diagonal form.

The following simple theorem covers the simplest of the cases described in the previous paragraph, i.e. when M is identity, and even more generally when M is non-singular modulo 2. The idea of the proof is at the bottom of the estimates in this section.

Theorem 2.7.1. *Let $Q \in \mathcal{P}(\mathbb{Z}, n)$, M be a matrix of Q modulo q, and $\mathbf{g} \in \{0,1\}^n$. If M is non-singular over $\mathbb{Z}_2$ and $t \in [q-1]$, then*

$$|E_{2,q}(\mathbf{g}, t\bar{Q})| \leq \gamma_q^n.$$

Proof. By Lemma 2.4.1

$$E_{2,q}(\mathbf{g}, t\bar{Q}) = \sum_{\mathbf{y} \in K(M,\mathbf{g})} (\mathrm{i} \sin t\varphi_q)^{|\mathbf{y}|} (\cos t\varphi_q)^{n-|\mathbf{y}|}.$$

Since M is non-singular over $\mathbb{Z}_2$, there is a unique $\mathbf{y} \in \{0,1\}^n$ such that $M^T\mathbf{y} \equiv_2 \mathbf{g}$. Hence

$$|E_{2,q}(\mathbf{g}, t\bar{Q})| = |\sin t\varphi_q|^{|\mathbf{y}|} |\cos t\varphi_q|^{n-|\mathbf{y}|}.$$

□

In the following we will prove theorems which cover the more general cases of the example described above, and in particular when Q is a linear polynomial. First, let us define a generalization of the notion of affine space.

Definition 2.7.1. *Let $M_i \in \{0,1\}^{m_i \times n}$ for $i \in [k]$, and $\mathbf{g} \in \{0,1\}^n$. Define*

$$J(M_1, \ldots, M_k; \mathbf{g}) = \left\{ (\mathbf{g}_1, \ldots, \mathbf{g}_k) \in I(M_1) \times \cdots \times I(M_k) : \sum_{i \in [k]} \mathbf{g}_i \equiv_2 \mathbf{g} \right\}.$$

For vectors $\mathbf{g}_1, \ldots, \mathbf{g}_k$, $\mathbf{g}_i \in \{0,1\}^{m_i}$, let the weight of $(\mathbf{g}_1, \ldots, \mathbf{g}_k)$, $\mathrm{wt}(\mathbf{g}_1, \ldots, \mathbf{g}_k)$, be the number of non-zero vectors. Define the weight of $J(M_1, \ldots, M_k; \mathbf{g})$

$$\mathrm{wt}(M_1, \ldots, M_k; \mathbf{g}) = \min\{\mathrm{wt}(\mathbf{g}_1, \ldots, \mathbf{g}_k) : (\mathbf{g}_1, \ldots, \mathbf{g}_k) \in J(M_1, \ldots, M_k; \mathbf{g})\}.$$

To see that the definition of $J(M_1, \ldots, M_k; \mathbf{g})$ contains as a special case the definition of the affine space $K(M, \mathbf{g})$, notice that if M_i, $i \in [m]$, is the i-th row of M, then $J(M_1, \ldots, M_k; \mathbf{g})$ is essentially $K(M, \mathbf{g})$. Just like $|K(M, \mathbf{g})| = 2^{m - \mathrm{rk}_2(M)}$, we have that if $M_1, \ldots, M_k$ form a partition of the rows of M, then

$$|J(M_1, \ldots, M_k; \mathbf{g})| = 2^{\sum_{i \in [k]} \mathrm{rk}_2(M_i) - \mathrm{rk}_2(M)}.$$

When M is the identity matrix with each of its repeated a certain number of times, i.e. when Q is a linear polynomial, then in general M is not non-singular modulo 2. If we let $M_1, \ldots, M_n$ be the partition of the rows of M such that each block contains all copies of the same row and assume that $\mathbf{g} \in I(M)$, then we recover the property that a linear systems in which the matrix is non-singular has a unique solution, which was used in the proof of Theorem 2.7.1, in the sense that $J(M_1, \ldots, M_k; \mathbf{g})$ has exactly one element. The following theorem generalizes this idea by bounding the exponential sum, provided that the size of $J(M_1, \ldots, M_k; \mathbf{g})$ is small and its weight is large.

Theorem 2.7.2. *Let $Q \in \mathcal{P}(\mathbb{Z}, n)$, M be a matrix of Q modulo q, and $\mathbf{g} \in \{0,1\}^n$. If $M_1, \ldots, M_k$ is a partition of M and $t \in [q-1]$, then*

$$|E_{2,q}(\mathbf{g}, t\bar{Q})| \leq |J(M_1, \ldots, M_k; \mathbf{g})| \gamma_q^{\mathrm{wt}(M_1, \ldots, M_k; \mathbf{g})}.$$

In addition to having the size of $J(M_1, \ldots, M_k; \mathbf{g})$ to be small, to give a good bound this theorem needs $\mathrm{wt}(M_1, \ldots, M_k; \mathbf{g})$ to be large. One particular case when this happens is when $\mathbf{g} = \mathbf{1}$ and for $i \in [k]$, all vectors in $I(M_i)$ have weight at most d. In this case $\mathrm{wt}(M_1, \ldots, M_k; \mathbf{1}) \geq n/d$. This covers the case when Q is a linear polynomial, i.e. $d = 1$. Thus, the above theorem contains Goldmann's result [Gol95] on the correlation between parity and linear polynomials as a special case. Furthermore, this result implies an exponentially small upper bound on the absolute value of the correlation between parity and polynomials possibly with arbitrarily large degree that previous techniques did not apply to. It is also very interesting that we get exponentially small correlation with respect to parity over arbitrary nonempty subsets of the variables.

We will need the following lemmas.

Lemma 2.7.3. *Let $Q \in \mathcal{P}(\mathbb{Z}, n)$ and $\mathbf{g} \in \{0,1\}^n$.*

(i) If $\mathbf{g} \neq \mathbf{0}$, then $|E_{2,q}(\mathbf{g}, \bar{Q})| \leq \gamma_q$.

(ii) $|E_{2,q}(\mathbf{0}, \bar{Q})| < 1$ iff $\bar{Q}$ is not constant modulo q.

Proof. (i) Assume w.l.o.g. that the last coordinate of $\mathbf{g}$ is nonzero. Let

$$Q(y_1, \ldots, y_{n-1}, z) = R(\mathbf{y}) + S(\mathbf{y})z.$$

Let $\mathbf{h} \in \{0,1\}^{n-1}$ be the vector $\mathbf{g}$ without its last coordinate. Then

$$E_{2,q}(\mathbf{g}, \bar{Q}) = \frac{1}{2} \widehat{\sum_{\mathbf{x} \in \{0,1\}^{n-1}}} (-1)^{\mathbf{h}^T \mathbf{x}} \left(\omega_q^{\bar{R}(\mathbf{x}) + \bar{S}(\mathbf{x})} - \omega_q^{\bar{R}(\mathbf{x}) - \bar{S}(\mathbf{x})} \right).$$

Thus, there exists $\mathbf{x} \in \{0,1\}^{n-1}$ such that for $a = \bar{R}(\mathbf{x})$ and $b = \bar{S}(\mathbf{x})$,

$$|E_{2,q}(\mathbf{g}, \bar{Q})| \leq |\omega_q^{a+b} - \omega_q^{a-b}|/2 = |\sin b\varphi_q|.$$

(ii) First, if $\bar{Q}(\mathbf{x}) \equiv_q c$, for every $\mathbf{x} \in \{0,1\}^n$, then $E_{2,q}(\mathbf{0}, \bar{Q}) = \omega_q^c$ and the lemma follows. Otherwise, there exist $a \not\equiv_q b$ such that $\bar{Q}$ takes these values for some vectors in $\{0,1\}^n$. Observe that $|\omega_q^a + \omega_q^b| = |\omega_q^{(a-b)/2} + \bar{\omega}_q^{(a-b)/2}| = 2|\cos \pi(a-b)/q| < 2$. Therefore, in this case, the claim follows by using triangle inequality applied to the definition of $E_{2,q}(\mathbf{0}, \bar{Q})$. □

Lemma 2.7.4. *Let $Q \in \mathcal{P}(\mathbb{Z}, n)$, M be a matrix of Q modulo q, and $\mathbf{g} \in \{0,1\}^n$. If $M_1, \ldots, M_k$ are a partition of the rows of M, $Q_1, \ldots, Q_k$ are their polynomials modulo q, and $t \in [q]$, then*

$$E_{2,q}(\mathbf{g}, t\bar{Q}) = \sum_{(\mathbf{g}_1, \ldots, \mathbf{g}_k) \in J(M_1, \ldots, M_k; \mathbf{g})} \prod_{i=1}^{k} E_{2,q}(\mathbf{g}_i, t\bar{Q}_i).$$

Proof. Using Lemma 2.4.1,

$$\begin{aligned} E_{2,q}(\mathbf{g}, t\bar{Q}) &= 2^{-m} \sum_{\mathbf{y} \in K(M, \mathbf{g})} (\omega_q^t - \bar{\omega}_q^t)^{|\mathbf{y}|} (\omega_q^t + \bar{\omega}_q^t)^{m - |\mathbf{y}|} \\ &= \sum_{(\mathbf{g}_1, \ldots, \mathbf{g}_k) \in J(M_1, \ldots, M_k; \mathbf{g})} \prod_{i=1}^{k} \left(2^{-m_i} \sum_{\mathbf{y} \in K(M_i, \mathbf{g}_i)} (\omega_q^t - \bar{\omega}_q^t)^{|\mathbf{y}|} (\omega_q^t + \bar{\omega}_q^t)^{m_i - |\mathbf{y}|} \right) \\ &= \sum_{(\mathbf{g}_1, \ldots, \mathbf{g}_k) \in J(M_1, \ldots, M_k; \mathbf{g})} \prod_{i=1}^{k} E_{2,q}(\mathbf{g}_i, t\bar{Q}_i). \end{aligned}$$

□

Proof of Theorem 2.7.2. If $\mathbf{g} \notin I(M)$, then $E_{2,q}(\mathbf{g}, t\bar{Q}) = 0$. Assume that $\mathbf{g} \in I(M)$. By definition, every element $(\mathbf{g}_1, \ldots, \mathbf{g}_k)$ of $J(M_1, \ldots, M_k; \mathbf{g})$ has at least $\mathrm{wt}(M_1, \ldots, M_k; \mathbf{g})$ non-zero components. By Lemma 2.7.3(i), for every $i \in [k]$ such that $\mathbf{g}_i \neq \mathbf{0}$ we have that

$|E_{2,q}(\mathbf{g}_i, t\bar{Q}_i)| \leq \gamma_q$, where Q_i is the polynomial of M_i modulo q. Finally, using Lemma 2.7.4,

$$\begin{aligned}|E_{2,q}(\mathbf{g}, t\bar{Q})| &\leq \sum_{(\mathbf{g}_1,\ldots,\mathbf{g}_k)\in J(M_1,\ldots,M_k;\mathbf{g})} \prod_{i=1}^{k} |E_{2,q}(\mathbf{g}_i, t\bar{Q}_i)| \\ &\leq |J(M_1,\ldots,M_k;\mathbf{g})|\gamma_q^{\mathrm{wt}(M_1,\ldots,M_k;\mathbf{g})}.\end{aligned}$$

□

As we mentioned earlier, Theorem 2.7.2, applies when the size of $J(M_1,\ldots,M_k;\mathbf{g})$ is small and its weight is large. The next theorem gives a bound independent of the weight of $J(M_1,\ldots,M_k;\mathbf{g})$, but it requires more properties from the partition of M.

Theorem 2.7.5. *Let $Q \in \mathcal{P}(\mathbb{Z}, n)$, M be a matrix of Q modulo q, and $\mathbf{g} \in \{0,1\}^n$. Let $M_1,\ldots,M_k$ be a partition of M and Q_i be the polynomial of M_i modulo q. Let $r = \max_{i\in[k]} \mathrm{rk}_2(M_i)$. Assume that for every $i \in [k]$, $\bar{Q}_i$ is not constant modulo q. Then there exists $\beta \in [0,1)$ (depending only on q and r) such that,*

$$|E_{2,q}(\mathbf{g}, \bar{Q})| \leq |J(M_1,\ldots,M_k;\mathbf{g})|\beta^k.$$

Furthermore, if for every $i \in [k]$ the coefficients of Q_i modulo q are in $\mathbb{Z}_q^* \cup \{0\}$, then for every $t \in [q-1]$ the bound holds for $t\bar{Q}$.

First let us prove a lemma that will allow us to assume that the number of variables of Q is equal to the rank of M over $\mathbb{Z}_2$.

Lemma 2.7.6. *Let $Q \in \mathcal{P}(\mathbb{Z}, n)$, M be a matrix of Q modulo q, and $\mathbf{g} \in \{0,1\}^n$. Let $\mathrm{rk}_2(M) = r$ and the first r columns of M are linearly independent over $\mathbb{Z}_2$. Let $\hat{M} \in \{0,1\}^{m\times r}$ be M with its last $n-r$ columns removed and R be the polynomial of $\hat{M}$ modulo q. Similarly, define $\hat{\mathbf{g}} \in \{0,1\}^r$ to be $\mathbf{g}$ with its last $n-r$ coordinates removed. Then for every $t \in [q]$, $E_{2,q}(\mathbf{g}, t\bar{Q})$ is either 0 or $E_{2,q}(\hat{\mathbf{g}}, t\bar{R})$.*

Proof. For $\mathbf{y} \in \{0,1\}^r$, let $\check{\mathbf{y}} \in \{0,1\}^n$ be $\mathbf{y}$ with $n-r$ 0's added to it. Given $\mathbf{x} \in \{0,1\}^n$, let $\mathbf{y} \in \{0,1\}^r$ be the unique vector such that $\hat{M}\mathbf{y} \equiv_2 M\check{\mathbf{y}} \equiv_2 M\mathbf{x}$. There is a unique $\mathbf{y}$ with these properties, because the first r columns of M are linearly independent over $\mathbb{Z}_2$

and $\mathrm{rk}_2(M) = r$. Then for $\mathbf{z} = (\mathbf{x}+\check{\mathbf{y}}) \bmod 2$ we have that $M\mathbf{z} \equiv_2 \mathbf{0}$, that is $\mathbf{z} \in K(M, \mathbf{0})$. Thus,

$$\begin{aligned} E_{2,q}(\mathbf{g}, t\bar{Q}) &= \widehat{\sum_{\mathbf{x}\in\{0,1\}^n}} (-1)^{\mathbf{g}^T\mathbf{x}} \omega_q^{t\mathbf{1}^T(-1)^{M\mathbf{x}}} \\ &= 2^{-n} \sum_{\mathbf{y}\in\{0,1\}^r} \sum_{\mathbf{z}\in K(M,\mathbf{0})} (-1)^{\mathbf{g}^T(\check{\mathbf{y}}+\mathbf{z})} \omega_q^{t\mathbf{1}^T(-1)^{M(\check{\mathbf{y}}+\mathbf{z})}} \\ &= 2^{r-n} \left(\sum_{\mathbf{z}\in K(M,\mathbf{0})} (-1)^{\mathbf{g}^T\mathbf{z}} \right) \widehat{\sum_{\mathbf{y}\in\{0,1\}^r}} (-1)^{\hat{\mathbf{g}}^T\mathbf{y}} \omega_q^{t\mathbf{1}^T(-1)^{\hat{M}\mathbf{y}}}. \end{aligned}$$

Since $\sum_{\mathbf{z}\in K(M,\mathbf{0})}(-1)^{\mathbf{g}^T\mathbf{z}}$ is either 0 or 2^{n-r} (see (ii) in section 2.5), depending on M and $\mathbf{g}$, the lemma follows. □

Proof of Theorem 2.7.5. Define

$$\begin{aligned} \alpha &= \max_R |E_{2,q}(\mathbf{0}, \bar{R})|, \\ \beta &= \max\{\alpha, \gamma_q\}, \end{aligned}$$

where the maximum for α is taken over all polynomial R on r variables such that $\bar{R}$ is not constant modulo q. Lemma 2.7.3(ii) implies that $\alpha < 1$.

Consider $(\mathbf{g}_1, \ldots, \mathbf{g}_k) \in J(M_1, \ldots, M_k; \mathbf{g})$ and $i \in [k]$. If $\mathbf{g}_i \neq \mathbf{0}$, then according to Lemma 2.7.3(i), $|E_{2,q}(\mathbf{g}_i, \bar{Q}_i)| \leq \gamma_q$. Assume that $\mathbf{g}_i = \mathbf{0}$. Let $\hat{M}_i$ and R_i be as in Lemma 2.7.6. Then $E_{2,q}(\mathbf{0}, \bar{Q}_i)$ is either 0 or $E_{2,q}(\mathbf{0}, \bar{R}_i)$. Therefore $|E_{2,q}(\mathbf{0}, \bar{Q}_i)| \leq \alpha$, by the definition of α. The proof of the theorem now follows along the lines of the proof of Theorem 2.7.2.

To see the furthermore part notice that for every polynomial R, if $\bar{R}$ is not constant modulo q and the coefficients of R modulo q are in $\mathbb{Z}_q^* \cup \{0\}$, then $t\bar{R}$ is not constant modulo q, for every $t \in [q-1]$. □

Using Lemma 2.2.1 and this theorem we can bound the correlation between parity and polynomials which have a matrix with nice partition, as described in the theorem. As Example 3 in section 2.6 shows, if we want to get the bound for every $t \in [q-1]$, then the additional requirement on the coefficients of each Q_i is necessary. Although

$\beta < 1$, for q odd, it can get arbitrarily close to 1, if r grows with n. However, if r is small enough, certainly if r is a constant independent of n, Theorem 2.7.5 gives exponentially small upper bounds. Also, it is not essential for our argument that for every $i \in [k]$, $\mathrm{rk}_2(M_i) \leq r$, $\bar{Q}_i$ is not constant modulo q, and the coefficients of Q_i are in $\mathbb{Z}_q^* \cup \{0\}$ – it is enough for getting small upper bounds that a large number of $i \in [k]$ have this property. Of course, we also need that the size of $J(M_1, \ldots, M_k; \mathbf{g})$ is small. In particular, this theorem applies to the most general case of the example described before Theorem 2.7.1, provided that there are many sets in the partition of the variables, and each Q_i has the properties required by the theorem.

2.8 More general framework

In section 2.4 we saw how to represent the evaluation of a given polynomial modulo q using a linear transformation over $\mathbb{Z}_2$ and then in section 2.7 we used properties of this linear transformation to bound exponential sums. This section generalizes this approach and shows that this generalization includes the result of Cai, Green, and Thierauf for symmetric polynomials.

Definition 2.8.1. *For $\mathbf{g} \in \{0,1\}^n$, $A \in \mathbb{N}^{m \times n}$, and $\mathbf{b} \in \mathbb{N}^m$, define*

$$\kappa(\mathbf{g}, A, \mathbf{b}) = \sum_{\mathbf{x} \in \{0,1\}^n : A\mathbf{x} = \mathbf{b}} (-1)^{\mathbf{g}^T \mathbf{x}}.$$

For $\mathbf{z} = (z_1, \ldots, z_m)$, define

$$T(\mathbf{g}, A, \mathbf{z}) = \sum_{\mathbf{b} \in I_A} \kappa(\mathbf{g}, A, \mathbf{b}) z_1^{b_1} \cdots z_m^{b_m},$$

where $I_A = \{\mathbf{b} \in \mathbb{Z}^m : \mathbf{0} \leq \mathbf{b} \leq A\mathbf{1}\}$.

The definition of κ includes as a special case the definition of the Krawtchouk polynomials [Sze39]. To see this notice that $\kappa(\mathbf{g}, \mathbf{1}_{1 \times n}, k) = K_k^{(n)}(|\mathbf{g}|)$, where $K_k^{(n)}(l)$ is the k-th Krawtchouk polynomial, i.e. $K_k^{(n)}(l) = \sum_{i=0}^{k} (-1)^i \binom{l}{i} \binom{n-l}{k-i}$ which is the coefficient of y^k of the polynomial $(1-y)^l (1+y)^{n-l}$.

In Definition 2.8.1 $\kappa(\mathbf{g}, A, \mathbf{b})$ is not necessarily a polynomial in $|\mathbf{g}|$, except in special cases, but it gives the coefficient of the monomial $z_1^{b_1} \cdots z_m^{b_m}$ in $T(\mathbf{g}, A, \mathbf{z})$, considered as a polynomial in $z_1, \ldots, z_m$, which can be written in the following form.

$$T(\mathbf{g}, A, \mathbf{z}) = \prod_{j \in [n]: g_j = 1} \left(1 - \prod_{i \in [m]} z_i^{a_{ij}}\right) \prod_{j \in [n]: g_j = 0} \left(1 + \prod_{i \in [m]} z_i^{a_{ij}}\right). \tag{2.7}$$

This expression can be verified by expanding its right side and then grouping the terms in $z_1, \ldots, z_m$ of the same form together. Thus, in some sense the functions $\kappa(\mathbf{g}, A, \mathbf{b})$ are analogues of the Krawtchouk polynomials in a more general setting.

We have the following general expression for $E_{2,q}(\mathbf{g}, f)$.

Lemma 2.8.1. *Let $q \in \mathbb{N}^+$, $\mathbf{g} \in \{0,1\}^n$, and $f : \{0,1\}^n \to \mathbb{Z}$. Assume that there exists $r \in \mathbb{N}^+$, $A \in \mathbb{Z}^{m \times n}$, and $h : \mathbb{Z}^m \to \mathbb{Z}$ such that for every $\mathbf{x} \in \{0,1\}^n$ and $\mathbf{z} \in \mathbb{Z}^m$, if $\mathbf{z} \equiv_r A\mathbf{x}$, then $f(\mathbf{x}) \equiv_q h(\mathbf{z})$. Then*

$$E_{2,q}(\mathbf{g}, f) = 2^{-n} \sum_{\mathbf{y} \in [r]^m} T(\mathbf{g}, A, \omega_r^{\mathbf{y}}) D_{r,q}(\mathbf{y}, h).$$

Proof. For $\mathbf{z} \in [r]^m$, define

$$R_r(\mathbf{g}, A, \mathbf{z}) = \sum_{\mathbf{b} \in I_A : \mathbf{b} \equiv_r \mathbf{z}} \kappa(\mathbf{g}, A, \mathbf{b}).$$

Let us see that

$$R_r(\mathbf{g}, A, \mathbf{z}) = \widehat{\sum_{\mathbf{y} \in [r]^m}} \bar{\omega}_r^{\mathbf{y}^T \mathbf{z}} T(\mathbf{g}, A, \omega_r^{\mathbf{y}}), \tag{2.8}$$

Indeed

$$\begin{aligned} R_r(\mathbf{g}, A, \mathbf{z}) &= \sum_{\mathbf{b} \in I_A : \mathbf{b} \equiv_r \mathbf{z}} \kappa(\mathbf{g}, A, \mathbf{b}) = \sum_{\mathbf{b} \in I_A} \left(\widehat{\sum_{\mathbf{y} \in [r]^m}} \omega_r^{\mathbf{y}^T (\mathbf{b} - \mathbf{z})} \right) \kappa(\mathbf{g}, A, \mathbf{b}) \\ &= \widehat{\sum_{\mathbf{y} \in [r]^m}} \bar{\omega}_r^{\mathbf{y}^T \mathbf{z}} \sum_{\mathbf{b} \in I_A} \omega_r^{\mathbf{y}^T \mathbf{b}} \kappa(\mathbf{g}, A, \mathbf{b}) \\ &= \widehat{\sum_{\mathbf{y} \in [r]^m}} \bar{\omega}_r^{\mathbf{y}^T \mathbf{z}} T(\mathbf{g}, A, \omega_r^{\mathbf{y}}), \end{aligned}$$

where in the second equality we use (2.4).

Finally, recall that by definition, $E_{2,q}(\mathbf{g}, f) = \widehat{\sum}_{\mathbf{x}\in\{0,1\}^n}(-1)^{\mathbf{g}^T\mathbf{x}}\omega_q^{f(\mathbf{x})}$, and note that

$$\sum_{\mathbf{x}\in\{0,1\}^n}(-1)^{\mathbf{g}^T\mathbf{x}}\omega_q^{f(\mathbf{x})} = \sum_{\mathbf{z}\in[r]^m}\left(\sum_{\mathbf{x}\in\{0,1\}^n:A\mathbf{x}\equiv_r\mathbf{z}}(-1)^{\mathbf{g}^T\mathbf{x}}\right)\omega_q^{h(\mathbf{z})} = \sum_{\mathbf{z}\in[r]^m}R_r(\mathbf{g}, A, \mathbf{z})\omega_q^{h(\mathbf{z})}$$
$$= \sum_{\mathbf{y}\in[r]^m}T(\mathbf{g}, A, \omega_r^{\mathbf{y}})\left(\widehat{\sum_{\mathbf{z}\in[r]^m}}\bar{\omega}_r^{\mathbf{y}^T\mathbf{z}}\omega_q^{h(\mathbf{z})}\right),$$

where in the third equality we use (2.8). □

As we will see below, this lemma can be used to derive Lemma 2.4.1, which we use to exploit properties of linear transformations over $\mathbb{Z}_2$ when estimating exponential sums and correlation as described in section 2.7. Furthermore, this lemma can be used to derive the bound of Cai, Green and Thierauf [CGT96] for symmetric polynomials. Interestingly, the lemma yields these two results by working with different parts of the expression which appears in it. To obtain the results in section 2.7 we estimate $D_{r,q}$, but we set things up so that T has only one possible nonzero value, and we just have to argue about when is T nonzero. See the end of this section on how Lemma 2.4.1 is derived from Lemma 2.8.1. To obtain the bounds of [CGT96], we estimate T, and use only a trivial bound on $D_{r,q}$, namely that $|D_{r,q}| \leq 1$. The key to derive the bounds of [CGT96] from Lemma 2.8.1 is to show that for symmetric functions there is a matrix A with only one row that has the desired properties. As we will see in Corollary 2.8.2, whenever we can guarantee that for an odd r there exists A with the desired properties which also has a small number of rows, we can obtain exponentially small bounds on the exponential sum. The details of the argument for symmetric polynomials are in section 2.8.1.

To apply Lemma 2.8.1, one has to argue that suitable A and h exist and then estimate the corresponding expressions. This still requires a lot of work. However, we immediately get the following simple corollary.

Corollary 2.8.2. *If $r \in \mathbb{N}^+$ odd and $A \in \mathbb{Z}^{m\times n}$ as in Lemma 2.8.1 exist, then*

$$|E_{2,q}(\mathbf{1}, f)| \leq r^m\gamma_r^n.$$

Proof. By (2.7), $|2^{-n}T(\mathbf{1}, A, \omega_r^{\mathbf{y}})| = \prod_{j=1}^{n} \left|\sin \frac{\pi}{r}\mathbf{y}^T\mathbf{a}_j\right|$, where $\mathbf{a}_j$ is the j-th column of A, using $|1-\omega_r^c| = |\omega_r^{c/2}(\omega_r^{-c/2} - \omega_r^{c/2})| = 2|\sin\frac{\pi c}{r}|$. The statement follows, because $|D_{r,q}| \leq 1$ always holds by the triangle inequality. □

In the case of symmetric polynomials, it can be shown that taking A to consist of a single row of all 1's, and letting $h(z)$ to be the value of the polynomial on inputs containing z 1's, we get the desired properties for an odd r. Applying Corollary 2.8.2 with $m = 1$ and the appropriate r yields the bounds of [CGT96] for symmetric polynomials.

Lemma 2.4.1 can be obtained as a corollary of Lemma 2.8.1, by taking $r = 2$, A to be a matrix for Q modulo q, and $h(\mathbf{z}) = \mathbf{1}^T(-1)^{\mathbf{z}}$. Let $\mathbf{y} \in \{0,1\}^m$. By (2.7)

$$2^{-n}T(\mathbf{g}, M, (-1)^{\mathbf{y}}) = \begin{cases} 1, & \mathbf{y} \in K(M, \mathbf{g}), \\ 0, & \text{otherwise.} \end{cases}$$

Finally,

$$\begin{aligned} D_{2,q}(\mathbf{z}, th) = \widehat{\sum_{\mathbf{z}\in\{0,1\}^m}} (-1)^{\mathbf{z}^T\mathbf{y}}\omega_q^{th(\mathbf{z})} &= 2^{-m} \sum_{\mathbf{z}\in\{0,1\}^m} (-1)^{\sum_i z_i y_i}\omega_q^{t\sum_i (-1)^{z_i}} \\ &= 2^{-m}(\omega_q^t - \bar{\omega}_q^t)^{|\mathbf{y}|}(\omega_q^t + \bar{\omega}_q^t)^{m-|\mathbf{y}|}. \end{aligned}$$

So far all our expressions have been precise, except the estimate given in Corollary 2.8.2, and we have exact representations of the exponential sum.

2.8.1 Bounds for symmetric polynomials

In this section we use Lemma 2.8.1 to derive the bounds of [CGT96] for symmetric polynomials. The proof builds on ideas of [CGT96], and as in [CGT96], we need the period of binomial coefficients modulo q, which is given by the following theorem of Zabek [Zab56].

Theorem 2.8.3 (Zabek, [Zab56]). *Let $q, d \in \mathbb{N}^+$. Let $q = p_1^{e_1}\cdots p_s^{e_s}$ be the prime factorization of q and $l_i = \lfloor \log_{p_i} d \rfloor$, $i \in [s]$. Then the period of $h(n) = \binom{n}{d} \bmod q$ is $r = qp_1^{l_1}\cdots p_s^{l_s}$.*

Lemma 2.8.4. *Let $q, d \in \mathbb{N}^+$ and $P \in \mathcal{P}^{sym}(\mathbb{Z}, n, d)$. Let $q = p_1^{e_1}\cdots p_s^{e_s}$ be the prime factorization of q and $l_i = \lfloor \log_{p_i} d \rfloor$, $i \in [s]$. Then, for every $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$, if $\mathbf{1}^T\mathbf{x} \equiv_r \mathbf{1}^T\mathbf{y}$, then $P(\mathbf{x}) \equiv_q P(\mathbf{y})$, where $r = qp_1^{l_1}\cdots p_s^{l_s}$.*

Proof. Since $P(\mathbf{x})$ is symmetric of degree at most d we have that there exist $a_0, \ldots, a_d \in \mathbb{Z}$ such that for every $\mathbf{x} \in \{0,1\}^n$,

$$P(\mathbf{x}) = \sum_{i=0}^{d} a_i \binom{|\mathbf{x}|}{i}.$$

By Theorem 2.8.3 the period of $h_i(k) = \binom{k}{i} \bmod q$ is $r_i = q p_1^{\lfloor \log_{p_1} i \rfloor} \cdots p_s^{\lfloor \log_{p_s} i \rfloor}$. Therefore if $h(k) = \sum_{i=0}^{d} a_i \binom{k}{i}$ and r' is the least common multiple of $\{r_0, \ldots, r_d\}$ then for every $x, y \in \mathbb{N}$, if $x \equiv_{r'} y$, then $h(x) \equiv_q h(y)$. The lemma follows, because $P(\mathbf{x}) = h(\mathbf{1}^T \mathbf{x})$ and $r = r'$. □

Theorem 2.8.5 (Cai–Green–Thierauf, [CGT96]). *Let $q, d \in \mathbb{N}^+$ and s be the number of distinct prime divisors of q. For every $P \in \mathcal{P}^{sym}(\mathbb{Z}, n, d)$,*

$$|E_{2,q}(\mathbf{1}, P)| \leq q d^s 2^{-\Omega(n/qd^s)}$$

Proof. Let $q = p_1^{e_1} \cdots p_s^{e_s}$ be the prime factorization of q and $l_i = \lfloor \log_{p_i} d \rfloor$, $i \in [s]$. Let $A = \mathbf{1}_{1 \times n}$ and $r = q p_1^{l_1} \cdots p_s^{l_s}$. For $z \in \mathbb{Z}$, let $h(z) = P(\mathbf{x}_z)$, where $\mathbf{x}_z$ is an arbitrary vector in $\{0,1\}^n$ with weight $z \bmod r$. By Lemma 2.8.4 for every $\mathbf{x} \in \{0,1\}^n$ and $z \in \mathbb{Z}$, if $z \equiv_r A\mathbf{x}$, then $P(\mathbf{x}) \equiv_q h(z)$. Thus, we can use Corollary 2.8.2 with $m = 1$. Since $r \leq q d^s$ and $\gamma_r^n = 2^{-\Omega(n/r)}$, the theorem follows. □

2.8.2 Generalized symmetric polynomials

In this section we generalize the result of Cai, Green, and Thierauf to functions on not too many generalized symmetric polynomials. Before we explain this result in more details we will take a detour into symmetric polynomials over $\mathbb{Z}_q$.

For a prime p and $n \in \mathbb{N}^+$, let $\mathrm{ord}_p n$ be the exponent of the largest power of p dividing n. It is known that $\mathrm{ord}_p n! = (n - \sigma_p(n))/(p-1)$, where $\sigma_p(m)$ is the sum of digits in the p-ary expansion of m. This implies that $\mathrm{ord}_p n! = O(n)$, where the constant in O depends on p.

For $k \in \mathbb{N}^+$ and $\mathbf{x} = (x_1, \ldots, x_n)$ define the k-th elementary symmetric polynomial

$$\sigma_k(\mathbf{x}) = \sum_{1 \leq i_1 < \ldots < i_k \leq n} x_{i_1} \cdots x_{i_k},$$

and the k-th power-sum polynomial

$$\pi_k(\mathbf{x}) = \sum_{i \in [n]} x_i^k.$$

By the Fundamental Theorem for Symmetric Polynomials [LN97] every symmetric polynomial over a ring can be expressed in terms of the elementary symmetric polynomials.

Theorem 2.8.6 (Fundamental Theorem for Symmetric Polynomials). *Let R be a ring and $P \in \mathcal{P}^{sym}(R, n, d)$. Then there exists a unique $Q \in \mathcal{P}(R, d)$ such that $P(\mathbf{x}) = Q(\sigma_1(\mathbf{x}), \ldots, \sigma_d(\mathbf{x}))$. Furthermore, if $\prod_{j \in S} y_j^{d_j}$ is a term of $Q(\mathbf{y})$, for some $S \subseteq [d]$, then $\sum_{j \in S} j d_j \leq d$. In particular, Q has degree at most d.*

By the Fundamental Theorem for Symmetric Polynomials, and since each π_d is symmetric, we have that for every $d \in \mathbb{N}^+$, there exist $T_d \in \mathcal{P}(\mathbb{Z}, n, d)$ such that for $\mathbf{x} \in \mathbb{Z}^n$

$$\pi_d(\mathbf{x}) = T_d(\sigma_1(\mathbf{x}), \ldots, \sigma_d(\mathbf{x})).$$

Since T_d has integer coefficients, we have that for every $q \in \mathbb{N}^+$ and $\mathbf{x} \in \mathbb{Z}^n$

$$\pi_d(\mathbf{x}) \equiv_q T_d(\sigma_1(\mathbf{x}), \ldots, \sigma_d(\mathbf{x})). \tag{2.9}$$

The Newton–Girard formulas [LN97] state the converse of Theorem 2.8.6 over $\mathbb{Q}$, i.e. that over $\mathbb{Q}$ the Fundamental Theorem for Symmetric Polynomials holds with the power-sum polynomials instead of the elementary symmetric polynomials.

Theorem 2.8.7 (Newton–Girard). *For every $d \in \mathbb{N}^+$*

$$d\sigma_d(\mathbf{x}) + \sum_{k \in [d]} (-1)^k \pi_k(\mathbf{x}) \sigma_{d-k}(\mathbf{x}) = 0.$$

In particular, there exists $N_d \in \mathcal{P}(\mathbb{Q}, d, d)$ such that $d! N_d \in \mathcal{P}(\mathbb{Z}, d, d)$ and

$$\sigma_d(\mathbf{x}) = N_d(\pi_1(\mathbf{x}), \ldots, \pi_d(\mathbf{x})).$$

Since N_d has rational coefficients we cannot claim, like in (2.9), that Theorem 2.8.7 holds modulo any $q \in \mathbb{N}^+$. Part of this section proves that we can preserve the possibility

of expressing σ_d in terms of $\pi_1, \ldots, \pi_d$ in the following sense: for $q, d \in \mathbb{N}^+$ there exists $r \in \mathbb{N}^+$ and $\tau : \mathbb{Z}_q \to \mathbb{Z}_r$ such that for every $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$, if $\pi_k(\tau(\mathbf{x})) \equiv_r \pi_k(\tau(\mathbf{y}))$ for every $k \in [d]$, then $\sigma_d(\mathbf{x}) \equiv_q \sigma_d(\mathbf{y})$. r will have the following form: if $p_1^{e_1} \cdots p_s^{e_s}$ is the prime decomposition of q, then $r = q p_1^{l_1} \cdots p_s^{l_s}$, where the exponent l_i depends on d and p_i.

The statement in the previous paragraph can be rephrased like this: on integer inputs, the value of $\pi_k(\tau(\mathbf{x}))$ modulo r, $k \in [d]$, determines the value of $\sigma_d(\mathbf{x})$ modulo q, for r possibly different than q. The statement is different for the other direction, where, by (2.9), we can take $r = q$. More informally, to express the elementary symmetric polynomials in terms of the power-sum polynomials we have to "increase the precision". This expression makes more sense in the case where q is a prime power p^e, then $r = p^{e+l}$, so the "precision" is increased from e to $e + l$. The situation is similar to approximate computation of a given continuous real valued function f, where for given ε, given the arguments to f with precision δ depending on ε, determines the value of f with precision ε. The important question here is what is the additional precision that we need. Following a simple argument based on the "continuity" intuition, we will prove (Theorem 2.8.11) that $l_i = \mathrm{ord}_{p_i} d!$, i.e. an additional precision of $O(d)$ is enough. A more complicated argument due to Voloch ([Vol05], Theorem 2.8.12) shows that in some cases actually additional precision of $O(\log d)$ is enough. We have an application (Theorem 2.8.13) of those results to bounding exponential sums involving generalized symmetric polynomials where it is obvious why do we want as small precision as possible.

First, we give an argument which implies that precision $O(d)$ is enough. The following lemma shows that if we have a polynomial P with rational coefficients which on integer inputs is integer, e.g. the polynomials N_d in Theorem 2.8.7, then the value of P modulo q, is determined by the value of its inputs modulo r, i.e. with increased precision. The equivalent statement over the reals, i.e. that to compute P with some precision ε on real inputs it is enough to have the inputs with some precision δ, follows because P being a polynomial is continuous. In our case we need that P is continuous over the p-adics (for p prime), because it is a polynomial with rational coefficients. Notice that the added precision is $\mathrm{ord}_{p_i} m$ where m is the least common denominator of the coefficients of P.

The proof of Lemma 2.8.9 is a rephrasing of the argument using the continuity of P over the p-adics into a number theory language.

In the following we use the Chinese Remainder Theorem.

Theorem 2.8.8 (Chinese Remainder Theorem). *Let $q_1, \ldots, q_s \in \mathbb{N}^+$ be relatively prime. Then for every $a_1, \ldots, a_s \in \mathbb{Z}$, there exists a unique $a \in [q_1 q_2 \cdots q_s]$ such that $a \equiv_{q_i} a_i$ for every $i \in [s]$.*

Lemma 2.8.9. *Let $q \in \mathbb{N}^+$ and $q = p_1^{e_1} \cdots p_s^{e_s}$ be the prime factorization of q. Let $P \in \mathcal{P}(\mathbb{Q}, n)$ be such that for every $\mathbf{x} \in \mathbb{Z}^n$, $P(\mathbf{x}) \in \mathbb{Z}$. Let $m \in \mathbb{Z}$ be such that $mP \in \mathcal{P}(\mathbb{Z}, n)$. Then for every $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$, if $\mathbf{x} \equiv_r \mathbf{y}$, then $P(\mathbf{x}) \equiv_q P(\mathbf{y})$, where $r = q p_1^{\mathrm{ord}_{p_1} m} \cdots p_s^{\mathrm{ord}_{p_s} m}$.*

Proof. We will need the following simple observation.

Observation 2.8.10. *Let $r \in \mathbb{N}^+$. Then for every $P \in \mathcal{P}(\mathbb{Z}, n)$ and $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$, if $\mathbf{x} \equiv_r \mathbf{y}$, then $P(\mathbf{x}) \equiv_r P(\mathbf{y})$.*

First assume that q is a prime power $q = p^e$. Let $l = \mathrm{ord}_p m$. Then $m = p^l M$, where M and p are relatively prime. Let $Q(\mathbf{x}) = M' m P(\mathbf{x})$, where M' is a multiplicative inverse of M modulo p^{e+l}. By Observation 2.8.10 applied to $Q(\mathbf{x})$, if $\mathbf{x} \equiv_{p^{e+l}} \mathbf{y}$, then

$$p^l P(\mathbf{x}) \equiv_{p^{e+l}} Q(\mathbf{x}) \equiv_{p^{e+l}} Q(\mathbf{y}) \equiv_{p^{e+l}} p^l P(\mathbf{y})$$

and therefore

$$P(\mathbf{x}) \equiv_{p^e} P(\mathbf{y}).$$

Now let $q = p_1^{e_1} \ldots p_s^{e_s}$. According to the above, for every $i \in [s]$, if $\mathbf{x} \equiv_{p^{e_i + \mathrm{ord}_{p_i} m}} \mathbf{y}$, then

$$P(\mathbf{x}) \equiv_{p_i^{e_i}} P(\mathbf{y}).$$

Finally, use Chinese Remainder Theorem to finish the proof. □

The next theorem shows that to express σ_d modulo q in terms of π_k, $k \in [d]$, additional precision $O(d)$ is enough. This essentially follows because N_d fulfills the conditions of Lemma 2.8.9, and so additional precision of $\mathrm{ord}_{p_i} m$ is enough, where m is the least common denominator of the coefficients of N_d. Since by Theorem 2.8.7 $m = d!$, we have that an additional precision of $\mathrm{ord}_{p_i} d! = O(d)$ is enough.

Theorem 2.8.11. *Let $q, d \in \mathbb{N}^+$ and $P \in \mathcal{P}^{sym}(\mathbb{Z}, n, d)$. Then for every $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$, if $\pi_k(\mathbf{x}) \equiv_r \pi_k(\mathbf{y})$ for every $k \in [d]$, then $P(\mathbf{x}) \equiv_q P(\mathbf{y})$, where $r = qp_1^{\mathrm{ord}_{p_1} d!} \cdots p_s^{\mathrm{ord}_{p_s} d!}$ $(\leq q(p_1 \cdots p_s)^d)$.*

Proof. According to the Newton-Girard formulas (Theorem 2.8.7), for every $k \in \mathbb{N}^+$ $k!N_k \in \mathcal{P}(\mathbb{Z}, k)$ and for every $\mathbf{x} \in \mathbb{Z}^n$, $\sigma_k(\mathbf{x}) = N_k(\pi_1(\mathbf{x}), \ldots, \pi_k(\mathbf{x}))$. Hence, by Lemma 2.8.9, for every $k \in \mathbb{N}^+$, $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$, if $\pi_l(\mathbf{x}) \equiv_{r_k} \pi_l(\mathbf{y})$ for every $l \in [k]$, then $\sigma_k(\mathbf{x}) \equiv_q \sigma_k(\mathbf{y})$, where $r_k = qp_1^{\mathrm{ord}_{p_1} k!} \cdots p_s^{\mathrm{ord}_{p_s} k!}$.

Since $P \in \mathcal{P}^{sym}(\mathbb{Z}, n, d)$, by the Fundamental Theorem for Symmetric Polynomials there exists $Q \in \mathcal{P}(\mathbb{Z}, d)$ such that $P(\mathbf{x}) = Q(\sigma_1(\mathbf{x}), \ldots, \sigma_d(\mathbf{x}))$. Consider $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$ such that $\pi_k(\mathbf{x}) \equiv_r \pi_k(\mathbf{y})$ for every $k \in [d]$. Take $k \in [d]$. Since $r_k | r$, $\pi_l(\mathbf{x}) \equiv_{r_k} \pi_l(\mathbf{y})$ for every $l \in [k]$, and therefore $\sigma_k(\mathbf{x}) \equiv_q \sigma_k(\mathbf{y})$. Using Observation 2.8.10 this implies that $P(\mathbf{x}) \equiv_q P(\mathbf{y})$. □

The following theorem is due to Voloch [Vol05] and is an improvement over Theorem 2.8.11, because it says that if q is odd and does not have prime powers in its prime decomposition, then to express σ_d modulo q in terms of π_k, an additional precision of $O(\log d)$ is enough. It is interesting whether an analogous statement holds for any q.

For p prime and $l \in \mathbb{N}^+$, define $\tau_{p^l} : \mathbb{Z}_p \to \mathbb{Z}_{p^l}$ in the following way. If $x = 0$, then let $\tau_{p^l}(x) = 0$. Otherwise, $x \in \mathbb{Z}_p^*$ and then let o be the order of x, i.e. the smallest k such that $x^k = 1$. Since $\mathbb{Z}_{p^l}^*$ is cyclic of order $p^{l-1}(p-1)$ it has an elements of order o. Then let $\tau_{p^l}(x)$ be this element of $\mathbb{Z}_{p^l}^*$.

Let $q \in \mathbb{N}^+$ and $p_1 \ldots p_s$ be its prime decomposition. Let $l_1, \ldots, l_k \in \mathbb{N}^+$ and $r = p_1^{l_1} \cdots p_s^{l_s}$. Define $\tau_r : \mathbb{Z}_q \to \mathbb{Z}_r$ by applying the Chinese Remainder Theorem in the following way. Given $x \in \mathbb{Z}_q$ consider the unique $(x_1, \ldots, x_s) \in \mathbb{Z}_{p_1^{l_1}} \times \cdots \times \mathbb{Z}_{p_s^{l_s}}$ that corresponds to it. Let $y_i = \tau_{p_i^{l_i}}(x_i) \in \mathbb{Z}_{p_i^{l_i}}$, $i \in [s]$. Finally, let $\tau_r(x)$ be the unique element of $\mathbb{Z}_r$ which corresponds to $(y_1, \ldots, y_s)$.

We also think of τ_r as a function from $\mathbb{Z}$ to $\mathbb{Z}$ where given an integer first we reduce it modulo q, then we apply τ_r and interpret the resulting element of $\mathbb{Z}_r$ as an element of $[r]$.

Theorem 2.8.12 (Voloch, [Vol05]). *Let $q, d \in \mathbb{N}^+$, q odd, $p_1 \cdots p_s$ be the prime decomposition of q, and $P \in \mathcal{P}^{sym}(\mathbb{Z}, n, d)$. Then for every $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$, if $\pi_k(\tau_r(\mathbf{x})) \equiv_r \pi_k(\tau_r(\mathbf{y}))$ for every $k \in [d]$, then $P(\mathbf{x}) \equiv_q P(\mathbf{y})$, where $r = qp_1^{\lfloor \log_{p_1} d \rfloor} \cdots p_s^{\lfloor \log_{p_s} d \rfloor}$ $(\leq qd^s)$.*

Let us draw an analogy of the above discussion with Lucas's Theorem, and more precisely with its extension due to Zabek. Since for $k \in \mathbb{N}^+$ and $\mathbf{x} \in \{0,1\}^n$, $\pi_k(\mathbf{x}) = \pi_1(\mathbf{x})$, Zabek's Theorem (see Lemma 2.8.4) essentially gives the statement of Theorem 2.8.12 for any q, except that it restricts the inputs to the symmetric polynomial to be in $\{0,1\}$. Thus, if we prove an analogue of Theorem 2.8.12 for any $q \in \mathbb{N}^+$ (not just products of primes), then we will have a generalization of Zabek's Theorem.

Definition 2.8.2. *Let $q, d \in \mathbb{N}^+$ with $p_1^{e_1} \cdots p_s^{e_s}$ the prime factorization of q. Define $\lambda_i \in \mathbb{N}^+$, $i \in [s]$, to be the smallest numbers for which there exists $\tau : \mathbb{Z}_q \to \mathbb{Z}_r$ such that for every $P \in \mathcal{P}^{sym}(\mathbb{Z}, n, d)$ and $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^n$, if $\pi_k(\tau(\mathbf{x})) \equiv_r \pi_k(\tau(\mathbf{y}))$ for every $k \in [d]$, then $P(\mathbf{x}) \equiv_q P(\mathbf{y})$, where $r = qp_1^{\lambda_1} \cdots p_s^{\lambda_s}$.*

By Theorem 2.8.11, we have that $\lambda_i = O(d)$. By Theorem 2.8.12, $\lambda_i = O(\log d)$, if q is odd and does not have prime powers in its prime decomposition.

The following theorem is an application of the above considerations to generalized symmetric polynomials. A *generalized symmetric polynomial* T is specified by a symmetric polynomial P and a vector $\mathbf{a}$ such that $T(\mathbf{x}) = P(\mathbf{a} \cdot \mathbf{x})$. T itself is not a symmetric polynomial, but it is very close to being one. Of course a symmetric polynomial is a special case, if we take $\mathbf{a} = \mathbf{1}$. The result of Cai, Green, and Thierauf (Theorem 2.8.5) shows that if P is a symmetric polynomial of degree d and q is odd, then

$$|E_{2,q}(\mathbf{1}, P)|\, d^s \leq 2^{-\Omega(n/d^s)}.$$

If $T(\mathbf{x}) = P(\mathbf{a} \cdot \mathbf{x})$ is a generalized symmetric polynomial, then the same result on T follows like this. Let a be the most popular modulo q amongst the a_i's, i.e. there are at least n/q i's such that $a_i \equiv_q a$. Use an averaging argument to fix all x_i's such that $a_i \not\equiv_q a$ without decreasing $|E_{2,q}(\mathbf{1}, T)|$. The resulting polynomial T' has at least n/q variables and is symmetric. Thus Theorem 2.8.5 applies to generalized symmetric polynomials.

Theorem 2.8.13 extends this to any function of up to $k = o(n)$ generalized symmetric polynomials. Notice that in this case the argument with fixing to the most popular a_i does not work, unless $k = \varepsilon \log n$ with $\varepsilon \in [0, 1)$ depending on q.

The significance of requiring as small a precision, i.e. as small an r as possible, when expressing σ_d modulo q in terms of π_k modulo r, is the following. Since the bound implied by Theorem 2.8.13 is $2^{O(kd \log r) - \Omega(n/r)}$ (using that $\gamma^n = 2^{-\Omega(n/r)}$) to get an exponentially small bound we need $r = o(n)$. By Theorem 2.8.11 we can take $r = O(q^d)$, which means that we can only allow $d = \varepsilon \log n$, for $\varepsilon \in [0, 1)$ depending on q. Note that in this case the restriction on the degree is the same as the restriction in Bourgain's result (Theorem 2.3.3) and we cannot even get the bound of Theorem 2.8.5 which is interesting for $d = \text{polylog}(n)$ (and even n^ε for some $\varepsilon \in [0, 1)$ depending on q). If on the other hand Theorem 2.8.12 holds for any q (not only products of primes), then $r = O(d^s)$ and we extend the result of Theorem 2.8.5 by allowing a function of up to $k = n^\delta$ generalized symmetric polynomials of degree n^ε, where $\varepsilon, \delta \in [0, 1)$ depend on q. In any case since Theorem 2.8.12 holds for q odd and a product of distinct primes, Theorem 2.8.13 certainly extends Theorem 2.8.5 to this case.

In Theorem 2.8.13, g can be an arbitrary function which depends on its inputs only modulo q. A simple example of such function is any polynomial with integer coefficients (see Observation 2.8.10 in the proof of Lemma 2.8.9). If q is prime, then all functions in $\mathbb{Z}_q$ are polynomials, but this is not true, if q is composite (see the next section for more details).

Theorem 2.8.13. *Let $q \in \mathbb{N}^+$ be odd and $q = p_1^{e_1} \cdots p_s^{e_s}$ be the prime factorization of q. Let $g : \mathbb{Z}^k \to \mathbb{Z}$ be such that for every $\mathbf{x}, \mathbf{y} \in \mathbb{Z}^k$, if $\mathbf{x} \equiv_q \mathbf{y}$, then $g(\mathbf{x}) \equiv_q g(\mathbf{y})$. Let $P_1, \ldots, P_k \in \mathcal{P}^{sym}(\mathbb{Z}, n, d)$, $\mathbf{a}_1, \ldots, \mathbf{a}_k \in \mathbb{Z}^n$, and $P(\mathbf{x}) = g(P_1(\mathbf{a}_1 \cdot \mathbf{x}), \ldots, P_k(\mathbf{a}_k \cdot \mathbf{x}))$. Then for $r = q p_1^{\lambda_1} \cdots p_s^{\lambda_s}$, where the λ_i's are as in Definition 2.8.2,*

$$|E_{2,q}(\mathbf{1}, P)| \leq r^{kd} \gamma_r^n.$$

Proof. First notice that for $\mathbf{x} \in \{0, 1\}^n$ and $\mathbf{b} \in \mathbb{Z}^n$, $\pi_l(\mathbf{b} \cdot \mathbf{x}) = \sum_i b_i^l x_i$. Define $A_i \in \mathbb{Z}^{d \times n}$, $i \in [k]$, so that its l-th row is $(\tau(a_{i1})^l, \ldots, \tau(a_{in})^l)$, where τ is as in Definition 2.8.2 and

$\mathbf{a}_i = (a_{i1}, \ldots, a_{in})$. From Definition 2.8.2, follows that there exist $h_1, \ldots, h_m : \mathbb{Z}^d \to \mathbb{Z}$ such that for every $\mathbf{x} \in \{0,1\}^n$ and $\mathbf{y} \in \mathbb{Z}^m$, if $\mathbf{y} \equiv_r A_i\mathbf{x}$, then $P_i(\mathbf{a}_i \cdot \mathbf{x}) \equiv_q h_i(\mathbf{y})$. Define $A \in \mathbb{Z}^{(dk)\times n}$ to be the rows of A_1, followed by the rows A_2, and so on until A_k. For $y_{ij} \in \mathbb{Z}$, $i \in [k]$ and $j \in [d]$, define $h(\mathbf{y}) = g(h_1(\mathbf{y}_1), \ldots, h_k(\mathbf{y}_k))$, where $\mathbf{y}_i = (y_{i1}, \ldots, y_{id})$. Finally notice that the requirements of Corollary 2.8.2, with $m = dk$, are fulfilled and the bound in the theorem follows. □

2.8.2.1 Functions modulo a composite number

In this section we prove the claim made before Theorem 2.8.13 that not every function over $\mathbb{Z}_q$ is a polynomial for q composite. More precisely, we will prove that there is no polynomial $P \in \mathcal{P}(\mathbb{Z}, 1)$ such that

$$P(x) \not\equiv_q 0 \text{ iff } x \equiv_q 0. \tag{2.10}$$

Assume, for the sake of contradiction, that such a polynomial $P(x) = \sum_{i=0}^{d} a_i x^i$ exists. Then

$$a_0 = P(0) \not\equiv_q 0. \tag{2.11}$$

First, assume that q is not a prime power and let $p_1^{e_1} \ldots p_s^{e_s}$, $k \geq 2$, be the prime decomposition of q. Since $p_1^{e_1} \not\equiv_q 0$, by (2.10) we have that $P(p_1^{e_1}) \equiv_q 0$. Hence, because $p_1^{e_1} | q$, $P(p_1^{e_1}) \equiv_{p_1^{e_1}} 0$. Therefore $a_0 \equiv_{p_1^{e_1}} 0$. A similar argument proves that $a_0 \equiv_{p_i^{e_i}} 0$ for every $i \in [s]$. By the Chinese Remainder Theorem, $a_0 \equiv_q 0$, which contradicts (2.11).

Now assume that $q = p^e$, for some $e \geq 2$. Then for every $i \geq 1$ we have that $(e-1)i \geq e$ and therefore

$$P(p^{e-1}) = \sum_{i=1}^{n} a_i p^{(e-1)i} + a_0 \equiv_{p^e} a_0$$

By (2.10) and since $p^{e-1} \not\equiv_{p^e} 0$, $P(p^{e-1}) \equiv_{p^e} 0$, which again contradicts (2.11).

Chapter 3

Undirected st-connectivity

In this chapter we flip the coin of our complexity considerations and look at an upper bound on the space complexity of undirected st-connectivity. The approach we take in solving this problem, as suggested by Prof. Ramachandran, is to simulate space-efficiently with a sequential algorithm the parallel algorithm of Chong and Lam. In section 3.2 we fuse ideas from this algorithm with the exploration walks on trees of Koucký to form the abstract base of our space-efficient algorithm. Using this abstract base, which ensures the correctness of the algorithm, in section 3.3 we give a precise description of the algorithm which allows us to reason about its space complexity. This description exhibits clearly the bottlenecks which stand in the way of a space-efficient algorithm and we show a solution against those bottlenecks which results in an $O(\log n \log\log n)$ space algorithm. In section 3.3.1 we use communicating oracles to give an intuition for the transition between the abstract base and the precise description of the algorithm.

3.1 Overview of the parallel algorithm of Chong and Lam

The Chong–Lam (CL) algorithm [CL95] uses a hook and contract approach. This approach to USTCONN can be summarized by the following simple observation. Consider a spanning subgraph H of G, i.e. H has the same vertex set as G and a subset of its edges. Call each connected component of H a *cluster*. For every cluster, choose one of its vertices to be its representative. Form a new graph G' whose vertices are the representatives of the clusters of H and two vertices are connected in G' iff two vertices from the two corresponding clusters of H are connected in G. We can solve USTCONN on G by

solving it recursively on G', because s and t are connected in G iff their representatives are connected in G'. We call the process of obtaining H from G *hooking* and the process of obtaining G' from H and G *contraction*.

One simple example of the above idea is when H contains only one edge of G, e.g. the smallest edge of G in some fixed linear ordering of its edges. Then G' is obtained by collapsing the ends of this edge. This results in an algorithm which has linear number of hookings and contractions.

Another example is obtained, if we can guarantee that after hooking every cluster has at least two vertices, because this way contraction reduces the number of vertices of the graph by a factor of at least two and so we have only $O(\log n)$ hookings and contractions. The advantage of such algorithm is that it has a potential of being efficiently implementable in parallel as long as the hooking and the contraction operations are efficiently implementable in parallel. For example the following hooking operation has this properties. For every vertex u of G put in H the edge $\{u, v\}$, where v is the smallest neighbor of u in some fixed linear ordering of the vertices of G. Call $\{u, v\}$ the *hooking edge* of u. The clusters of H are tree-loops, i.e. connected graphs which have at most one simple cycle. Choosing a hooking edge for every vertex could be trivially performed in parallel $O(\log n)$ time with linear number of processors. The same is true for contracting tree-loops, although the solution is slightly more complicated. This results in an $O(\log^2 n)$ time parallel algorithm for USTCONN.

The clusters resulting from the hooking operation of the CL algorithm are trees, i.e. H is a spanning forest of G, which we call *hooking trees*. The CL algorithm differs from the hook and contract approach as described above, because during contraction not all clusters are contracted. In particular, during the algorithm the vertices of the graph go through three states: active, inactive, and done. Done vertices are non-representative vertices which were part of a contracted hooking tree; once a vertex becomes done it stays done. Only active vertices participate in hooking such that each active vertex chooses one of its edges to be its hooking edge in a manner somewhat similar to the one explained in the previous paragraph. After contraction the vertices whose hooking trees

were not contracted become inactive and their hooking trees are preserved for subsequent contraction. During hooking, the active vertices either form new hooking trees or become part of existing ones.

We postpone the exact details of the hooking operation of the CL algorithm to next sections. For now let us just say that to determine its hooking edge a vertex looks at the degrees and the states of vertices which are at most two edges away; also, some vertices decide not to choose a hooking edge (this essentially is why the resulting clusters are trees and not tree-loops) and we say that they hooked to themselves. Thus, hooking can be done in parallel time $O(\log d)$ with linear number of processors by performing pointer jumping, where d is an upper bound on the degrees of the active vertices, if the current graph is represented by its adjacency lists. Pointer jumping [JáJ92] is a standard primitive for traversing linked lists in parallel algorithms. For a given linked list of length at most d it involves $\lceil \log d \rceil$ jumps of its pointers, where in a jump all pointers of the list are advanced one element forward. Each jump can be performed in constant time given d processors.

Let the total degree of a hooking tree be the sum of the degrees in G of its vertices. During contraction only hooking trees with total degree at most c are contracted, where c is a parameter of the contraction operation. The representative of a hooking tree is the only vertex in it which hooked to itself. For every contracted hooking tree its representative becomes a new active vertex and the rest of its vertices become done and are removed from further consideration. Also all multi-edges between new active vertices are cleaned-up. Finally, the vertices of every uncontracted tree become inactive. The important part of the contraction procedure is checking the degree of a hooking tree. In parallel this could be done in $O(\log c)$ time, where c is the value of the contraction parameter, by using pointer jumping and edge-list plugging [TV85]. Edge-list plugging combines into one list the adjacency lists of the vertices of a hooking tree. It proceeds like this. Every edge of the current graph participates in two adjacency lists, i.e. for every edge there are two entries in two different linked lists describing the graph. We assign a processor to every edge of the graph and for every hooking edge in constant time the processor assigned to

it swaps the pointers for next element of its two entries. It is not hard to see that this way the adjacency list of every hooking tree are combined into one list on which we can perform pointer jumping.

Finally the CL algorithm is given by the following recursive procedure. Here `MaxHook` and `Contract`(c) denote correspondingly a hooking operation and a contracting operation with parameter c.

```
procedure Connect(k)
    MaxHook;
    if k > 0 then
        Contract(2^(2^k));
        Connect(k − 1);
        Connect(k − 1);
    Contract(2^(2^(k+1)));
```

The correctness of the CL algorithm ensures that a call to `Connect`($\lceil \log \log n \rceil$) contracts every connected component of the graph to a single vertex. All the other vertices are organized in a set of rooted trees such that the root of the tree of a vertex u is the vertex to which the connected component of u contracted. These rooted trees are updated after every contraction by making the parent in such a tree of all vertices which become done and which are part of the same hooking tree T to be the representative of T.

The CL algorithm can be simulated trivially with a sequential algorithm using linear space. To this end, we fix an ordering of the edges incident to a vertex and instead of performing the hooking in parallel for all active vertices, we do it sequentially for each of them.

3.2 Abstract components of the space-efficient sequential algorithm

Our space-efficient algorithm is based on the parallel algorithm of Chong and Lam. From this algorithm we extract a set of abstract definitions, given in this section, which allow

us to prove the correctness of our algorithm. The space-efficient algorithm based on these abstract components and a proof of its space bound are given in section 3.3. The proof of the correctness is based entirely on the proof of Chong and Lam of the correctness of their algorithm with the only modification that we change their hooking strategy slightly and use exploration walks on trees instead of pointer-jumping and edge-list plugging. Both of these changes do not influence the correctness.

3.2.1 Graphs and exploration walks on graphs

An undirected multi-graph is a graph with possibly multiple edges between two vertices and such that every edge has a label on each side, where the labels of the edges incident to a vertex v have distinct labels on the side of v. We also have a single self-loop with label 0 at every vertex. Formally we have

Definition 3.2.1. *An* undirected multi-graph G *is a triple* $\langle V, \delta, \mu \rangle$*, where* V *is a set,* $\delta : V \to \mathbb{N}$*, and* $\mu : E \to E$ *is a bijection such that* $\mu(\mu(v,i)) = (v,i)$ *and* $\mu(v,0) = (v,0)$*, where* $E = \{(v,i) : v \in V \text{ and } 0 \leq i \leq \delta(v)\}$ *and we write* $\mu(v,i)$ *instead of* $\mu((v,i))$*.*

V is the set of vertices *of G, E is the* set of edges *of G, $\delta(v)$ is the* degree *of v, and $\mu(e)$ is the* reverse edge *of e. For an edge e, call the set $\{e, \mu(e)\}$ an* undirected edge*.*

*Let $\eta : E \to V$ and $\beta : E \to \mathbb{N}$ be the first and the second component of μ. Then $\eta(v,i)$ is the i-*th neighbor *of v, i is the* label *of the edge (v,i), and $\beta(v,i)$ is its* back-label*.*

Define the size *of G,* size(G)*, to be $|V|$.*

In the following a graph means undirected multi-graph.

Definition 3.2.2. *A graph $G' = \langle V', \delta', \mu' \rangle$ is a subgraph of a graph $G = \langle V, \delta, \mu \rangle$, if $V' \subseteq V$ and for every $u, v \in V'$ the number of edges between u and v in G' is at most the number of edges between u and v in G, i.e.*

$$|\{i : \eta'(u,i) = v\}| \leq |\{i : \eta(u,i) = v\}|.$$

Note that a subgraph does not have to inherit the edge-labels of its ambient graph.

Define (simple) path, connected vertices, forest and tree in the usual way.

An exploration walk is a procedure for visiting the edges of a given graph. An exploration walk moves along adjacent edges of the graph according to the following definition.

Definition 3.2.3. *Let G be a graph. Let $\Delta : E \times \mathbb{Z} \to E$ be such that $\Delta((v,i),j)$ changes by j the label of the edge (v,i). More precisely for $i \neq 0$,*

$$\Delta((v,i),j) = (v, 1 + (i - 1 + j \bmod \delta(v))).$$

Define $\Gamma_{G,k}, \Gamma'_{G,k} : E \to E$ inductively on $k \geq 0$. First $\Gamma_{G,0}(e) = \Gamma'_{G,0}(e) = e$. Now let

$$\begin{aligned} \Gamma_{G,k+1}(e) &= \Delta(\mu(\Gamma_{G,k}(e)), 1), \\ \Gamma'_{G,k+1}(e) &= \mu(\Delta(\Gamma'_{G,k}(e), -1)). \end{aligned}$$

$\Gamma_{G,k}(e)$ is called an exploration walk *starting from the edge e. $\Gamma'_{G,k}$ is called* the reverse exploration walk. *Let $e_k = (v_k, i_k) = \Gamma_{G,l}(e)$. v_k and e_k are correspondingly the k-th vertex and the k-th edge visited by the exploration walk.*

Exploration walks were introduced by Koucký [Kou01b]. In general an exploration walk is induced by an exploration sequence, which for every step of the walk prescribes the amount by which to change the label of the current edge. We define exploration walks only for the exploration sequence which always changes by one the label of the current edge, i.e. the exploration sequence is all-1's, because this is the case of interest to us. Exploration sequences are a simple extension of traversal sequences, which for every step prescribe the next edge label. The most important property of exploration walks, not shared by traversal walks, is that they are reversible [Kou01b]. In our case we have that $\Gamma'_{G,k}(\Gamma_{G,k}(e)) = e$. The reversibility property of exploration walks allows them to visit some edges of the graph and then return to the edge from which they started without "memorizing" all edges visited by the intermediate steps of the walk.

Similar to the case of traversal sequences, we have exploration sequences which are universal for a given class of graphs in the following sense. Given a graph G from

the class and an edge e from the graph, an exploration walk induced by a universal exploration sequence and starting at e visits all vertices of the graph. Despite the similarity with traversal sequences, Koucký showed that there are simple constructions of universal exploration sequences for classes of graphs for which the corresponding construction of universal traversal sequences was very complicated and even non-existent. For example he showed that the all-1's sequence is universal for simple cycles and trees. For simple cycles the construction of universal traversal sequences is much more complicated [BNBK$^+$89, Bri87, Ist88, Kou01a] and for trees such constructions are not known. Koucký also showed a simple construction of universal exploration sequence for graphs with small diameter (e.g. expander graphs) and the only known [HW93] construction for expander graphs of universal traversal sequences requires them to be consistently labeled.

The following proposition states formally the observation that the all-1's exploration sequence is universal for trees.

Proposition 3.2.1 (Koucký, [Kou01b]). *Let $e = (v, i) \in E$, $i \neq 0$, and $e_j = (v_j, i_j) = \Gamma_{G,j}(e)$, $j \geq 0$. If G is a tree with at most one undirected edge between any two vertices and $k = 2(\mathrm{size}(G) - 1)$, then $e_k = e$ and every edge of G which is not a self-loop appears exactly once in $e_0, \ldots, e_{k-1}$. Furthermore, $2(\mathrm{size}(G) - 1)$ is the smallest k such that v appears $\delta(v) + 1$ times in $v_0, \ldots, v_k$.*

3.2.2 Configurations

In this section we define a configuration to be a structure which describes the state of the CL algorithm. We also define a sequence of configurations, where each next configuration is obtained from the previous by a hooking or a contraction operation. The exact order of the hooking and contraction operations is derived from "linearizing" the recursive definition of the CL algorithm. Following [CL95] we prove properties of the elements of the sequence of configuration which on one hand imply the correctness properties of the CL algorithm, but on the other allow us also to implement the definitions as a space-efficient algorithm.

Definition 3.2.4. *A* configuration *is a tuple* $\mathcal{C} = \langle G, A, I, D, H, R\rangle$*, where* G *is a graph with* $V = [n]$*, for some* $n \in \mathbb{N}$*.* A*,* I*, and* D *form a partition of* V*, and* $\delta(v) = 0$*, for* $v \in D$*.* $H : V \to \mathbb{N}$ *and* $R : V \to V$ *are such that* $H(v) \leq \delta(v)$ *and if* $R(v) \neq v$*, then* $v \in D$*.*

The elements of A*,* I*, and* D *are called correspondingly the* active*, the* inactive*, and the* done *vertices of* G*. For* $u \in V$*,* $(u, H(u))$ *is the* hooking edge *of* u*, and* $R(u)$ *is the* immediate representative *of* u*.*

We require that H *and* R *do not have non-trivial cycles in the following sense. Let* $v_1, \ldots, v_k \in V$*,* $k \geq 2$*. Then*

1) if $v_{i+1} = \eta(v_i, H(v_i))$*,* $i \in [k-1]$*, and* $v_1 = \eta(v_k, H(v_k))$*, then* $H(v_1) = 0$*, and*

2) if $v_{i+1} = R(v_i)$*,* $i \in [k-1]$*, and* $v_1 = R(v_k)$*, then* $R(v_1) = v_1$*.*

We also require that there is at most one undirected edge between any two active vertices, i.e. if $u, v \in A$*, then* $|\{i : \eta(v, i) = u\}| \leq 1$*, and there is no hooking edge from an inactive to an active vertex, i.e. if* $u \in I$*, then* $\eta(u, H(u)) \in I$*.*

Define the representative of v according to R to be

$$\mathrm{rep}_R(v) = \begin{cases} v, & R(v) = v, \\ \mathrm{rep}_R(R(v)), & \text{otherwise.} \end{cases}$$

By 2) of Definition 3.2.4, this definition is correct.

Definition 3.2.5. *Let* $\mathcal{C} = \langle G, A, I, D, H, R\rangle$ *be a configuration.* H *defines a subforest* $F = \langle V, \delta_F, \mu_F\rangle$ *of* G*, called the* hooking forest *of* $\mathcal{C}$*, with at most one undirected edge between any two vertices in the following way. Fix* $v \in V$*. Let* ε *be 1, if* $H(v) \neq 0$*, and 0, otherwise. Let* $0 < i_1 < \cdots < i_k$ *be the labels of the edges of* v *along which its neighbors hooked to it, i.e.*

$$\{i_1, \ldots, i_k\} = \{i : \exists u \in V \text{ such that } \mu(u, H(u)) = (v, i)\}.$$

First define $\delta_F(v) = k + \varepsilon$*. Now define* $\eta_F(v, j) = \eta(v, i_j)$*, for* $1 \leq j \leq k$*, and, if* $\varepsilon = 1$*,* $\eta_F(v, k+\varepsilon) = \eta(v, H(v))$*. Finally define* $\beta_F(v, j) = i$*, where* $\eta_F(v, j) = u$ *and* $\eta_F(u, i) = v$*.*

Let T be a maximal connected subtree of the forest F. We call T a hooking tree *in $\mathcal{C}$. The* root *of T,* root(T)*, is the only vertex v in T such that $H(v) = 0$. The* degree *of T,* deg(T)*, is $\sum_{v \in V_T} \delta(v)$. For a vertex $v \in V$ we denote with T_v the subtree of F which contains v.*

The correctness of this definition and the fact that F is a forest with at most one undirected edge between any two vertices follow from 1) of Definition 3.2.4.

3.2.3 Operations on configurations

3.2.3.1 Hooking

We define Hook$(\mathcal{C})$ so that, if $\mathcal{C}$ describes the state of the sequential algorithm, then Hook$(\mathcal{C})$ is its state after one hooking step.

Being elements of $\mathbb{N}$, the elements of V are ordered. Define the linear ordering $<_d$ on V so that

$$u <_d v \text{ iff } \delta(u) < \delta(v) \text{ or } \delta(u) = \delta(v) \text{ and } u < v.$$

The result of the hooking operation Hook$(\mathcal{C}) = \langle G, A, I, D, H', R \rangle$ is the configuration defined in the following way. If v is inactive, then $H'(v) = H(v)$. If v is active, let $v_1, \ldots, v_{\delta(v)}$ be the neighbors of v, i.e. $v_i = \eta(v, i)$. For the rest of the definition when we have to choose an index i, we always pick the smallest one with the corresponding property. If v has an inactive neighbor v_i, let $H'(v) = i$. If all neighbors of v are active, let v_i be the largest according to $<_d$ amongst the neighbors of v. If $v <_d v_i$, let $H'(v) = i$. If all neighbors of v are active and smaller than v according to $<_d$, then if v has a neighbor v_i which has an inactive neighbor, let $H'(v) = i$. If all neighbors of v and their neighbors are active, then if v has a neighbor v_i which has a neighbor larger than v according to $<_d$, let $H'(v) = i$. Finally, if all neighbors of v and their neighbors are active and smaller than v according to $<_d$, define $H'(v) = 0$.

The hooking strategy described above differs from the hooking strategy of the CL algorithm because it gives preference to neighbors which are inactive or have an inactive neighbor. For example, in our hooking strategy a vertex hooks to an inactive neighbor, if it has one, regardless of its degree. In the hooking strategy of the CL algorithm, a vertex

hooks to its largest according to $<_d$ neighbor, regardless of its state. The new hooking strategy does not change the correctness of the CL algorithm because first an active vertex still declares itself a representative iff all of its neighbors are active and hooked to it, and second along a sequence of hooking edges of active vertices the vertices increase according to $<_d$ the same way as in the CL algorithm. The reason we chose the new strategy is to ensure that we do not lookup the degree of an inactive vertex. The essential properties of the hooking operation are given by the following lemmas. The proofs are the same as in [CL95].

Lemma 3.2.2 ([CL95]). *Let $k \geq 3$ and $v_i \in A$, $i \in [k]$, are distinct and such that $v_{i+1} = \eta(v_i, H'(v_i))$, $i \in [k-1]$. Then $v_1 <_d \max_d\{v_{k-1}, v_k\}$.*

Proof. By induction on k. For $k = 3$, the statement holds, because v_1 hooked to v_2, either because $v_2 >_d v_1$ or because $v_2 <_d v_1$, but v_2 had an inactive neighbor or an active neighbor larger than v_1. Since v_2 is hooked to v_3, the second must be the case. So $v_1 <_d \max_d\{v_2, v_3\}$. For the inductive step, we have that $v_{k-1} <_d \max_d\{v_k, v_{k+1}\}$ and $v_1 <_d \max_d\{v_{k-1}, v_k\}$, and so $v_1 <_d \max_d\{v_k, v_{k+1}\}$. □

Corollary 3.2.3 ([CL95]). Hook($\mathcal{C}$) *is a configuration.*

Proof. Since Hook only changes the hooking edges of active vertices, everything we have to check is that it does not create non-trivial cycles of active vertices. Assume that there is $k \geq 2$, and $v_i \in A$, $i \in [k]$, distinct and such that $v_{i+1} = \eta(v_i, H'(v_i))$ and $v_1 = \eta(v_k, H'(v_k))$. The case $k = 2$ is impossible because we must either have $v_1 <_d v_2$ or $v_1 >_d v_2$. In the first case v_2 hooks to v_1, only if v_1 hooked to a vertex different than v_2. The second case is also impossible, hence $k \geq 3$. By Lemma 3.2.2 $v_1 <_d \max_d\{v_{k-1}, v_k\}$ and $v_2 <_d \max_d\{v_k, v_1\}$. If $v_{k-1} <_d v_k$, then $v_1, v_2 <_d v_k$, a contradiction with Lemma 3.2.2 for v_k, v_1, and v_2. If $v_k <_d v_{k-1}$, then $v_1, v_k <_d v_{k-1}$, a contradiction with Lemma 3.2.2 for v_{k-1}, v_k, and v_1. Thus we must have that $H'(v_1) = 0$. □

Lemma 3.2.4 ([CL95]). *Let T be a hooking tree in* Hook($\mathcal{C}$) *composed entirely of active vertices. Then* $\text{size}(T) \geq 2$ *and* $\deg(T) < \text{size}^2(T)$.

Proof. Let $r = \text{root}(T)$. r hooks to itself, i.e. $H'(r) = 0$, iff all of its neighbors are active and hooked to it. So T must contain at least two vertices – r and its neighbors.

Let us see now that for $v \in V_T$,

$$\delta(v) \leq \delta(r). \tag{3.1}$$

If v is a neighbor of r in T, then since r hooked to itself, we must have that $v <_d r$ and hence that $\delta(v) \leq \delta(r)$. Otherwise, let u be the neighbor of r on the path in T from v to r. Then, by Lemma 3.2.2, $v <_d \max_d\{u, r\}$, and so again $\delta(v) \leq \delta(r)$.

We have that

$$\deg(T) = \sum_{v \in V_T} \delta(v) \leq \text{size}(T)\delta(r) < \text{size}^2(T),$$

where in the first inequality we use (3.1), and the second follows because, as explained earlier, T must contain all neighbors of r. $\delta(r)$ is exactly the number of distinct neighbors of r, since there are no multiple edges between active vertices. □

3.2.3.2 Contraction

We define Contract$(\mathcal{C}, d)$ so that, if $\mathcal{C}$ describes the state of the sequential algorithm, then Contract$(\mathcal{C}, d)$ is its state after one contraction step with parameter d. A hooking tree T in $\mathcal{C}$ is *d-contractable*, if $\deg(T) \leq d$.

The result of Contract$(\mathcal{C}, d)$ is the configuration $\mathcal{C}' = \langle G', A', I', D', H', R' \rangle$ defined in the following way. First define

$$\begin{aligned} A'' &= \{v : v \notin D \text{ and } \deg(T_v) \leq d \text{ and } \text{root}(T_v) = v\}, \\ I' &= \{v : v \notin D \text{ and } \deg(T_v) > d\}, \\ D'' &= \{v : v \in D \text{ or } \deg(T_v) \leq d \text{ and } \text{root}(T_v) \neq v\}. \end{aligned}$$

Now define

$$H'(v) = \begin{cases} H(v), & v \in I', \\ 0, & \text{otherwise}, \end{cases}$$

and

$$R'(v) = \begin{cases} R(v), & v \in D, \\ \text{root}(T_v), & v \in D'' - D, \\ v, & \text{otherwise}. \end{cases}$$

Let T be a hooking tree in $\mathcal{C}$ and $s = \text{size}(T)$. Let $v_1, \ldots, v_s$ be the enumeration of the vertices of T visited by the exploration walk starting from the edge $(\text{root}(T), 1)$ of T, where we enumerate a vertex only the first time it is visited by the exploration walk. Let $e_1, \ldots, e_k$ be the enumeration of the edges of G incident to the vertices of T defined in the following way – enumerate all edges incident to v_1, then all edges incident to v_2, and so on. Obviously $k = \deg(T)$.

Let us define now G'. For $u \in A''$, define $l_u \in \mathbb{N}$ and the following enumeration of edges $e_{u,j}$, $1 \leq j \leq l_u$. First consider the enumeration $e_1, \ldots, e_k$ of the edges of T_u from the previous paragraph. Remove from this enumeration all edges which are internal to T_u, i.e. such that $\eta(e_i) \in V_{T_u}$. From every subsequence of edges whose other end belongs to the same d-contractable hooking tree leave only the first edge, i.e. for every d-contractable hooking tree $T \neq T_u$ of $\mathcal{C}$, if $e_{i_1}, \ldots, e_{i_h}$ are all the edges in the sequence $e_1, \ldots, e_k$ such that $\eta(e_{i_j}) \in V_T$, leave only e_{i_1}. Let l_u be the number of remaining edges in the enumeration and $e_{u,j}$, $1 \leq j \leq l_u$, be their enumeration. Naturally we call the edges $e_{u,j}$, the *remaining edges of* T_u.

Define

$$\begin{aligned} A' &= A'' - \{v \in A'' : l_v = 0\}, \\ D' &= D'' \cup \{v \in A'' : l_v = 0\}, \end{aligned}$$

and

$$\delta'(v) = \begin{cases} l_v, & v \in A', \\ \delta(v), & v \in I', \\ 0, & v \in D'. \end{cases}$$

We are left to define $\mu'(v,i)$. First assume $v \in A'$. Let $(u,j) = \mu(e_{v,i})$. If T_u is not d-contractable, then define $\mu'(v,i) = (u,j)$. If T_u is d-contractable, then define $\mu'(v,i) = (w,k)$, where $w = \text{root}(T_u)$ and k is the only index such that $\eta(e_{w,k}) \in V_{T_v}$. Now assume $v \in I'$. Let $(u,j) = \mu(v,i)$. If T_u is not d-contractable, then define $\mu'(v,i) = (u,j)$. If T_u is d-contractable, let $\mu'(v,i) = (w,k)$, where $w = \text{root}(T_u)$ and k is the only index such that $e_{w,k} = (u,j)$.

Lemma 3.2.5. *Let* $\mathcal{C}' = \mathrm{Contract}(\mathcal{C}, d)$. *Then* $\mathcal{C}'$ *is a configuration such that* $\delta'(v) \leq d$, *for* $v \in A'$, *and* $\deg(T_v) > d$, *for* $v \in I'$. *Furthermore, in the hooking forest of* $\mathcal{C}'$ *every* $v \in A' \cup D'$ *is in a hooking tree which contains only* v.

Proof. Follows from the definition of $\mathrm{Contract}(\mathcal{C}, d)$. □

3.2.4 Sequence of configurations

For every $k \in \mathbb{N}$, we define a sequence of configurations

$$\mathcal{C}_l = \langle G_l, A_l, I_l, D_l, H_l, R_l \rangle,\ 0 \leq l \leq r(k),$$

where

$$r(k) = 5 \cdot 2^k - 3.$$

First, define recursively a sequence of pairs, the first element of which is always either $Hook$ or $Contract$, and the second is a natural number, in the following way

$$S_k = \begin{cases} (\langle Hook, 0 \rangle, \langle Contract, 1 \rangle), & k = 0, \\ (\langle Hook, k \rangle, \langle Contract, k \rangle, S_{k-1}, S_{k-1}, \langle Contract, k+1 \rangle), & k > 0. \end{cases}$$

The definition of S_k is obtained by "linearizing" the recursive definition of $\texttt{Connect}(k)$ from section 3.1.

By induction on k, S_k has $r(k)$ elements $P_1, \ldots, P_{r(k)}$. For $1 \leq l \leq r(k)$, let Op_l and Arg_l be the first and the second component of P_l, i.e. $P_l = \langle Op_l, Arg_l \rangle$, $Op_l \in \{Hook, Contract\}$, and $Arg_l \in \mathbb{N}$. Let $\mathcal{C}_0$ be some configuration. Assume that we have already defined $\mathcal{C}_0, \ldots, \mathcal{C}_{l-1}$ for $1 \leq l \leq r(k)$. Define

$$\mathcal{C}_l = \begin{cases} \mathrm{Hook}(\mathcal{C}_{l-1}), & Op_l = Hook, \\ \mathrm{Contract}(\mathcal{C}_{l-1}, \mathrm{dexp}(Arg_l + 1)), & \text{otherwise}, \end{cases}$$

where

$$\mathrm{dexp}(x) = 2^{2^x}.$$

Definition 3.2.6. *A configuration* $\mathcal{C}_l$, $0 \leq l < r(k)$ *is* nice, *if*

1. *if* $Op_{l+1} = Hook$, *then* $\mathrm{size}(T_v) > \mathrm{dexp}(Arg_{l+1}+1)$ *and* $\deg(T_v) > \mathrm{dexp}(Arg_{l+1}+2)$, *for* $v \in I_l$, *and*

2. $\delta_l(v) \leq \mathrm{dexp}(Arg_{l+1} + 2)$, *for* $v \in A_l$.

By definition, if $Op_{l+1} = Hook$, then $\mathcal{C}_l$ is nice iff the state described by it fulfills the preconditions given in [CL95] for executing $\texttt{Connect}(Arg_{l+1})$. These preconditions ensure the correctness of the algorithm and that it can be executed efficiently in parallel. Considering the correspondence between the sequence $\mathcal{C}_0, \ldots, \mathcal{C}_{r(k)}$ and the $\texttt{Connect}(k)$ procedure of the CL algorithm, the following theorem is a consequence of the results in [CL95].

Theorem 3.2.6 (Chong–Lam, [CL95]). *If $\mathcal{C}_0$ is a nice configuration, then $\mathcal{C}_l$ is nice, for every $1 \leq l < r(k)$, $|A_{r(k)}| \leq \max\{|A_0|/\mathrm{dexp}(k), 1\}$, and $\mathrm{size}(T_v) > \mathrm{dexp}(k+1)$, for $v \in I_{r(k)}$.*

Finally, again by [CL95], we have the following corollary, which says that if we initialize $\mathcal{C}_0$ according to some undirected graph G, then in $\mathcal{C}_r$ all components of G are contracted.

Corollary 3.2.7 (Chong–Lam, [CL95]). *Let G be a graph with at most one undirected edge between any two vertices and $V = [n]$. Let $k = \lceil \log \log n \rceil$ and $\mathcal{C}_0 = \langle G, A, I, D, H, R \rangle$, where $A = \{v : \delta(v) \neq 0\}$, $I = \emptyset$, $D = V - A$, $H(v) = 0$ and $R(v) = v$, for $v \in V$. Then u and v are connected in G iff $\mathrm{rep}_{R_{r(k)}}(u) = \mathrm{rep}_{R_{r(k)}}(v)$.*

The proofs of Theorem 3.2.6 and Corollary 3.2.7 are a direct translation into our notation of equivalent statements from [CL95].

Lemma 3.2.8 ([CL95]). *Assume that $\mathcal{C}_l$ is nice and $P_{l+1} = \langle Hook, k \rangle$.*

1. *If $v \in I_l$, then $v \in I_{l+r(k)}$, $\delta_{l+r(k)}(v) = \delta_l(v)$, and $H_{l+r(k)}(v) = H_l(v)$.*
2. *If $v \in I_{l+r(k)} - I_l$, then $\delta_{l+r(k)}(v) \leq \mathrm{dexp}(k+2)$.*
3. *$\mathcal{C}_{l+1}$ is nice. If $k > 0$, $\mathcal{C}_{l+2}$ is nice.*

Proof. 1. The state and degree of a vertex change only during a contraction operation. The same holds for the hooking edges of inactive vertices. Let T be the hooking tree of

v in $\mathcal{C}_l$. Since $\mathcal{C}_l$ is nice, $\deg(T) > \text{dexp}(k+2)$. For $l+1 \leq i \leq l+r(k)$, we have that $Arg_i \leq k+1$, so trees of degree more than $\text{dexp}(k+2)$ are never contracted. Therefore the status, degree, and hooking edge of v is never changed.

2. Since $v \in I_{l+r(k)} - I_l$ and because a done vertex stays done, we have that $v \in A_l$, and so $\delta_l(v) \leq \text{dexp}(k+2)$, because $\mathcal{C}_l$ is nice. Hooking does not change degrees. For $l+1 \leq i \leq l+r(k)$, we have that $Arg_i \leq k+1$, so any active vertex appearing after a contraction must have degree at most $\text{dexp}(k+2)$. Let i be the largest $l \leq i \leq l+r(k)-1$, such that $v \in A_i$. Then $\delta_i(v) \leq \text{dexp}(k+2)$ and its degree does not change afterwards. Hence $\delta_{l+r(k)}(v) \leq \text{dexp}(k+2)$.

3. We have that $P_{l+2} = \langle Contract, k \rangle$ or $P_{l+2} = \langle Contract, 1 \rangle$, if $k = 0$. Since a hooking operation only changes the hooking edges of active vertices, $\mathcal{C}_{l+1}$ is nice. Furthermore, for a hooking tree T' in $\mathcal{C}_{l+1}$ which contains an inactive vertex, we have that $\text{size}(T') > \text{dexp}(k+1)$

If $k > 0$, then $P_{l+3} = \langle Hook, k-1 \rangle$. Let T be a hooking tree in $\mathcal{C}_{l+2}$ composed of inactive vertices. Because $P_{l+2} = \langle Contract, k \rangle$, we have that $\delta_{l+2}(v) \leq \text{dexp}(k+1)$, for $v \in A_{l+2}$, and $\deg(T) > \text{dexp}(k+1)$. There are two possibilities for T – either it was a hooking tree in $\mathcal{C}_{l+1}$ composed entirely of active vertices or it contains a hooking tree T' of $\mathcal{C}_{l+1}$ which contained an inactive vertex. In the first case by Lemma 3.2.4, $\text{size}(T) > \text{dexp}(k)$. This is also true in the second case because, as mentioned at the end of the previous paragraph, $\text{size}(T) > \text{dexp}(k+1) > \text{dexp}(k)$. Therefore $\mathcal{C}_{l+2}$ is nice. □

Lemma 3.2.9 ([CL95]). *Assume that $\mathcal{C}_l$ is nice and $P_{l+1} = \langle Hook, k \rangle$.*

1. If $k > 0$, then $\mathcal{C}_{l+2+r(k-1)}$ and $\mathcal{C}_{l+2+2r(k-1)}$ are nice.

2. If T is hooking tree in $\mathcal{C}_{l+r(k)}$ composed of inactive vertices, then $\text{size}(T) > \text{dexp}(k+1)$ *and* $\deg(T) > \text{dexp}(k+2)$.

Proof. Let $l_0 = l+2$, $l_1 = l_0 + r(k-1)$, $l_2 = l_1 + r(k-1)$, and $l_3 = l_2 + 1 (= l + r(k))$.

Since $P_{l_3} = \langle Contract, k+1 \rangle$, for any hooking tree T of $\mathcal{C}_{l_3}$ composed of inactive vertices, we have that

$$\deg(T) > \text{dexp}(k+2). \tag{3.2}$$

Let T' be a hooking tree in $\mathcal{C}_l$ composed of inactive vertices. By Lemma 3.2.8.1, there exists a unique hooking tree T in $\mathcal{C}_{l_3}$ such that $T' \subseteq T$. Since $\mathcal{C}_l$ is nice, $\text{size}(T') > \text{dexp}(k+1)$, and hence $\text{size}(T) > \text{dexp}(k+1)$.

Let T be a hooking tree in $\mathcal{C}_{l_3}$ which does not contain vertices from I_l. We use induction on k to show that $\text{size}(T) > \text{dexp}(k+1)$. If $k = 0$, then by Lemma 3.2.4 and (3.2), $\text{size}(T) > \text{dexp}(1)$.

Let $k > 0$. By Lemma 3.2.8.3, $\mathcal{C}_{l_0}$ is nice, thus by the inductive hypothesis (notice that $P_{l_0+1} = \langle Hook, k-1 \rangle$), for any hooking tree T' in $\mathcal{C}_{l_1}$ composed of inactive vertices, we have that $\text{size}(T) > \text{dexp}(k)$ and $\deg(T) > \text{dexp}(k+1)$. Furthermore, since $P_{l_1} = \langle Contract, k \rangle$, for any $v \in A_{l_1}$, $\delta_{l_1}(v) \leq \text{dexp}(k+1)$. Thus $\mathcal{C}_{l_1}$ is nice. Similarly $\mathcal{C}_{l_2}$ is nice. This proves the first item of the lemma.

From $\mathcal{C}_l$ nice and $P_{l_0} = \langle Contract, k \rangle$, follows that $I_l \subseteq I_{l_0}$. Also by Lemma 3.2.8.1, $I_{l_0} \subseteq I_{l_1} \subseteq I_{l_2}$. Finally $I_{l_3} \subseteq I_{l_2}$, because $Op_{l_3} = Contract$.

For $l_0 + 1 \leq i \leq l_2$, we have that $Arg_i \leq k$. Therefore we can use the same argument as in Lemma 3.2.8.2 to show that for any $v \in I_{l_2} - I_{l_0}$,

$$\delta_{l_2}(v) \leq \text{dexp}(k+1). \tag{3.3}$$

Since $I_{l_3} \subseteq I_{l_2}$, we have that $V_T \subseteq I_{l_2}$. If $V_T \cap I_{l_0} = \emptyset$, then $\delta_{l_2}(v) \leq \text{dexp}(k+1)$, for $v \in V_T$, and hence $\deg(T) \leq \text{size}(T)\text{dexp}(k+1)$. Thus, from (3.2), follows that $\text{size}(T) > \text{dexp}(k+1)$.

Assume that $V_T \cap I_{l_0} \neq \emptyset$. From Lemma 3.2.8.1, follows that $V_T \cap I_{l_0}$ form a hooking tree T' in $\mathcal{C}_{l_0}$. Let $d = \deg(T')/\text{size}(T')$. If $d \leq \text{dexp}(k+1)$, then $\deg(T') \leq \text{dexp}(k+1)\text{size}(T')$. Also for $v \in V_T - V_{T'} \subseteq I_{l_2} - I_{l_0}$, by (3.3), $\delta_{l_2}(v) \leq \text{dexp}(k+1)$. Hence

$$\deg(T) = \sum_{v \in V_{T'}} \delta_{l_2}(v) + \sum_{v \in V_T - V_{T'}} \delta_{l_2}(v) \leq \text{dexp}(k+1)\text{size}(T),$$

and so $\text{size}(T) > \text{dexp}(k+1)$, because of (3.2).

Finally assume that $d > \text{dexp}(k+1)$. Since $V_{T'} \subseteq V_T$, we have that $V_{T'} \cap I_l = \emptyset$. Hence $V_{T'} \subseteq A_l$ and so, by Lemma 3.2.4, $\text{size}(T') > \deg(T')/\text{size}(T') = d > \text{dexp}(k+1)$.

Therefore, by (3.3)

$$\begin{aligned}\deg(T) &\leq \deg(T') + \mathrm{dexp}(k+1)(\mathrm{size}(T) - \mathrm{size}(T')) \\ &< \mathrm{size}^2(T') + \mathrm{size}(T')(\mathrm{size}(T) - \mathrm{size}(T')) = \mathrm{size}(T')\mathrm{size}(T) \leq \mathrm{size}^2(T).\end{aligned}$$

Hence in this last case and using (3.2) again, $\mathrm{size}(T) > \mathrm{dexp}(k+1)$ as well. □

Lemma 3.2.10 ([CL95]). *Assume that $\mathcal{C}_l$ is nice and $P_{l+1} = \langle Hook, k\rangle$. Then for every $v \in A_{l+r(k)}$,*

$$|\{u \in A_l : \mathrm{rep}_{l+r(k)}(u) = v\}| \geq \mathrm{dexp}(k).$$

Proof. We do induction on k. Let $k = 0$. By Lemma 3.2.8.1, $I_{l+2} \subseteq I_l$, and so $A_l \subseteq A_{l+2}$. Consider $v \in A_{l+2}$. v must be a root of a hooking tree T in $\mathcal{C}_{l+1}$ and $V_T \subseteq A_{l+1}$, because $\mathcal{C}_l$ is nice and $P_{l+2} = \langle Contract, 1\rangle$. Hooking does not change the states of vertices and so $A_l = A_{l+1}$. By Lemma 3.2.4, T has at least two vertices and therefore v represents at least two vertices from A_l.

Let the hypothesis be true for $k - 1 \geq 0$. Let $l_0 = l + 2$, $l_1 = l_0 + r(k-1)$, $l_2 = l_1 + r(k-1)$, and $l_3 = l_2 + 1 (= l + r(k))$. As we saw in the proof of Lemma 3.2.9, we have that $A_{l_2} \subseteq A_{l_1} \subseteq A_{l_0} \subseteq A_l$ and $A_{l_2} \subseteq A_{l_3}$. Let $v \in A_{l_3}$. We have that v is in either A_{l_2} or in I_{l_2}.

Assume first that $v \in I_{l_2}$. Since $v \in A_{l_3}$ and $P_{l_3} = \langle Contract, k+1\rangle$, v is a root of a hooking tree T of $\mathcal{C}_{l_2}$ comprised of inactive vertices which contracts to v. $V_T \subseteq A_l$, because if $u \in V_T \cap I_l$, then by Lemma 3.2.8.1 $u \in V_T \cap I_{l_3}$, but we have that $V_T \cap I_{l_3} = \emptyset$. By Lemma 3.2.9.1 $\mathcal{C}_{l_1}$ is nice and therefore by Lemma 3.2.9.2, $\mathrm{size}(T) > \mathrm{dexp}(k)$. Thus v represents at least $\mathrm{dexp}(k)$ vertices from A_l.

Assume now that $v \in A_{l_2}$. Let

$$U_v = \{u \in A_{l_1} : \mathrm{rep}_{l_2}(u) = v\}.$$

By the inductive hypothesis $|U_v| \geq \mathrm{dexp}(k-1)$. Let $u \in U_v$. Again by the inductive hypothesis $|\{w \in A_{l_0} : \mathrm{rep}_{l_1}(w) = u\}| \geq \mathrm{dexp}(k-1)$. Hence

$$|\{w \in A_{l_0} : \mathrm{rep}_{l_2}(w) = v\}| \geq \mathrm{dexp}(k-1)\mathrm{dexp}(k-1) = \mathrm{dexp}(k).$$

Since $A_{l_0} \subseteq A_l$, the statement follows in this case as well.

□

We are ready to prove Theorem 3.2.6.

Proof of Theorem 3.2.6. The statement that $|A_{r(k)}| \leq \max\{|A_0|/\mathrm{dexp}(k), 1\}$ follows from Lemma 3.2.10 and the fact that the sets of vertices represented by different vertices from $A_{r(k)}$ are disjoint. $\mathrm{size}(T_v) > \mathrm{dexp}(k+1)$, for $v \in I_{r(k)}$, follows from Lemma 3.2.9.2.

We have to prove now that $\mathcal{C}_l$ is nice, for $1 \leq l < r(k)$. We prove by induction on t that, if $\mathcal{C}_l$ is nice and $P_{l+1} = \langle Hook, t \rangle$, then $\mathcal{C}_i$ is nice for $l+1 \leq i < l + r(t)$ and if $0 < t < k$, then $\mathcal{C}_{l+r(t)}$ is also nice.

By Lemma 3.2.8.3, $\mathcal{C}_{l+1}$ is nice. This takes care of $t = 0$. Assume that $t > 0$. By Lemma 3.2.8.3 and Lemma 3.2.9.1, $\mathcal{C}_{l+2}$, $\mathcal{C}_{l+2+r(t-1)}$, and $\mathcal{C}_{l+2+2r(t-1)}$, are nice. By the inductive hypothesis applied twice $\mathcal{C}_i$ is nice, for $l+3 \leq i < l+2+r(t-1)$ and $l+3+r(t-1) \leq i < l+2+2r(t-1)$. Since $P_{l+r(t)} = \langle Contract, t+1 \rangle$, we have that $\delta_{l+r(t)}(v) \leq \mathrm{dexp}(t+2)$, for $v \in A_{l+r(t)}$. By Lemma 3.2.9.2, for any hooking tree of $\mathcal{C}_{l+r(t)}$ composed of inactive vertices $\mathrm{size}(T) > \mathrm{dexp}(t+1)$ and $\deg(T) > \mathrm{dexp}(t+2)$. If $t < k$, then $P_{l+r(t)+1}$ is either $\langle Contract, t+2 \rangle$ or $\langle Hook, t \rangle$, and in both cases $\mathcal{C}_{l+r(t)}$ is nice. □

Finally let us prove Corollary 3.2.7.

Proof of Corollary 3.2.7. Let $k = \lceil \log\log n \rceil$. By Theorem 3.2.6,

$$|A_{r(k)}| \leq \max\{n/\mathrm{dexp}(k), 1\} = 1.$$

On the other hand, again by Theorem 3.2.6, every hooking tree in $\mathcal{C}_{r(k)}$ composed entirely of inactive vertices has size more than $\mathrm{dexp}(k+2) \geq n^2$. Therefore $I_{r(k)} = \emptyset$.

Suppose that u and v are connected in G, but $r_1 = \mathrm{rep}_{r(k)}(u) \neq \mathrm{rep}_{r(k)}(v) = r_2$. As we saw $|A_{r(k)}| \leq 1$ and $I_{r(k)} = \emptyset$, so w.l.o.g. we can assume that $r_1 \in D_{r(k)}$. Let l be the largest such that $r_1 \notin D_l$. $\delta_{l+1}(r_1) > 0$, because $r_1 \neq r_2$, r_1 and r_2 are connected in G, and r_1 inherits the neighbors of the vertices it represents. This contradicts the fact that r_1 becomes done, if it is a root of a hooking tree and its degree is 0 (it can also become

done if it is a non-root of a hooking tree which is contracted, but then it would not be a representative). □

3.3 The space-efficient sequential algorithm

Having built the abstract base of our algorithm in the previous section and reasoned about its correctness this section describes the algorithm itself and proves its space bound. Since giving directly the Turing Machine which solves the problem and reasoning about its space complexity is rather cumbersome, we define the algorithms in the following sections using definitions of recursive functions in pseudo-code and then translate the pseudo-code to a Turing Machine. The algorithms, as given by the pseudo-code, are almost a literal rephrasing of the definitions given in the previous sections into a precise language in which analyzing space complexity is possible. The deviations from a literal rephrasing exist only to decrease the space requirements of the algorithms. Besides that, the correctness of the algorithms follows essentially from Corollary 3.2.7. In fact, the algorithm in section 3.3.3 can be thought of as a literal $O(\log^2 n)$ space implementation of the sequential algorithm outlined at the end of section 3.1.

Section 3.3.1 uses the informal framework of communicating oracles to describe the transition between the abstract definitions of section 3.2 and the definition of the algorithm using pseudo-code. In section 3.3.2 we outline an $O(\log^2 n)$ space implementation of the sequential algorithm derived from the definitions in section 3.2, which instead of storing all of the current configuration, recomputes parts of it when it needs them. Section 3.3.3 describes the changes we make to the algorithm from section 3.3.2 to reduce its space complexity to $O(\log n \log\log n)$. In section 3.3.5 we discuss in detail the pseudo-code for the algorithm from section 3.3.3.

3.3.1 Communicating oracles: an informal description

The material included in this section constitutes a transition between the abstract definitions of section 3.2 and the precise description of the space-efficient algorithm in the rest of this chapter. The point of view presented here is not necessary to establish formal

properties of our space-efficient algorithm, e.g. its correctness and space-bound, and is rather an informal road-map of the connection between the mathematical structures of section 3.2, in which the correctness is evident, but the space bound is not relevant, and the algorithm, to which the space-bound becomes essential.

At the bottom of our informal description is the notion of *oracle*. From a mathematical point of view an oracle is a function from a set whose elements we call questions to a set whose elements we call answers. To keep the metaphor, we say that we ask an oracle a question to which it provides an answer. From a computational point of view an oracle has limited power, more precisely it can use only a limited amount of space. Furthermore, communication with an oracle is done through a *communication space*, so that to ask a question from the oracle we write it on the communication space, and once the oracle knows the answer it writes it back there, overwriting our last question. This is somewhat different from the traditional oracles of computational complexity which are assumed to have unlimited power. In our context, we use the word oracle to emphasize the possibility of interaction through questions and answers.

One more feature of the oracles in our context is that they can be organized in a *communication network* such that to provide an answer an oracle can ask questions from other oracles. In this scenario, when we talk about the space of an oracle to answer a query we mean only its internal space. On the other hand, the total space of a query is the sum of the internal and communication spaces of all oracles which were queried because of propagation of queries in the network while the initial query was answered. An example of a communication network is a simple path – i.e. outside queries can only be asked from the last oracle in the path, the first oracle knows the input to the problem being solved, and to answer a query every oracle can query the previous in the path.

Let us exemplify the above with a simple problem – Cycle Length. In this problem we have a 2-regular undirected graph on n vertices, i.e. a collection of cycles, a vertex v from this graph, and a number l. We would like to know whether the cycle to which v belongs is of length at most l. The input graph to this problem is represented through an oracle whose set of questions and answers are the neighbor pairs (u, i) of a vertex u

and a edge-label $i \in [2]$, so that if asked what is the i-th neighbor of u, the oracle answers with (w, j), where w is the i-th neighbor of u and u is the j-th neighbor of w. Initially the communication space is $(v, 1)$. We are interested in what is the smallest amount of space necessary to solve this problem, where we count only the space aside from the communication space of the oracle, which is always $\Theta(\log n)$.

The first solution to this problem uses $\Theta(\log n)$ space in the following way. Store v after reading it from the communication space. Then, while the vertex u in the communication space is not v (we check this by comparing u to the value we stored at the beginning) query the oracle, and after it gives an answer, flip the edge-label on the communication space. Notice that in this solution we have to store the vertex v at the beginning, because after a query the oracle overwrites the content of the communication space and we need a way of knowing when is the traversal of the cycle of v over.

The second solution to the problem uses space $O(\log l + \log\log n)$, which can be substantially smaller than $\Theta(\log n)$. For this solution we avoid storing the starting vertex v in the following way. Keep a counter i which can be at most l and compare v with the vertex v_i which is i steps away from v on the cycle. For this, assume that the content of the communication space is $(v, 1)$. Store the first bit of v and make i queries to the oracle, flipping the edge-label after each query. After this, the vertex on the communication space is v_i and we can compare its first bit with the first bit of v, which we kept stored. After this comparison we can go back to v by making again i queries to the oracle and flipping the edge-label after each query. At this point the communication space contains again $(v, 1)$ and we can store the second bit of v and repeat the same, this time comparing second bits. We can do this $\Theta(\log n)$ times comparing v_i and v bit by bit. For this we also need a counter, which goes up to $\Theta(\log n)$, i.e. takes space $\Theta(\log\log n)$, of which bit we are comparing at the moment. This way we compare v to each of the vertices on its cycle in order, without ever storing v.

The above problem and its space-efficient solution can be extended to graphs which are a collection of disjoint trees, i.e. to the Tree Size problem. For this we use Proposition 3.2.1 and substitute "flip the edge-label" with "increase by one the edge-label modulo the

degree of the current vertex" in the algorithm described in the previous paragraph. The oracle must also be able to return the degree of a vertex and we must have enough space to store such a degree, if it is at most s. In any case, using that according to Proposition 3.2.1 to determine whether we have traversed the whole tree of v it is enough to count how many times we visit v, we can determine whether the tree of v has size at most s in space $O(\log s + \log\log n)$.

Let us go back to undirected st-connectivity. For this problem we have oracles which perform operations on configurations as described in section 3.2.3. More specifically, we have two types of oracles – hooking and contracting, each responsible for one type of operation. Furthermore, each oracle answers queries regarding a configuration, i.e. each oracle is able to answer questions about the degree of a vertex, its neighbors, back-labels, hooking edge, status, and representative, as given by the definition of configuration in section 3.2.2. To answer a query each oracle can itself ask queries from another oracle regarding the configuration it is modifying. The oracles are arranged in a communication path as described earlier, such that the l-th oracle on this path provides the description of $\mathcal{C}_l$, $0 \leq l \leq r(\lceil \log\log n \rceil)$, as given in section 3.2.4 and Corollary 3.2.7, where $r(k) = 5 \cdot 2^k - 3$. As described in section 3.2.4, the type of the l-th oracle depends only on l and it communicates only with the $(l-1)$-th oracle (the 0-th oracle knows only the input and describes $\mathcal{C}_0$ as given in Corollary 3.2.7).

The fact that each oracle can answer a query using only space $O(\log n)$ is rather straightforward – in that much space we can store a constant number of vertices and the hooking and contraction operation do not need anything more than that. See the proof of Claim 3.3.1 in the next section for more details. Also the communication space of each oracle can be set to be $\Theta(\log n)$ since the queries to each oracle contain a vertex. Since we have $\Theta(\log n)$ oracles each having $O(\log n)$ internal and communication space, the total space for the last oracle to answer a query is $O(\log^2 n)$. This essentially is the algorithm outlined in the next section.

To reduce the total space for the last oracle to answer a query we use the idea of the space-efficient solution to Cycle Length and Tree Size described above. Namely, the

oracles are defined so that they never store a vertex internally. Furthermore, instead of each oracle having its own communication space, all oracles share this space. Thus, if we call the vertex present in the communication space with an oracle before it is queried, the *current vertex*, then there is only one current vertex for the whole communication network and every oracle answers a query regarding this vertex. This of course requires a much more careful use of the communication space. In particular, the oracles are defined so that queries which do not return a vertex, e.g. a query regarding the degree of the current vertex, do not change the current vertex. On the other hand, queries which return a vertex, e.g. a query requesting the i-th neighbor of the current vertex, change the current vertex according to their answer, and to keep the intuitive picture, we say that they *move* the current vertex.

To allow an oracle to move the current vertex temporarily, do something at the new vertex, e.g. query its degree or lookup its label, and then restore it to its original value, we use the notion of *path description* relative to the current vertex. For example, a hooking oracle needs to lookup the degree of the i-th neighbor of the current vertex. A path description specifies a short list of queries which move the current vertex. For example, a neighbor index, i.e. edge-label, is a path description and so is the length of an exploration walk in a hooking tree. In general, a path description is a sequence of constant length of neighbor indices and exploration walk lengths. Think of neighbor indices specifying short, local jumps in the graph, and exploration walk lengths specifying long, global jumps. A path description identifies a vertex, but contrary to the vertex itself, which takes $\Theta(\log n)$ bits to be stored locally, a path description might take $o(\log n)$ bits. Thus a path description, lets an oracle walk to a vertex whenever it needs to. This exactly corresponds to the space-efficient solution to Cycle Length described above, where the variable i was a path description to the vertex v_i, and we used that description to move to v_i every time we wanted one of its bits. The important property of path descriptions is that they are reversible, in the sense that if P is a path description which takes v to u, then there is a path description P' which takes u to v. For example, the back-label to an edge-label of a vertex is its reverse, and the reversibility of exploration walks on

trees follows by the reversibility of exploration walks as noticed by Koucký. Thus, path descriptions do exactly what they are supposed to – allow the oracle to move the current vertex temporarily.

Using only one current vertex and employing short path descriptions whenever it is necessary to move the current vertex temporarily, the oracles can be defined so that they never store a vertex internally. This opens up the possibility for the oracles to be defined so that their internal spaces are different. More precisely, the l-th oracle can be defined so that its internal space is $O(2^{Arg_l} + \log\log n)$, where Arg_l was defined at the beginning of section 3.2.4. We will see in section 3.3.4 that

$$\sum_{l=1}^{r(\lceil \log\log n \rceil)} 2^{Arg_l} = O(\log n \log\log n).$$

At this point, using oracles to describe the space-efficient algorithm becomes too cumbersome, and to give the precise details of the above considerations we abandon communicating oracles and switch to a more traditional descriptive framework, namely recursive functions. The correspondence between communicating oracles and recursive functions is that a level of recursion, or at least functions executed at a certain level of recursion, correspond to an oracle working on a query. The internal space of an oracle becomes local variables accessible only from within a function, and the common communication space of the oracles becomes global variables shared by all functions.

3.3.2 First attempt: an $O(\log^2 n)$ space algorithm

Let G be a graph with $V = [n]$ and with at most one undirected edge between any two vertices. Let $r = 5 \cdot 2^{\lceil \log\log n \rceil} - 3$. Consider the sequence of configurations $\mathcal{C}_0, \ldots, \mathcal{C}_r$ from Corollary 3.2.7 for $k = \lceil \log\log n \rceil$. The starting point for a space-efficient algorithm comes directly from the definition of this sequence. More precisely, we define functions `Active`(l, v), `Inactive`(l, v), `Done`(l, v), `Degree`(l, v), `Neighbor`(l, v, i), `BackLabel`(l, v, i), `Hook`(l, v), and `Rep`(l, v), where $0 \leq l \leq r$, v is a vertex, and i is the label of an edge incident to v, which return the corresponding component of $\mathcal{C}_l$.

Call the value of the parameter l, the *level of recursion*. Thus the levels of recursion of our algorithm correspond to the elements of the sequence $\mathcal{C}_0, \ldots, \mathcal{C}_r$. For $l = 0$ all of

these functions just use the input graph G (this is the bottom of the recursion), and their output for level $l+1$ is determined from outputs for level l according to the definitions in section 3.2.4. Using these functions, to solve undirected st-connectivity we apply Corollary 3.2.7.

If $Op_l = Contract$, then $\mathcal{C}_l$ is obtained from $\mathcal{C}_{l-1}$ by contracting some of its hooking trees as defined in section 3.2.3. In this case for a hooking tree T in $\mathcal{C}_{l-1}$ we must be able to determine its degree and to enumerate its vertices as they are visited by the exploration walk on T starting from $(v, 1)$, for $v \in V_T$. For these purposes we define $\texttt{TreeSize}(l, v)$ and $\texttt{TreeWalk}(l, v, i)$. Let T be the hooking tree in $\mathcal{C}_{l-1}$ containing v. If s is the size of T, $\texttt{TreeSize}(l, v)$ returns $2(s-1)$. Notice that, as stated in Proposition 3.2.1, $2(s-1)$ is the shortest length of the exploration walk on T which visits all the vertices of T. $\texttt{TreeWalk}(l, v, i)$ returns $\Gamma_{T,i}(v, 1)$.

Claim 3.3.1. *The functions* $\texttt{Active}(l, v)$, $\texttt{Inactive}(l, v)$, $\texttt{Done}(l, v)$, $\texttt{Degree}(l, v)$, $\texttt{Neighbor}(l, v, i)$, $\texttt{BackLabel}(l, v, i)$, $\texttt{Hook}(l, v)$, *and* $\texttt{Rep}(l, v)$, *correctly return the corresponding components of* $\mathcal{C}_l$. *The total space taken by the execution of each of these functions is* $O(l \log n)$.

Proof. The definitions of the functions mentioned in the claim follow the formal descriptions in sections 3.2.3 and 3.2.4. For example, the hooking operation is described in section 3.2.3 and to define $\texttt{Hook}(l, v)$ we follow this description, i.e. we define $\texttt{Hook}(l, v)$ so that it considers the neighbors of v and their neighbors as prescribed by the Hook operation applied to $\mathcal{C}_{l-1}$ and returns the hooking edge of v after this operation. In its definition Hook uses various components of $\mathcal{C}_{l-1}$ and those uses are replaced by recursive calls to corresponding functions at level $l-1$, e.g. if Hook requires the label of the i-th neighbor of v, i.e. $\eta_{l-1}(v, i)$, then we use $\texttt{Neighbor}(l-1, v, i)$. The fact that $\texttt{Hook}(l, v)$ may be defined so that it uses space $O(\log n)$, aside from the space incurred by the recursive calls, is immediate, because in that much space we can afford to store a constant number vertices in local variables of Hook. Notice that the definition of Hook essentially requires two nested loops – the outside over the neighbors of v and the inside over the neighbors of a fixed neighbor u of v. As mentioned earlier, if u is the i-th neighbor of v, then we

obtain it as a result of a call to `Neighbor`$(l-1, v, i)$ and store it in a local variable. Then when we need the j-th neighbor of u we use `Neighbor`$(l-1, u, j)$.

The definitions of the functions whose result is obtained after contraction, e.g. `Degree`(l, v), are perhaps more mysterious due to the involved description of the contraction operation in section 3.2.3. The main part of Contract is visiting the vertices of the hooking tree of v. To see that we can do this in $O(\log n)$ space, notice that to perform an exploration walk on a tree we just have to store locally the current state of the walk, i.e. a vertex and a tree-edge label, and then perform the walk until we visit v sufficiently many times, as given in Proposition 3.2.1. Again since we have $O(\log n)$ space there is no problem in storing the current state of the walk. Once we have a way of performing an exploration walk on a tree in space $O(\log n)$, the components of $\mathcal{C}_l$ after contraction are obtained exactly as described in section 3.2.3. For example, to compute $\delta_l(v)$, `Degree`(l, v) visits the vertices of the hooking tree T in $\mathcal{C}_{l-1}$ of v one by one to determine whether v will be active after contraction. For this, `Degree`(l, v) has to check whether v is the root of T and whether the degree of T is sufficiently small. If v will be active, another traversal visits the vertices of T and for each of them checks how many of their edges remain after the contraction observing that there should not be multiple edges between active vertices and removing any edges internal to T, as described in section 3.2.3. □

3.3.3 The $O(\log n \log\log n)$ space algorithm

The $O(\log n)$ space per level in Claim 3.3.1 comes mainly from having to store vertices in the local variables of the functions, since each vertex takes $\Theta(\log n)$ space. To see more precisely what is going on consider the following. The definition of $H_l(v)$ contains a comparison of the i-th neighbor of v in $\mathcal{C}_{l-1}$ with its j-th neighbor, i.e. the definition of `Hook`(l, v) contains a comparison `Neighbor`$(l-1, v, i) =$ `Neighbor`$(l-1, v, j)$. Let us say that v is passed to `Hook` through a global variable (see section 3.3.5 for more details on passing arguments to functions). Obviously, this global variable must be stored locally before the execution of such comparison, because otherwise its value might be overwritten during the two calls to `Neighbor`. To take care of this bottleneck we define the functions so that they never store a vertex in their local variables.

The first step towards such definitions is to remove the vertex v from the argument list of the functions. Instead of this argument, we maintain one current vertex in a global variable and all functions return information about this vertex. A function which otherwise must return a vertex is defined so that after its execution the current vertex is its result. In this case we say that the function moves the current vertex. It is a responsibility of the calling function to keep enough information locally to restore the original current vertex, if it needs to. Denote the current vertex with cv.

To implement this, first we change some of our functions. Instead of `Neighbor`(l, v, i), we have `Neighbor`(l, i), which moves the current vertex to $\eta_l(cv, i)$, its i-th neighbor in $\mathcal{C}_l$. Let T be the hooking tree of cv in $\mathcal{C}_{l-1}$. Instead of `TreeWalk`(l, v, i) we have `TreeForward`(l, i), which returns j and moves the current vertex to u, where $(u, j) = \Gamma_{T,i}(cv, 1)$. Similarly we have `TreeBack`$(l, i, j)$ which moves the current vertex to the end vertex of $\Gamma'_{T,i}(cv, j)$.

The most important part of our idea to avoid storing vertices locally is to be able to move the current vertex temporarily, perform something at the new current vertex, and then return to the original current vertex. For this, define `Move`(l, i) to return $\beta_{l-1}(cv, i)$, i.e. `BackLabel`$(l-1, i)$, and move the current vertex to $\eta_{l-1}(cv, i)$, i.e. a call to `Neighbor`$(l-1, i)$. Call `Move`(l, i) and `TreeForward`(l, i) *forward moves*. For a forward move M, let `Reverse`(M, j) be its reverse, i.e. it is correspondingly `Move`(l, j) or `TreeBack`(l, i, j), where j is the result of M. We use the reversibility of exploration walks here, so that `TreeBack` reverts `TreeForward`.

We use forward moves to change the current vertex and their reverses to restore it. Call a sequence of forward moves *path description relative to the current vertex*. If `P` is a path description relative to the current vertex and `B` is some instruction(s), then define **after** `P` **do** `B` to change the current vertex according to `P`, perform `B`, and then use the reverses of the moves in `P` to restore the current vertex.

A simple example of the use of `after` is the comparison operator `=`, which compares two vertices given their path descriptions relative to the current vertex and returns true iff they are the same. Using `after` we can move to the first vertex and store it in a local

variable, then go to the second vertex and compare the two. This takes $\Theta(\log n)$ space. Instead of this, going back and forth between the two vertices, using the reversibility of the moves along the edges and the exploration walks on the trees, we perform the comparison bit by bit. Aside from the information stored for the ways back, this takes only the $\Theta(\log\log n)$ space necessary to store the index of a bit. This way the bottleneck of $\Omega(\log n)$ space is reduced to $\Omega(\log\log n)$.

For example, the `=` operator is used in the definition of `TreeSize` in the following way. Using Proposition 3.2.1, we can make steps from the exploration walk on a hooking tree T in $\mathcal{C}_{l-1}$ until we go back to the starting vertex v sufficiently many times. For this we can store v, so that we can compare it with each new vertex of the walk. This takes $\Theta(\log n)$ space, independent of the degree and the size of T. Instead, we incrementally find i with properties as in Proposition 3.2.1, where to check if the i-th vertex of the walk is equal to v, we keep the current vertex at v and use the `=` operator, as defined above, to compare it to the vertex with path description `TreeForward`(l, i). Thus, if $\text{size}(T) \leq s$, then we can find its size in space $O(\log s + \log\log n)$ (the $\log\log n$ appears because of counters used during comparison of vertices). Alternatively, if $\text{size}(T) > s$, we can still use only $O(\log s + \log\log n)$ space to learn this without actually computing $\text{size}(T)$, because it is enough to stop the exploration walk on T as soon as we learn that $\text{size}(T) > s$.

The second part of our idea to reduce the space of the algorithm is to have an upper bound $v(l)$ on the values which variables can take at level l, i.e. during the execution of all functions at level l the values of their local variables are at most $v(l)$. We set

$$v(l) = 2 \cdot \text{dexp}(Arg_l + 2),$$

where $\text{dexp}(x) = 2^{2^x}$ and Arg_l is defined at the beginning of section 3.2.4. Call a number x *valid* for level l, if it is at most $v(l)$. A vertex v is *valid* for level l, if its degree $\delta_{l-1}(v)$ is valid for level l.

Using the concept of a current vertex, we can eliminate the need to store a vertex in a local variable and thus our local variables contain essentially only degrees of vertices, labels and back-labels of edges, and lengths of exploration walks on hooking trees. For

example, the information stored for reversing a forward move is a back-label (for `Move`) or a tree-edge label (for `TreeForward`). We still have to make sure that every time we store a value in a local variable it is valid. For this the following observation is helpful.

Observation 3.3.2. *1) The labels of the edges incident to vertex v valid for level l are valid for level l. This is not necessarily true for their back-labels. 2) All vertices which are active in $\mathcal{C}_{l-1}$ are valid for level l. 3) If $Op_l = Contract$, then if a hooking tree T in $\mathcal{C}_{l-1}$ has degree at most $\mathrm{dexp}(Arg_l + 1)$, then all of its vertices are valid for level l.*

The first item of the observation is trivial, the second follows from Theorem 3.2.6, because all $\mathcal{C}_l$ are nice, and the third follows because $\mathrm{dexp}(Arg_l + 1) < v(l)$.

Our goal has become to prove the following lemma. Let T be the hooking tree in $\mathcal{C}_{l-1}$ of the current vertex cv. Let (v, i) be an edge of T and $u = \eta_T(v, i)$. We call a move along (v, i) *possible for level l*, if u is valid for level l. T is *contractable for level l*, if $\deg(T) \leq \mathrm{dexp}(Arg_l + 1)$.

Lemma 3.3.3. *1.* `Active`(l), `Inactive`(l), `Done`(l), `Hook`(l), *and* `Degree`(l), *correctly return the value of the corresponding component of $\mathcal{C}_l$ for cv.*

2. *If T is uncontractable for level l or $cv \in D_{l-1}$, then* `Root`(l) *returns 0, otherwise it returns the index of the first occurrence of* $\mathrm{root}(T)$ *in the exploration walk on T starting from $(cv, 1)$.*

   `TreeSize`(l) *returns* $2(\texttt{size}(T) - 1)$, *if T is contractable for level l, and* `null` *otherwise.*

   Assume that cv is valid for level l. If all moves of $\Gamma_{T,i}(cv, 1)$ are possible for level l and it ends in (v, j), then `TreeForward`(l, i) *moves the current vertex to v and returns j. If all moves of $\Gamma'_{T,i}(cv, j)$ are possible for level l and it ends in v, then* `TreeBack`(l, i, j) *moves the current vertex to v.*

3. *Let $(v, j) = \mu_l(cv, i)$.* `Neighbor`(l, i) *moves the current vertex to v;* `BackLabel`(l, i) *returns j, if j is valid for the level at which* `BackLabel`(l, i) *was called (see the discussion on returning values in section 3.3.5), and* `null` *otherwise.*

   All local variables are valid.

The proof of the lemma is done by induction on the level of recursion. For this we need the correctness of the functions which a given function calls. Sometimes we have to use correctness for the same level of recursion, but this does not result in a circular reasoning because for any two functions `F` and `G`, there are no chains of function calls within the same level of recursion both from `F` to `G` and from `G` to `F`. The rest of this section should serve as a rather intuitive description as to why the lemma holds. More details for specific functions are given along with their definitions in section 3.3.5. In particular, the base the induction, i.e. $l = 0$, can be checked for each function by observing that in this case the function returns the value prescribed by the definition of $\mathcal{C}_0$ in Corollary 3.2.7. For example, the first lines of `Hook` and `Neighbor` do exactly this.

The correctness essentially follows from the correctness of the CL algorithm (Corollary 3.2.7). It can be easily seen that the introduction of one global current vertex and always returning information about this vertex, maintains the faithfulness of our implementation to the CL algorithm – the current vertex is an implicit argument to all functions describing a configuration and calling it "current" just facilitates our intuition about how the algorithm proceeds.

The only real deviation from the definitions given in section 3.2.2 is that we have an upper bound on the numerical values which can be stored at a level, and so we might be unable to process the result of a function, if it is invalid for the current level of recursion. Actually, as can be seen from Lemma 3.3.3, some functions are specified to return `null`, if their result is invalid for the level requesting it. By Observation 3.3.2, the only information we can derive from an invalid or `null` result is that either the current vertex is invalid (if `Degree`(l) is invalid), a neighbor of the current vertex is invalid (if `BackLabel`(l, i) is `null`), or that the current vertex is part of an uncontractable tree (if `TreeSize` is `null`). This information is enough to define the functions as in Lemma 3.3.3. First, we never move to a vertex from which we cannot return, i.e. along an edge with an invalid back-label, so we never have to store an invalid back-label locally. Second, during contraction vertices which are either invalid or part of an uncontractable tree become inactive and thus, by definition, inherit their properties, e.g. degree and hooking edge, from the previous level.

In a hooking operation (section 3.2.3), we do not need to lookup the degree of an invalid (and even inactive) vertex. In a contraction operation (section 3.2.3), we can stop the exploration walk on a tree as soon as the walk runs into an invalid vertex, because then the tree is clearly uncontractable.

To ensure that all local variables are valid for the current level of recursion we use Observation 3.3.2 in the following way. First, notice that since the value returned from a function resides in a global variable (see section 3.3.5 for more details on returning values from functions) we can check whether it is valid by simply inspecting this global variable. Furthermore, some functions (e.g. `TreeSize` and `BackLabel`) return `null`, if their result is invalid for the current level of recursion. In any case, we can easily learn when the return value of a function is invalid without having to store it locally. According to 1) of Observation 3.3.2, if the result of `BackLabel` is `null`, then the corresponding neighbor is invalid and we never move the current vertex along such edges. Also, 2) of Observation 3.3.2 ensures that we can always process locally active vertices. Finally, according to 3) of Observation 3.3.2, if the result of `TreeSize` is `null`, then the hooking tree of the current vertex is uncontractable.

3.3.4 Main theorem

Using Lemma 3.3.3 we can prove our main theorem concerning undirected st-connectivity

Theorem 3.3.4. *Undirected st-connectivity on a graph with n vertices can be solved in space $O(\log n \log\log n)$.*

By Corollary 3.2.7, s and t are connected iff $\mathrm{rep}_{R_r}(s) = \mathrm{rep}_{R_r}(t)$. Thus to solve USTCONN it is enough to define a function which moves the current vertex cv to $\mathrm{rep}_{R_r}(cv)$.

Let $m = \lceil \log\log n \rceil$. Recall that $r = 5 \cdot 2^m - 3 = O(\log n)$. The space complexity of the algorithm is dominated by the space taken by the execution stack. From Lemma 3.3.3 follows that each local variable at level l, except counters used in the comparison of vertices, is at most $v(l)$. Since there are constant number of local variables per function and the length of every chain of function calls within the same level of recursion is bounded

by a constant, the space taken by level l is $O(\log v(l) + m)$ (the additional $O(m)$ space appears because of the counters used during comparison of vertices). Since $\log v(l) = 4 \cdot 2^{Arg_l} + 1$, $1 \leq l \leq r$, and $r \cdot m = O(\log n \log\log n)$, we have to prove that

$$\sum_{l=1}^{r} 2^{Arg_l} = O(\log n \log\log n).$$

Consider the recurrence given by

$$S(k) = \begin{cases} 3, & k = 0, \\ 2S(k-1) + 2^{k+2}, & k > 0. \end{cases}$$

From the recursive definition of S_m given in section 3.2.4 follows that the left hand side of the equation which we want to prove is exactly $S(m)$. Finally, it is not hard to prove by induction on k that $S(k) = 2^k(4k+3)$. Hence $S(m) = O(\log n \log\log n)$.

3.3.5 Pseudo-code

The language of the pseudo-code is similar to Pascal. The most notable difference is that blocks are marked by indentation. We distinguish two cases of function definitions – procedures (marked by **procedure**) and functions (marked by **function**). Only functions return a value using the **return** statement. Procedures do not return a value. The **if** ... **then** ... **else** ..., **while** ... **do** ..., and **for** statements have the obvious semantics. **break** exits the closest surrounding loop, and **continue** continues with the next cycle of the execution of the closest surrounding loop. We use fixed width font to denote the names of functions and variables from the pseudo-code, e.g. `FooBar` and `i`, and a roman font for mathematical functions and variables, e.g. $FooBar$ and i.

P_1 **and** P_2 has the following semantics. If P_1 is false, then P_2 is not checked and the value of P_1 **and** P_2 is false. Otherwise the value of P_1 **and** P_2 is the value of P_2. P_1 **or** P_2 has similar semantics, except that the value of P_2 is checked only if P_1 is false.

3.3.5.1 Execution of the pseudo-code

The variable usage and execution of the pseudo-code is standard. During its execution a function can use only its own local variables and the global variables. For the execution of the functions we use a stack which contains the local variables of the functions executed

at the moment. The part of the stack devoted to the execution of a function is called the *stack frame of the function*. The top of this stack contains the stack frame of the function being executed at the moment. When the current function calls another function, first a new stack frame is allocated on the top of the stack and then the new function is executed. Part of the stack frame of a function contains information about the address (the place in the program) from which the function was called. Once the current function is finished its stack frame is removed from the top of the stack and the execution resumes from the address from which the current function was called.

In addition to functions which are executed using the stack we have *global functions*, which do not use the stack for their execution. Such functions use only global variables for their execution – i.e. their "local" variables are in fact global variables which are visible only from the particular global function. Since in what follows we are only concerned with the contents of the stack, we concentrate on functions which use the stack.

The execution of the pseudo-code proceeds in the following way. Every function is executed at some level of recursion. During its execution, a function can call other functions either on the same level of recursion or on a lower level of recursion. There is a constant c, such that the length of a chain of function calls within the same level of recursion is at most c. The stack frames of functions executed in the same level of recursion are consecutive on the stack. We call such a sequence of stack frames, the *stack frame for the level*. From the fact that any stack frame for a level contains at most c stack frames for functions and that each function uses a constant number of variables follows that there is a constant d such that the stack frame of every level contains a total of at most d variables.

The local and global variables contain only numerical and boolean values, and a special value `null`, different from any other value. For every level of recursion l we have an upper bound $v(l)$, computed by a global function, on the numerical values which are valid for this level, i.e. all numerical values at level l, except counters used during the comparison of vertices, are at most $v(l)$. Boolean values and `null` are always valid. The counters used for comparison of vertices are at most $\lceil \log n \rceil$.

3.3.5.2 Passing arguments and returning values

We have two methods of passing arguments to a function and returning a value. The first is through global variables and the second is through global arrays which contain an entry for every level of recursion.

The method which uses global variables is straightforward. We have global variables which are set to the arguments of the function before it is called. We denote the global variables for the arguments of a function `F` with `arg1F`, `arg2F`, and so on. During its execution a function can lookup its arguments from these global variables. It might decide to store them as local variables, but this might not always be possible, because the arguments might be invalid for the current level of recursion.

To return a value a function sets a global variable. After a return from a call of a function, the calling function decides whether it wants to store the returned value locally. We have a special assignment operator, `::=`, which assigns the return value r of a function returning through a global variable to a local variable in the following way: if r is valid for the current level of recursion, the result of the assignment is r, otherwise it is `null`.

The method which uses arrays is more subtle. Let `F` be a function which uses this method of passing arguments and returning values. This method can be used only, if all calls to `F` are recursive, i.e. all calls to `F` are `F`$(l-1,\ldots)$. We have two global arrays – one, `argF`, for passing arguments to `F` and one, `retF`, for returning a value from it. Those arrays contain exactly one entry for every possible level of recursion and each entry could be marked. Also each entry holds values which are valid for the corresponding level of recursion, so the space taken by each such array is the same as the space taken by the execution stack.

Let `H` calls `F`. If `H` uses the value returned by `F`, then before the call to `F` the entry of `retF` for the current level of recursion is marked, otherwise it is left unmarked. When `F` produces a result, it finds in `retF` the first marked entry after the entry for the current level of recursion and tries to store its result there. If the value produced is too large for the corresponding entry, `F` writes `null`. After the call to `F` returns, `H` unmarks the entry of `retF` for the current level of recursion. The only time when `H` does not use the value

returned by F is when H is actually F and the call to F is a recursive call whose result is passed back without modification, e.g. a call like **return** F($l-1,\ldots$) in the definition of F($l,\ldots$).

Similarly, if H provides arguments to F, then before the call to F, H marks the entry of argF for the current level of recursion and provides values for it. When F wants to access its arguments, it finds the first marked entry in argF after the entry for the current level of recursion and uses the values stored in the entry as its arguments. After the call to F returns, H unmarks the entry of argF for the current level of recursion. Again we have that the only time when H does not provide arguments to F is when H is F and the call to F is a recursive call whose arguments are the same as for the current call to F, e.g. a call like F($l-1,y$) in the definition of F(l,x) at a place where it is always true that $x=y$.

Since this method is used only if all calls to F are recursive, there is no danger of overwriting F's arguments or returned value. For passing arguments this method helps when we need to store an argument only at the level which generates it (where it is valid), but still allow for lower levels of recursion to access it (where it might be invalid). For returning values this method helps when we want to return a value exactly at the level which requested it. This method allows for nested calls from different levels of recursion to the same function, which the ordinary method of passing variables through global variables does not always allow.

The restriction we have on the space of a level is the reason why we chose those methods of passing arguments and returning value. The method using arrays is more unusual, and it is used only in two functions: BackLabel and BackLabelAux. The reason why we introduced it is explained in the notes for those functions.

In the code, we use F($x_1,\ldots,x_k$) to call a function F with arguments $x_1,\ldots,x_k$. If a function F takes arguments, but is called without ones, it uses the values currently located in the global variables (or the arrays) for F.

3.3.5.3 Translation of the pseudo-code to a Turing Machine

Let us address now the issue of translating the pseudo-code to a Turing Machine with a binary alphabet. Most of the details, like doing arithmetic and performing conditionals are rather straightforward, so we skip them and concentrate only on variable usage. Numerical values are represented in binary. To represent the `null` value, we use one additional bit to designate whether the value is `null` or not. We have a separate tape for each global variable (there are only constant number of them). The space taken by each global variable is $O(\log n)$.

There is a tape assigned for the stack and the head of this tape is positioned at the stack frame of the current function, i.e. at the top of the stack. The stack frame of a function contains the state to which the TM machine must return after the execution of the function (this takes a constant number of bits depending only on the TM) and the values of the local variables of the function.

The space taken by each local variable depends on the level of recursion at which the stack frame occurs. Since at a level l the value of every valid local variable is at most $v(l)$, the space $s(l)$ taken by such variable at level l is $O(\log v(l))$. This space is known to the current function, because it can be computed (by a global function) from the current level of recursion. So to use the i-th local variable the current function must move the head of the stack tape to the place where the variable is located. This place can be computed by the current function from i and $s(l)$. As discussed earlier there is a constant d such that the stack frame of the l-th level of recursion contains at most d variables. Thus the space taken by the stack frame of level l is $O(\log v(l) + \log\log n)$. The $O(\log\log n)$ appears because of comparison of vertices, as explained in section 3.3.3.

After the execution of the current function the state of the TM is restored to the value stored on the stack and the head of the stack tape is moved to the stack frame of the caller.

3.3.5.4 Preliminaries

To simplify the exposition of the algorithm, we remove the level of recursion from the argument lists of the functions. Instead we have one global variable, `level`, which contains the current level of recursion. `level` is set to $5 \cdot 2^{\lceil \log \log n \rceil} - 3$ initially. Let `F` be a function which calls a function `G` on the previous level of recursion. This task is performed by `Prev`, namely `Prev(G(...))` passes arguments to `G`, decreases the current level of recursion, calls `G`, and upon return from `G` increases the current level of recursion. This is the only way that the current level of recursion is changed – all functions can lookup the value of `level`, but none of them changes it. We denote the current level of recursion with cl.

The global variable `currvertex` contains the current vertex cv.

In the following, T denotes the hooking tree of the current vertex in $\mathcal{C}_{cl-1}$. A hooking tree in $\mathcal{C}_{cl-1}$ is called contractable, if its degree is at most $\mathrm{dexp}(Arg_{cl} + 1)$. A value is valid, if it is most $v(cl)$. A vertex v is valid if its degree $\delta_{cl-1}(v)$ is at most $v(cl)$. For the definition of Op_l, Arg_l, and dexp see the beginning of section 3.2.4.

We use the following observation. It follows from Observation 3.3.2 and the correctness of the functions mentioned in it.

Observation 3.3.5. *1) If* `TreeSize` $\neq$ `null`*, then all moves of* `TreeForward`(i) *are possible, for* $i \geq 0$*. 2) If* `MoveValid`(i) *is true, then the result of* `Move`(i) *is not* `null` *and valid, otherwise* $\eta_{cl-1}(cv, i)$ *is invalid.*

By this observation `TreeSize` and `MoveValid` serve as "safeguard" checks for forward moves. Thus before making a forward move to change the current vertex, if we want to be able to return, e.g. in an `after` statement, we always first make sure that the forward move returns a valid result. In this case we say that the forward move is valid.

All functions, except `BackLabel` and `BackLabelAux`, take arguments and return values through global variables.

Every function is preceded by paragraphs which give its specification. Also notes are made on the definition and correctness of the function, and on the validity of its local variables. In the notes we use interchangeably the name of a local variable, given in fixed font, and its value.

3.3.5.5 Important functions

Global function `ArgOp`. **Input** l. **Output** $2 \cdot Arg_l + \varepsilon_l$, where ε_l is 0, if $Op_l = Hook$, and 1, otherwise. **Assumes** $1 \leq l \leq 5 \cdot 2^{\lceil \log \log n \rceil} - 3$.

```
global function ArgOp
    k := ⌈log log n⌉;
    while true
        if k = 0 then return 2 · (l − 1) + (l − 1);
        else
            if l ≤ 2 then return 2 · k + (l − 1);
            else
                if l = 5 · 2^k − 3 then return 2 · (k + 1) + 1;
                else
                    if l ≥ 5 · 2^(k−1) then l := l − 5 · 2^(k−1) + 1;
                    else l := l − 2;
                    k := k − 1;
```

Global functions `Valid`, `ContractDegree`, and `Contraction`. **Input** None. **Output** Correspondingly $2 \cdot \mathrm{dexp}(Arg_{cl} + 2)$, $\mathrm{dexp}(Arg_{cl} + 1)$, and whether $Op_{cl} = Contract$.

```
global function Valid
    return 2 · dexp(2 + ArgOp(level) div 2);

global function ContractDegree
    return dexp(1 + ArgOp(level) div 2);

global function Contraction
    return ArgOp(level) mod 2 = 1;
```

Statement "**after** $\mathtt{M}_1, \mathtt{M}_2, \ldots, \mathtt{M}_k$ **do** `B`". **Input** $\mathtt{M}_1, \ldots, \mathtt{M}_k$ – path description relative to the current vertex, `B` some instruction(s). **Output** Moves the current vertex

according to the forward moves in $\mathtt{M}_1, \ldots, \mathtt{M}_k$, executes B, and finally restores the original current vertex. **Assumes** All forward moves are valid. **Local variables** $\mathtt{l}_1, \ldots, \mathtt{l}_k$.

define after $\mathtt{M}_1$, $\mathtt{M}_2$, ..., $\mathtt{M}_k$ **do** B

    $\mathtt{l}_1$:= $\mathtt{M}_1$; $\mathtt{l}_2$:= $\mathtt{M}_2$; ...; $\mathtt{l}_k$:= $\mathtt{M}_k$;
    B;
    Reverse($\mathtt{M}_k$, $\mathtt{l}_k$); ...; Reverse($\mathtt{M}_2$, $\mathtt{l}_2$); Reverse($\mathtt{M}_1$, $\mathtt{l}_1$);

Operators <, =, and $<_d$. **Input** $\mathtt{P}_1$ and $\mathtt{P}_2$ – path descriptions relative to the current vertex. **Output** Let v_1 and v_2 be the end vertex of $\mathtt{P}_1$ and $\mathtt{P}_2$, correspondingly. The three operators check correspondingly, whether $v_1 < v_2$, $v_1 = v_2$, and $v_1 <_d v_2$. **Assumes** All forward moves are valid. $<_d$ assumes also that v_1 and v_2 are valid. **Local variables** i, $\mathtt{b}_1$, and $\mathtt{b}_2$ for < and =. $\mathtt{d}_1$ and $\mathtt{d}_2$ for $<_d$. **Notes** Bit(s,t) returns the s-th most significant bit of t. $\mathtt{b}_1$ and $\mathtt{b}_2$ are single bits. i in the definition of those operators are the only variables which always take space $\Theta(\log\log n)$.

define $\mathtt{P}_1$ < $\mathtt{P}_2$

    for i := 1 **to** $\lceil \log \mathtt{n} \rceil$ **do**
        after $\mathtt{P}_1$ **do** $\mathtt{b}_1$:= Bit(i, currvertex);
        after $\mathtt{P}_2$ **do** $\mathtt{b}_2$:= Bit(i, currvertex);
        if $\mathtt{b}_1 \neq \mathtt{b}_2$ **then return** $\mathtt{b}_1 < \mathtt{b}_2$;
    return false;

define $\mathtt{P}_1$ = $\mathtt{P}_2$

    for i := 1 **to** $\lceil \log \mathtt{n} \rceil$ **do**
        after $\mathtt{P}_1$ **do** $\mathtt{b}_1$:= Bit(i, currvertex);
        after $\mathtt{P}_2$ **do** $\mathtt{b}_2$:= Bit(i, currvertex);
        if $\mathtt{b}_1 \neq \mathtt{b}_2$ **then return** false;
    return true;

define $\mathtt{P}_1$ $<_d$ $\mathtt{P}_2$

    after $\mathtt{P}_1$ **do** $\mathtt{d}_1$ ::= Prev(Degree);

```
    after P2 do d2 ::= Prev(Degree);
    return (d1 < d2) or (d1 = d2 and P1 < P2);
```

3.3.5.6 Status functions

Functions `Done`, `Active`, and `Inactive`. **Input** None. **Output** True, correspondingly, iff $cv \in D_{cl}$, $cv \in A_{cl}$, and $cv \in I_{cl}$. **Assumes** None. **Local variables** None.

```
function Done
    return (level = 0 and Degree = 0) or
        (level > 0 and not Contraction and Prev(Done)) or
        (level > 0 and Contraction and
            (Prev(Done) or Root ≠ 0 or Degree = 0));

function Active
    return (level = 0 and Degree ≠ 0) or
        (level > 0 and not Contraction and Prev(Active)) or
        (level > 0 and Contraction and
            not Done and TreeSize ≠ null);

function Inactive
    return (level > 0 and not Contraction and Prev(Inactive)) or
        (level > 0 and Contraction and
                not Done and TreeSize = null);
```

3.3.5.7 Hooking

Function `Hook`. **Input** None. **Output** $H_{cl}(cv)$. **Assumes** None. **Local variables** $\mathtt{d}_1$, $\mathtt{d}_2$, `i`, `j`, and `m`. **Notes** `Hook` is defined as given in section 3.2.3. In line 9 we use 1) of Observation 3.3.2 to deduce that $\eta_{cl-1}(cv, \mathtt{i}) \in I_{cl-1}$. At line 5 $cv \in A_{cl-1}$ and hence $\mathtt{d}_1$ is valid and non-`null`. So `i` and `m` are also valid. At line 12 we have that $\eta_{cl-1}(cv, \mathtt{i}), \eta_{cl-1}(cv, \mathtt{m}) \in A_{cl-1}$. At line 14 all neighbors of cv are in A_{cl-1}. Hence $\mathtt{d}_2$ and `j` are valid.

```
   function Hook
 1     if level = 0 then return 0;
 2     if not Contraction and Prev(Inactive) then return Prev(Hook);
 3     if Contraction and Inactive then return Prev(Hook);
 4     if Prev(Done) or Contraction then return 0;

 5     d1 ::= Prev(Degree);
 6     m := 0;
 7     for i := 1 to d1 do
 8         // if the i-th neighbor is inactive then hook to it
 9         if not MoveValid(i) then return i;
10         after Move(i) do fl := Prev(Inactive);
11         if fl then return i;

           // otherwise check if it is bigger than
           // the current biggest active neighbor
12         if Move(m) <d Move(i) then m := i;
13     if m > 0 then return m;

14     for i := 1 to d1 do
15         after Move(i) do d2 ::= Prev(Degree);
16         for j := 1 to d2 do
               // if the j-th neighbor of the i-th neighbor is inactive
               // hook to i
17             after Move(i) do fl := MoveValid(j);
18             if not fl then return i;
19             after Move(i), Move(j) do fl := Prev(Inactive);
20             if fl then return i;
```

```
            // otherwise hook to i, if its j-th neighbor is bigger
            // than currvertex
21          if Current <_d (Move(i), Move(j)) then m := i;
22      return m;
```

Function `IsHooked`. **Input** i. **Output** Let $(v, j) = \mu_{cl-1}(cv, i)$ and $h = H_{cl-1}(v)$. `IsHooked` is true iff $h = j$. **Assumes** $cl \geq 1$, $0 \leq i \leq \delta_{cl-1}(cv)$, and cv is valid or $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$. **Local variables** `i`, `j`, and `h`. **Notes** `i` is valid because of line 2, `j` is valid because of the assignment in line 5, and `h` is valid because of the assignment in line 9.

```
   function IsHooked
 1     if argIsHooked = 0 then return Prev(Hook) = 0;
 2     if argIsHooked > Valid then return Prev(IsHooked);
 3     i := argIsHooked;
 4     if Prev(Degree) > Valid then return Prev(IsHooked(i));

 5     j := Prev(BackLabel(i));
 6     if j = null then
 7         if Prev(Active) then return false;
 8         return Prev(IsHooked(i));
 9     after Move(i) do h ::= Prev(Hook);
10     return h = j;
```

Correctness We prove the correctness of `IsHooked` by induction on cl. Notice that for $cl = 1$, the checks in lines 2, 4, and 6 all fail because at level 1 all vertices are valid, and we compare `h` and `j` in line 10.

First consider the case when $cv \in A_{cl-1}$. In this case cv is valid by 2) of Observation 3.3.2. If `j` is invalid, then $v \in I_{cl-1}$, and it is not hooked to cv (otherwise $cv \in I_{cl-1}$ by Definition 3.2.4). We catch this in line 7. If `j` is valid, then in line 10 we check whether it is equal to `h`.

Let now $cv \in I_{cl-1}$ and $v \in A_{cl-1}$. Since v is valid, cv is valid also, because this follows from v valid, and cv valid or $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$. So `i`, `j` and `h` are valid and we can compare `h` and `j` in line 10.

Assume now that $cv, v \in I_{cl-1}$. In this case the only way `IsHooked` returns an answer without calling recursively is in line 10, then `j` is valid and we have compared it to `h`. Notice now that, if `IsHooked` calls itself recursively then $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$. This is true for the calls in lines 2 and 4, because then cv is invalid. For the call in line 8 this is true, because cv is valid and v is invalid. Since $cv, v \in I_{cl-1}$, we have that $cl \geq 2$, $\delta_{cl-2}(cv) = \delta_{cl-1}(cv)$, $\delta_{cl-2}(v) = \delta_{cl-1}(v)$, $(v, \mathtt{j}) = \mu_{cl-2}(cv, \mathtt{i})$, and $\mathtt{h} = H_{cl-2}(v)$. Thus the correctness in this case is ensured by the inductive hypothesis.

3.3.5.8 Exploration walk

The functions in this section come from the definition of a hooking forest of a configuration given in section 3.2.2. Throughout it is assumed that $cl \geq 1$ and $Op_{cl} = Contract$. T is the hooking tree of cv in $\mathcal{C}_{cl-1}$.

Function `TreeDegree`. **Input** None. **Output** If cv is valid, `TreeDegree` returns $\delta_T(cv)$, otherwise it returns `null`. **Assumes** None. **Local variables** `i`, `d`, and `td`. **Notes** In line 5 we use that cv is valid to apply the correctness of `IsHooked`. `d` is valid because of the assignment in line 1, and `i` and `td` are valid because at line 3 cv is valid.

```
   function TreeDegree
       // if currvertex is invalid return null
1      d ::= Prev(Degree);
2      if d = null then return null;

3      td := 0;
       // count the number of neighbors which are hooked to currvertex
4      for i := 1 to d do
5          if IsHooked(i) then td := td + 1;
       // add 1 if currvertex did not hook to itself
```

```
6     if Prev(Hook) ≠ 0 then td := td + 1;
7     return td
```

Function `TreeMove`. **Input** i. **Output** Let $(v, j) = \mu_T(cv, i)$. If a move along (cv, i) is possible, i.e. v is valid, `TreeMove` returns j and moves the current vertex to v, otherwise it does not change the current vertex and returns `null`. **Assumes** $0 \leq i \leq \delta_T(cv)$, cv is valid. **Local variables** `i`, `j`, `k`, `l`, `d`, $\mathtt{d}_1$, and `r`.

Notes Lines 2-7 convert from the label `i` of a tree-edge e to a label `j` of an edge in the graph. In line 5 we use that cv is valid to apply `IsHooked`. Lines 8-10 handle the case when v is invalid. Lines 11-26 compute the tree back-label `r` of e and move the current vertex to v. Lines 11-13 handle the case, when v hooked to the current vertex. Lines 14-26 handle the case when e is the hooking edge of the current vertex. In lines 19-21 we use that, if the `k`-th neighbor of v is invalid, then it is not cv, because cv is valid.

`i`, `j`, `l`, and `d` are valid, because cv is valid (the assignments in lines 3 and 7 are non-`null`). $\mathtt{d}_1$ is valid because of the assignment in line 9. `k` and `r` are valid because at line 14 v is valid.

```
   function TreeMove
       i := argTreeMove;
1      if i = 0 then return 0;

       // convert from tree-edge label to graph-edge label
2      l := i;
3      d ::= Prev(Degree);
4      for j := 1 to d do
5          if IsHooked(j) then l := l - 1;
6          if l = 0 then break;
7      if j > d then j ::= Prev(Hook);

       // if the new vertex is invalid return null
```

```
 8     if not MoveValid(j) then return null;
 9     after Move(j) do d1 ::= Prev(Degree);
10     if d1 = null then return null;

11     if i < TreeDegree or (i = TreeDegree and Prev(Hook) = 0) then
           // e goes to a neighbor which hooked to currvertex
12         Move(j);
13         return TreeDegree;

       // e is the hooking edge of currvertex

       // compute the tree back-label
14     r := 1;
15     for k := 1 to d1 do
16         after Move(j) do fl := IsHooked(k);
17         if not fl then continue;
           // enumerate all the edges with which neighbors
           // of the new vertex hooked to it

18         after Move(j) do fl := MoveValid(k);
19         if not fl then
20             // the k-th neighbor of the new vertex is invalid
21             r := r + 1;
22             continue;

           // check if this is the edge with which currvertex hooked
           // to the new vertex
23         if (Move(j), Move(k)) = Current then break;
24         r := r + 1;
```

```
        // move to the new vertex
25      Move(j);
        // return the tree back-label
26      return r;
```

Function `TreeForwardStep`. **Input** i. **Output** Let $(v, j) = \Gamma_{T,1}(cv, i)$. If a move along (cv, i) is possible, i.e. v is valid, then `TreeForwardStep` returns j and moves the current vertex to v, otherwise it returns `null` and does not change the current vertex. **Assumes** $0 \leq i \leq \delta_T(cv)$, cv is valid. **Local variables** `j`.

```
function TreeForwardStep
    j := TreeMove(argTreeForwardStep);
    if j = null then return null;
    j := j + 1;
    if j > TreeDegree then j := 1;
    return j;
```

Function `TreeForward`. **Input** i. **Output** If $\Gamma_{T,i}(v, 1)$ ends in (v, j), then `TreeForward` returns j and moves the current vertex to v. **Assumes** cv and i are valid, all moves of $\Gamma_{T,i}(v, 1)$ are possible. **Local variables** `i`, `j`, and `k`.

```
function TreeForward
    i := argTreeForward;
    j := 1;
    for k := 1 to i do
        j := TreeForwardStep(j);
    return j;
```

Function `TreeBack`. **Input** i. **Output** If $\Gamma_{T,i}(v, j)$ ends in v, then `TreeBack` moves the current vertex to v. **Assumes** $0 \leq j \leq \delta_T(cv)$, cv and i are valid, all moves of $\Gamma'_{T,i}(v, j)$ are possible. **Local variables** `i`, `j`, `k`. **Notes** `TreeBack` is defined in a way similar to `TreeForward` using a function `TreeBackStep`.

```
function TreeBackStep
    j := argTreeBackStep - 1;
    if j = 0 then j := TreeDegree;
    return TreeMove(j);

procedure TreeBack
    i := arg1TreeBack; j := arg2TreeBack;
    for k := 1 to i do
        j := TreeBackStep(j);
```

Function `TreeSize`. **Input** None. **Output** $2(\texttt{size}(T)-1)$, if T is contractable, and `null`, otherwise. **Assumes** None. **Local variables** `i`, $\texttt{i}_1$, $\texttt{k}_1$, $\texttt{k}_2$, `d`, and `td`.

Notes $2(\texttt{size}(T)-1)$ is the length of the exploration walk given in Proposition 3.2.1. The method to compute it is provided by the same proposition, i.e. `TreeSize` incrementally finds (line 6-22) the length of a walk which visits the current vertex exactly the number of times equal to its tree-degree plus 1 (the check is done in lines 13 and 14). Before increasing the length of the walk, we first makes sure that the next move is possible (lines 7-12). If it is not, `TreeSize` returns `null`. This is correct, because if T has an invalid vertex, it is not contractable (`Valid` $>$ `ContractDegree`). Otherwise it checks, if the walk went back to the starting vertex and returns, if the starting vertex was visited sufficiently many times. Also when `TreeSize` visits a vertex for the first time (lines 15-17), it adds its degree to the current total degree of T and returns `null`, if the total degree becomes larger than `ContractDegree` (lines 18-21).

The condition of the loop in line 6, makes sure that the current length `i` of the exploration walk is valid. If it is not, line 23 returns `null` because T is uncontractable. This is correct because on one hand $\text{size}(T) \leq \deg(T)$ (at line 6, T has at least one edge) and on the other, by Proposition 3.2.1, exploration walk of length $2(\text{size}(T)-1)$ visits all vertices of a tree of size $\text{size}(T)$ and returns to the starting vertex sufficiently many times. Since `Valid` ≥ 2 `ContractDegree`, if the length of the exploration walk becomes

bigger than Valid, then size(T) > ContractDegree, so deg(T) > ContractDegree and T is uncontractable.

i and i_1 are valid because of the condition of the loop in line 6. k_1 is valid because of the assumption that all vertices visited by the exploration walk of length i − 1 in line 7 are valid. k_2 is valid because of the condition on the output of TreeForwardStep in line 8. d is valid because of the condition on the output of TreeDegree in line 1. td is valid because at line 20 both td and d_1 are at most ContractDegree, and since Valid ≥ 2 ContractDegree, the addition in line 20 produces a valid result.

```
   function TreeSize
 1     d := TreeDegree;
       // if currvertex is invalid, then the tree is uncontractable
 2     if d = null then return null;
 3     if d = 0 then return 0;

 4     i := 1;
 5     td := 0;
 6     while i ≤ Valid do
           // check if we can make one more step from the exploration walk
 7         k1 := TreeForward(i-1);
 8         k2 := TreeForwardStep(k1);
 9         if k2 = null then
               // if we cannot then the tree is uncontractable
10             TreeBack(i-1, k1);
11             return null;
12         TreeBack(i, k2);

           // check if we have visited the starting vertex sufficiently
           // many times
13         if TreeForward(i) = Current then d := d - 1;
```

```
        // if yes, then return the current length of the exploration
        // walk
14      if d = 0 then return i;

        // check if the end of the current exploration walk is visited
        // for the first time
15      for i1 := 0 to i - 1 do
16          if TreeForward(i) = TreeForward(i1) then break;
17      if i1 = i then
            // if it is, add its degree to the total degree
18          after TreeForward(i) do d1 ::= Prev(Degree)
            // if the total degree becomes too large then the tree
            // is uncontractable
19          if d1 > ContractDegree then return null;
20          td := td + d1;
21          if td > ContractDegree then return null;

        // increase the length of the exploration walk by 1
22      i := i + 1;
23   return null;
```

Function `Root`. **Input** None. **Output** If T is uncontractable or $cv \in D_{cl-1}$, then `Root` returns 0, otherwise it returns the index of the first occurrence of root(T) in the exploration walk on T starting from $(cv, 1)$. **Assumes** None. **Local variables** `d` and `i`.

Notes According to the definition of root(T) given in section 3.2.2, `Root` enumerates the vertices of T using the exploration walk starting from $(cv, 1)$ (lines 3-5) and finds the first vertex which hooked to itself (line 4). `d` is valid because of the assignment in line 1, and `i` is valid because at line 3 T is contractable.

function `Root`

```
      // check if T is contractable
1     d := TreeSize;
2     if d = null or Prev(Done) then return 0;

      // if it is, find the vertex in it which hooked to itself
3     for i := 0 to d-1 do
4         after TreeForward(i) do fl := (Prev(Hook) = 0);
5         if fl then return i;
```

3.3.5.9 Contraction

The definitions of the functions in this section come from the definition of the contraction operation given in section 3.2.3 T is the hooking tree of cv in $\mathcal{C}_{cl-1}$.

Function `IsEdge`. **Input** i and j. **Output** If v is the end vertex of $\Gamma_{T,i}(cv, 1)$, then `IsEdge` returns true iff (v, j) is a remaining edge of T (see the definition in section 3.2.2). **Assumes** i is valid, $0 \leq j \leq \delta_{cl-1}(v)$, $cv = \text{root}(T)$ and T is contractable. **Local variables** `i`, `j`, $\texttt{j}_1$, `k`, $\texttt{k}_1$, `d`, $\texttt{d}_1$, and $\texttt{d}_2$.

Notes The definition of `IsEdge` follows exactly the definition of the remaining edges of T given in section 3.2.2. Let $e = (v, j)$, $w = \eta_{cl-1}(e)$, and T' be the hooking tree of w in $\mathcal{C}_{cl-1}$. Lines 2-5 check, if T' is contractable. Line 7 checks, if e is internal. Let u be the `k`-th vertex in the exploration walk of T starting from cv. Because of lines 9 and 10, at line 12 $(u, \texttt{j}_1)$ is an edge before e in the enumeration of the remaining edges of T given in section 3.2.2. Let $u' = \eta_{cl-1}(u, \texttt{j}_1)$. Lines 14-16 check whether u' is in T'. In line 13 we use that, if the hooking tree of u' in $\mathcal{C}_{cl-1}$ is uncontractable, then u' is not from T', because at this point T' is contractable.

`i`, `j`, and `d` are valid because T is contractable. $\texttt{d}_2$ and $\texttt{k}_1$ are valid, because at line 6 T' is contractable. Lines 9 and 10 ensure the validity of $\texttt{d}_1$ and $\texttt{j}_1$.

```
   function IsEdge
       i := arg1IsEdge; j := arg2IsEdge;
1      d := TreeSize;
```

```
    // if T' is uncontractable, then e remains
2   after TreeForward(i) do fl := MoveValid(j);
3   if not fl then return true;
4   after TreeForward(i), Move(j) do d2 := TreeSize;
5   if d2 = null then return true;

    // T' is contractable
6   for k := 0 to d-1 do
        // e does not remain, if it is an internal edge
7       if TreeForward(k) = (TreeForward(i), Move(j)) then
            return false;
8       if k > i then continue;

9       if k = i then d1 := j - 1;
        else
10          after TreeForward(k) do d1 ::= Prev(Degree);

        // e does not remain, if it is not the first edge
        // from T to T'
11      for j1 := 1 to d1 do
12          after TreeForward(k) do fl := MoveValid(j1);
13          if not fl then continue;

14          for k1 := 0 to d2-1 do
15              if (Treeforward(i), Move(j), TreeForward(k1)) =
                       (TreeForward(k), Move(j1)) then
16                  return false;
17  return true;
```

Statement "**after** P **for every edge** (i, j) **do** B". **Input** P a path description

relative to the current vertex, `i` and `j` names of local variables using this statement, `B` instruction(s) which might depend on the variables `i` and `j`. **Output** Let v be the vertex with path description `P` and T' is its hooking tree in $\mathcal{C}_{cl-1}$. This statement executes `B` for all possible values of $(\mathtt{i}, \mathtt{j})$ such that $(u, \mathtt{j})$ is a remaining edge of T', where u is the end vertex of $\Gamma_{T,\mathtt{i}}(v, 1)$. **Assumes** $cl \geq 1$, all forward moves in `P` are valid, T' is contractable and $v = \text{root}(T')$. **Local variables** $\mathtt{i}_1$, $\mathtt{d}_1$, $\mathtt{d}_2$.

Notes Lines 3-5 check, if this is the first time the exploration walk on T' visits the `i`-th vertex v. If so, lines 7-9 enumerate the remaining edges of T' incident to v. All local variables, and `i` and `j`, are valid because T' is contractable.

```
   define after P for every edge (i, j) do B
 1     after P do d1 := TreeSize;
 2     for i := 0 to d1 - 1 do
           // visit only once every vertex of T'
 3         for i1 := 0 to i - 1 do
 4             if (P, TreeForward(i)) = (P, TreeForward(i1)) then break;
 5         if i1 < i then continue;

 6         after P, TreeForward(i) do d2 ::= Prev(Degree);
 7         for j := 1 to d2 do
 8             after P do fl := IsEdge(i, j);
 9             if not fl then continue;

               // if (i, j) is a remaining edge, then execute B
10             B;
```

Function `Degree`. **Input** None. **Output** $\delta_{cl}(cv)$. **Assumes** None. **Local variables** `i`, `j`, and `td`. **Notes** To obtain the degree of the current vertex, we just enumerate all remaining edges of T. If T is not contractable, then, by definition, the degree comes from a previous level (line 2). Line 2 handles the case when $cv \in I_{cl}$, and line 3 the case when $cv \in D_{cl}$. All local variables are valid because at line 3 T is contractable.

`GraphDegree` returns the degree of the current vertex in the input graph G.

```
   function Degree
1      if level = 0 then return GraphDegree;
2      if not Contraction or TreeSize = null then return Prev(Degree);
3      if Prev(Done) or Root ≠ 0 then return 0;

4      td := 0;
5      after Current for every edge (i,j) do td := td + 1;
6      return td;
```

Procedure `Neighbor`. **Input** i. **Output** Moves the current vertex to $\eta_{cl}(cv, i)$. **Assumes** $0 \leq i \leq \delta_{cl}(cv)$. **Local variables** `l`, `j`, `i`, and `d`.

Notes The definition of `Neighbor` follows the definitions in section 3.2.2. First we make sure that T is contractable (lines 4 and 10). If not, then we call recursively. Otherwise, i is the index of a remaining edge e of T, and we locate e and move along it (lines 14-20). Once we move along e, we move the current vertex to the representative of the new current vertex, i.e. the root of the new current hooking tree T', if it is contractable (lines 6, 12, and 19).

`i` is valid because at line 8 `argNeighbor` is valid, and the other local variables are valid because at line 14 T is contractable.

`GraphNeighbor`(i) moves the current vertex to its i-th neighbor in the input graph G.

```
   procedure Neighbor
1      if level = 0 then GraphNeighbor(argNeighbor);
       if not Contraction then Prev(Neighbor);
2      // handle the self-loop case
3      if argNeighbor = 0 then return;

4      if argNeighbor > Valid then
```

```
        // if T is uncontractable, call recursively
 5      Prev(Neighbor(argNeighbor));
        // if T' is contractable, move to its root
 6      if TreeSize ≠ null then TreeForward(Root);
 7      return;
 8  i := argNeighbor;

 9  d := TreeSize;
10  if d = null then
        // T is uncontractable
11      Prev(Neighbor(i));
12      if TreeSize ≠ null then TreeForward(Root);
13      return;

    // T is contractable
14  after Current for every edge (l, j) do
15      i := i - 1;
        // check if (l, j) is e
16      if i > 0 then continue;

        // move to e and then along e
17      TreeForward(l);
18      Prev(Neighbor(j));
        // move to the root of T'
19      if TreeSize ≠ null then TreeForward(Root);
20      return;
```

Function BackLabel. **Input** i. **Output** $\beta_{cl}(cv, i)$. Uses the array method described in section 3.3.5 of taking arguments and returning values. **Assumes** $0 \leq i \leq \delta_{cl}(cv)$. **Local variables** l, j, $\mathtt{j}_1$, k, $\mathtt{k}_1$, i, d, nd, and r.

Notes The first case of `BackLabel` is when T is contractable. In this case we find the remaining edge e of T with index i (lines 12-14). Let $v = \eta_{cl-1}(e)$ and T' be the hooking tree of v in $\mathcal{C}_{cl-1}$. If T' is uncontractable, then we call recursively, because in this case the back-label comes from the previous level of recursion (line 22). Otherwise we have to find the index `nd` of the first remaining edge e' of T' which goes from T' to T (lines 24-32). This is the new back-label. To find the index of e', first we find the root of T' (line 20) and then enumerate all remaining edges of T' (lines 25-32). For each remaining edge of T' we check if it goes to T (line 30-32). In lines 27-29, we use that, if a remaining edge of T' goes to an uncontractable hooking tree, then it does not go to T, because at this point T is contractable. The case when T is uncontractable is handled by `BackLabelAux` (lines 5 and 10).

`i` is valid because of line 4. `d` is valid because of the assignment in line 8. `l`, `j`, and $\texttt{k}_1$ are valid because at line 12 T is contractable. `r` is valid because of line 20. $\texttt{j}_1$, `k`, and `nd` are valid because at line 24 T' is contractable.

The recursive call in line 22 does not assign the returned value to a local variable, i.e. this call returns a value at some higher level of recursion, depending on the array for returning values of `BackLabel`. This call is the reason why `BackLabel` returns through an array instead of a global variable. The conventional thing to do is to store the result of this call locally, and once the `after` statement has restored the original current vertex, return the stored value. This does not work for us, because the value returned from the recursive call might be invalid. Instead, using that the only reason why we store the returned value is to pass it back, when `BackLabel` produces a result we let it store the result at the level at which it is requested. This works because `BackLabel` is always called on the previous level of recursion.

`GraphBackLabel`(i) returns the back-label of the i-th edge incident to cv in the input graph G.

```
   function BackLabel
1      if level = 0 then return GraphBackLabel(argBackLabel);
2      if not Contraction then return Prev(BackLabel);
```

```
3     if argBackLabel = 0 then return 0;

      // if currvertex is invalid call BackLabelAux
4     if argBackLabel > Valid then
5         BackLabelAux;
6         return;
7     i := argBackLabel;

      // if T is uncontractable call BackLabelAux
8     d := TreeSize;
9     if d = null then
10        BackLabelAux;
11        return;

      // T is contractable
12    after Current for every edge (l, j) do
13        i := i - 1;
          // find e
14        if i > 0 then continue;

15        after TreeForward(l) do
16            fl := MoveValid(j);
17            if fl then
18                after Move(j) do
19                    fl := (TreeSize ≠ null);
20                    if fl then r := Root;

21        if not fl then
              // if T' is uncontractable call recursively
```

```
22              after TreeForward(l) do Prev(BackLabel(j));
23              return;

        // T' is contractable
24          nd := 0;
            // find the first edge of T' which goes to T and return
            // its index
25          after TreeForward(l), Move(j), TreeForward(r)
                    for every edge (k, j_1) do
26              nd := nd + 1;
27              after TreeForward(l), Move(j),
                            TreeForward(r), TreeForward(k) do
28                  fl := MoveValid(j_1);
29              if not fl then continue;

30              for k_1 := 0 to d-1 do
31                  if (TreeForward(l), Move(j),
                                TreeForward(r), TreeForward(k), Move(j_1))
                            = TreeForward(k_1) then
32                      return nd;
```

Function BackLabelAux. **Input** i. **Output** $\beta_{cl}(cv)$. To take argument and return value BackLabelAux uses the arrays of BackLabel. **Assumes** $0 \leq i \leq \delta_{cl}(cv)$, T is uncontractable. **Local variables** l, j, k, bl, nbl, r, and d.

Notes The definition of BackLabelAux follows the definitions given in section 3.2.2 when T is uncontractable. Let v and T' be as in the note for BackLabel. If T' is uncontractable, the back-label is inherited from the previous level of recursion, so we call BackLabel recursively (lines 3 and 10). Otherwise at line 12, T' is contractable, the current vertex is v (because of line 5), and bl is the back-label of e (because of line 1). So we have to find the index of $(v, \texttt{bl})$ in T' ($(v, \texttt{bl})$ is a remaining edge of T' because T

is uncontractable). Line 12 finds the root of T', and lines 13 and 14 find the index `k` of the first occurrence of v in the exploration walk of T' starting from its root. Lines 16-20 enumerate the remaining edges of T' until we find $(v, \texttt{bl})$.

`bl` is valid by the assumption for the return convention of `BackLabel` for line 1. `d` is valid because of the assignment in line 6. `r`, `l`, `j`, `k`, and `nbl` are valid because at line 12 T' is contractable.

Just like for `BackLabel`, the calls to `BackLabel` in lines 3 and 10 return values at some higher level of recursion. The calls to `BackLabel` in lines 1, 3, and 10 do not have arguments – by convention this means that the argument to `BackLabel` comes from a higher level of recursion.

The case when T is uncontractable is the reason why the argument to `BackLabel` is passed through an array instead of a global variable. More precisely, the problem is when the current vertex is invalid, then the argument i to `BackLabel`, which is the label of an edge incident to cv, might be invalid and storing it locally will be impossible. In this case we still want to be able to use the value of i after calling functions which can potentially change the value of a global argument to `BackLabel`. The decision is to let the value of the argument stay at the level which produced it, because it certainly is valid for this level. For this to work, it is important that the value of the argument stored in the array is not changed while processing the call to `BackLabel`. Fortunately this does not happen, because `BackLabel` is always called on the previous level of recursion.

```
   function BackLabelAux
1      bl := Prev(BackLabel);
2      if bl = null then
           // if T' is uncontractable call recursively
3          Prev(BackLabel);
4          return;

       // move along e
5      Prev(Neighbor(argBackLabel));
```

```
 6     d := TreeSize;
 7     if d = null then
 8         // if T' is uncontractable go back and call recursively
 9         Prev(Neighbor(bl));
10         Prev(BackLabel);
11         return;

       // T' is contractable
12     r := Root;
       // find the index of the first occurrence of v in
       // the exploration walk of T' starting from r
13     for k := 0 to d - 1 do
14         if TreeForward(r), TreeForward(k) = Current then break;

       // compute the new back-label
15     nbl := 0;
16     after TreeForward(r) for every edge (l, j) do
           // increase the new back-label by one
           // for every edge that happens before e
17         nbl := nbl + 1;

18         if l = k and j = bl then
               // if we are at (v,bl) move back and return
               // the new back-label
19             Prev(Neighbor(bl));
20             return nbl;
```

Function `Move`. **Input** i. **Output** Let $(v, j) = \mu_{cl-1}(cv, i)$. `Move` returns j and moves the current vertex to v. **Assumes** $cl \geq 1$, $0 \leq i \leq \delta_{cl-1}(cv)$, i and j valid. **Local variables** `i` and `j`, which are valid by the assumption about the argument of `Move`.

function `Move`

```
i := argMove;
j := Prev(BackLabel(i));
Prev(Neighbor(i));
return j;
```

Function `MoveValid`. **Input** i. **Output** True iff $\beta_{cl-1}(cv, i)$ is valid. **Assumes** $cl \geq 1$, $0 \leq i \leq \delta_{cl-1}(cv)$, i valid. **Local variables** None.

function `MoveValid`

```
return Prev(BackLabel(argMoveValid)) ≠ null;
```

3.3.5.10 Solving undirected st-connectivity

Procedure `MoveToRep`. **Input** None. **Output** Moves the current vertex to $\text{rep}_{R_{cl}}(cv)$. **Assumes** None. **Local variables** None.

procedure `MoveToRep`

```
if level > 0 then
    Prev(MoveToRep);
    if Contraction and TreeSize ≠ null then TreeForward(Root);
```

Global function `Connected`. **Input** s and t. **Output** True iff s and t are connected in G.

global function `Connected`

```
level := 5 · 2^⌈log log n⌉ − 3;
currvertex := s; MoveToRep;
r := currvertex;
currvertex := t; MoveToRep;
return r = currvertex;
```

Bibliography

[AB01] N. Alon and R. Beigel. Lower bounds for approximations by low-degree polynomials over $\mathbb{Z}_m$. In *Proceedings of the 16th IEEE Conference on Computational Complexity*, pages 184–187, 2001.

[Ajt83] M. Ajtai. Σ_1^1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[AKL+79] R. Aleliunas, R. Karp, R. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[AKS02] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P, 2002.

[All89] E. Allender. A note on the power of threshold circuits. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 580–584, 1989.

[AS83] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for ultracomputer and PRAM. In *Proceedings of the International Conference on Parallel Processing*, pages 175–179, 1983.

[ATSWZ97] R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. $SL \subseteq L^{\frac{4}{3}}$. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 230–239, 1997.

[BM89] R. Beigel and A. Maciel. Upper and lower bounds for some depth-3 circuit classes. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 580–584, 1989.

[BNBK+89] A. Bar-Noy, A. Borodin, M. Karchmer, N. Linial, and M. Werman. Bounds on universal sequences. *SIAM Journal on Computing*, 18(2), 1989.

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

[Bou05] J. Bourgain. Estimation of certain exponential sums arising in complexity theory. *C.R. Acad. Sci. Paris*, Ser. I 340, 2005.

[Bri87] M. Brigland. Univeral traversal sequences for paths and cycles. *Journal of Algorithms*, 8(3), 1987.

[CGT96] J.-Y. Cai, F. Green, and T. Thierauf. On the correlation of symmetric functions. *Mathematical Systems Theory*, 29:245–258, 1996.

[CHL01] K. Chong, Y. Han, and T. Lam. On the parallel time complexity of undirected connectivity and minimum spanning trees. *Journal of the ACM*, 48(2):297–323, 2001.

[CL95] K. Chong and T. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18(3):378–402, 1995.

[CLC82] F. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25:659–666, 1982.

[CV86] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[FSS84] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial hierarchy. *Mathematical Systems Theory*, 17:13–27, 1984.

[Gaz86] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 492–501, 1986.

[Gol95] M. Goldmann. A note on the power of majority gates and modular gates. *Information Processing Letters*, 53:321–327, 1995.

[Gre99] F. Green. Exponential sums and circuits with single threshold gate and mod-gates. *Theory Computing Systems*, 32:453–466, 1999.

[Gre02] F. Green. The correlation between parity and quadratic polynomials mod 3. In *Proceedings of the 17th IEEE Conference on Computational Complexity*, pages 65–72, 2002.

[Gro94] V. Grolmusz. A weight-size tradeoff for circuits with mod m gates. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 68–74, 1994.

[GRS05] F. Green, A. Roy, and H. Straubing. Bounds on an exponential sum arising in boolean circuit complexity. *C.R. Acad. Sci. Paris*, Ser. I 340, 2005.

[GT00] V. Grolmusz and G. Tardos. Lower bounds for MOD_p – MOD_m circuits. *SIAM Journal on Computing*, 29(4):1209–1222, 2000.

[GT06] A. Gál and V. Trifonov. On the correlation between parity and modular polynomials. In *Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science*, 2006. To appear.

[Hås86] J. Håstad. *Computational Limitations of Small-Depth Circuits.* MIT Press, 1986.

[HCS79] D. Hirschberg, A. Chandra, and D. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[HG84] J. Håstad and M. Goldmann. On the power of small depth threshold circuits. *Computational Complexity*, 1(2):113–129, 1984.

[HKP84] H. Hoover, M. Klawe, and N. Pippinger. Bounding fan-out on logical networks. *Journal of the ACM*, 31:13–18, 1984.

[HM04] K. Hansen and P. Miltersen. Some meet-in-the-middle circuit lower bounds. In *Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science*, pages 334–345, 2004.

[HMP+87] A. Hajnal, W. Maass, P. Pudlák, M. Szegedy, and G. Turán. Threshold circuits of bounded depth. In *Proceedings of the 8th IEEE Symposium on Foundations of Computer Science*, pages 99–110, 1987.

[HW93] S. Hooray and A. Wigderson. Universal traversal sequences for expander graphs. *Information Processing Letters*, 46(2):67–69, 1993.

[HZ96] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 438–447, 1996.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17:935–938, 1988.

[Ist88] S. Istrail. Polynomial universal traversing sequences for cycles are constructible. In *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pages 491–503, 1988.

[IW97] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the 29th ACM Symposium on the Theory of Computing*, pages 220–229, 1997.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[JM91] D. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} |V|)$ parallel time for the CREW PRAM. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 688–697, 1991.

[JM92] D. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 363–372, 1992.

[KNP92] D. Karger, N. Nisan, and M. Parnas. Fast connected components algorithm for the EREW PRAM. In *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 373–381, 1992.

[Kou01a] M. Koucký. Log-space constructible universal traversal sequences for cycles of length $O(n^{4.03})$. Technical Report ECCC-TR01-13, The Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc, 2001.

[Kou01b] M. Koucký. Universal traversal sequences with backtracking. In *Proceedings of the 16th IEEE Conference on Computational Complexity*, pages 21–27, 2001.

[KP94] M. Krause and P. Pudlák. On the computational power of depth 2 circuits with threshold and modulo gates. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 48–57, 1994.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science–Volume A: Algorithms and Complexity*. MIT Press, 1990.

[LN97] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 1997.

[LP82] H. Lewis and C. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19:161–187, 1982.

[Nis90] N. Nisan. Psuedorandom generators for space-bounded computation. In *Proceedings of the 22nd ACM Symposium on Theory of Computing*, pages 204–212, 1990.

[NSW92] N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 24–29, 1992.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PR02] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895, 2002.

[Rad94] T. Radzik. Computing connected components on EREW PRAM. Technical Report 94/02, King's College London, 1994.

[Rei85] J. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[Rei05] O. Reingold. Undirected st-connectivity in log-space. In *Proceedings of the 37th ACM Sympoisium on Theory of Computing*, pages 376–385, 2005.

[RW93] A. Razborov and A. Wigderson. $n^{\Omega(\log n)}$ lower bounds on the size of depth-3 threshold circuits with AND gates at the bottom. *Information Processing Letters*, 45:303–307, 1993.

[Sav70] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[SJ81] C. Savage and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal on Computing*, 10(4):682–691, 1981.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 77–82, 1987.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[SV84] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13:309–423, 1984.

[SZ95] M. Saks and S. Zhou. RSPACE(S) $\subseteq$ DSPACE($S^{3/2}$). In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, pages 344–353, 1995.

[Sze39] G. Szegő. *Orthogonal Polynomials*. American Mathematical Society, 1939.

[Sze87] E. Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the EATCS*, 33:96–100, 1987.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tri05] V. Trifonov. An $O(\log n \log\log n)$ space algorithm for undirected st-connectivity. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, pages 626–633, 2005.

[TV85] R. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[Vol05] J. F. Voloch. Symmetric functions and Witt vectors. Personal communication, December 2005.

[Yao85] A. Yao. Separating the polynomial hierarchy by oracles. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*, pages 1–10, 1985.

[Yao90] A. Yao. On ACC and threshold circuits. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, pages 619–627, 1990.

[Zab56] S. Zabek. Sur la périodicité modulo m des suites de nombres $\binom{n}{k}$. *Ann. Univ. Mariae Curie Sklodowska*, pages 37–47, 1956.

VITA

Vladimir Traianov Trifonov was born in Pleven, Bulgaria, on August 22, 1975, the son of Traian Tonchev Trifonov and Galina Stoianova Trifonova. After completing his work at the Mathematical High School "Geo Milev", Pleven, Bulgaria, in 1995, he entered Sofia University, Sofia, Bulgaria. He spent Spring 1998 in the University College London and the University of Greenwich, London, United Kingdom, and received the degree of Bachelor of Science from Sofia University in May 1999. In September 1999 he entered the Graduate School of the University of Texas at Austin.

Permanent address: 40 Grenaderska Str., en. B, ap. 17
Pleven, Bulgaria 5800

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.