Extending Branching Programs with Memory

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computer Science and Engineering by Research

by

Nithish Raja
2021201079
nithish.raja@research.iiit.ac.in

Advisor: Dr. Suryajith Chillara

International Institute of Information Technology Hyderabad
500 032, India

June 2024

International Institute of Information Technology Hyderabad
Hyderabad, India

CERTIFICATE

This is to certify that work presented in this thesis titled ***Extending branching programs with memory*** by *Nithish Raja* has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Advisor: Dr. Suryajith Chillara

Abstract

In this study, we continue the line of research that was initiated by Mengel [Men13] wherein he showed that by augmenting algebraic branching programs with a single stack as memory and random-access memory we obtain characterizations for the classes VP and VNP respectively. In this thesis, we introduce k-stack branching programs, a generalization of Mengel's single-stack branching program, and show that when $k \geqslant 2$, the computational model characterizes the class VNP. Due to this, we get a clean characterization of the classes VBP, VP, and VNP using algebraic branching programs with 0, 1, and 2 stacks as memory. Similar to what happens with pushdown automata, we show that a branching program with 2 stacks as memory is equivalent to a branching program with a queue as memory.

Next, we impose restrictions such as bounded stack height and bounded branching program width upon stack branching programs and show that any polynomial family in VP can be computed by a polynomial-sized stack branching program with the stack height bounded by $O(\log(n)^2)$. Finally, we detail a technique using which we can construct a $(k+1)$-stack branching program that computes $\sum_{\bar{e}\in\{0,1\}^{m-n}} q_m(x_1, \cdots, x_n, \bar{e})$ when given a k-stack branching program that computes $q_m(x_1, \cdots, x_n, \bar{e})$. Using this we give an alternate proof for the fact that exponential summation of a polynomial family in VP gives a polynomial family in VNP which was originally shown in [Val82]. This technique combined with one of the previous result of adding more stacks to a 2-stack branching program does not increase the computational power gives us alternate proofs for

- exponential summation of a polynomial family in VBP gives a polynomial family in VNP and
- exponential summation of a polynomial family in VNP gives a polynomial family in VNP.

Contents

Chapter Page

1 Introduction . . . 1
1.1 Background . . . 1
1.2 Motivation . . . 3
1.3 Our Contribution . . . 4

2 Preliminaries . . . 6
2.1 Notation . . . 6
2.2 Definitions . . . 6
2.2.1 Polynomials of interest . . . 6
2.2.2 Computational Models . . . 7
2.2.3 Algebraic Complexity Classes . . . 8
2.2.4 Computational Models with Memory . . . 9

3 Improved characterization of VP . . . 11
3.1 Stack Branching Programs with polylogarithmic stack height . . . 11

4 Characterizing VNP using branching programs with memory . . . 15
4.1 2-Stack Branching Programs . . . 15
4.2 k-Stack Branching Programs . . . 21
4.3 Queue Branching Programs . . . 24

5 A closer look at stack branching programs . . . 28
5.1 Restricting stack height . . . 28
5.2 Restricting branching program width . . . 29

6 Exponential summation . . . 31
6.1 Simulating exponential summation using stacks . . . 31

7 Hardness of computing permanent . . . 35
7.1 Permanent under row operations . . . 35
7.2 Reduction . . . 36

8 Discussion . . . 37

Bibliography . . . 38

Chapter 1

Introduction

1.1 Background

Complexity theory is the study of tractability of tasks where there are constraints on resources. Furthermore, in algebraic complexity theory, the task of interest is computing polynomials and resources are arithmetic operations. In their seminal work [Val79], Valiant showed that the polynomials can be categorized as algebraic complexity classes, VP and VNP (see 2.2.3 for definitions), similar to boolean languages in computational complexity theory. Roughly speaking, VP is the class of polynomial families that can be computed efficiently by an algebraic circuit and VNP is the class of polynomial families that can be defined efficiently by an algebraic circuit. Furthermore, Valiant also showed the existence of polynomial families that are complete for the classes under the notion of p-projections (p-projections can be thought of as the algebraic equivalent of polynomial time reductions).

Thereafter, two new algebraic complexity classes, VF and VBP (see 2.2.3 for definitions), were formalised. While VP was the class of polynomial families that can be efficiently computed by an algebraic circuit, VF and VBP are the class of polynomial families that can be computed efficiently by an algebraic formula and algebraic branching program respectively. The relation between the four complexity classes are shown in equation 1.1. One of the main goals of algebraic complexity is to make the relationship between the complexity classes explicit i.e., either show that two classes are equivalent or show that one is a strict subset of the other.

$$\mathsf{VF} \subseteq \mathsf{VBP} \subseteq \mathsf{VP} \subseteq \mathsf{VNP}. \tag{1.1}$$

One approach to separate two algebraic complexity classes is to show super-polynomial lower bounds for a polynomial family in the larger class. Proving unconditional lower bounds has always been a difficult task in complexity theory and it is no different in algebraic complexity theory. To obtain insights on separating the algebraic complexity classes, many studies looked into restricted models of computation. Some of the restrictions considered were *monotone* computations, *non-commutative* computations and *multilinear* computations. Some of the well known lower bounds and separations in these restricted settings are mentioned below.

In the monotone computation setting, negation is not allowed i.e., operations such as multiplying by a negative number or subtracting two numbers is not allowed. Due to this restriction, any monomial computed at any intermediate step will show up in the final output. Valiant [Val80] showed the existence of polynomial familiess that can be computed efficiently in the general setting i.e., using negations, but are hard to compute without negations. This result essentially shows a separation between the classes VP and monotone VP (mVP $\subsetneq$ VP). This line of work was extended in a study by Hrubes and Yehudayoff [HY09], who proved the same separation using a polynomial family having a constant degree.

Hrubes and Yehudayoff [HY16] showed a separation between mVBP and mVP was shown. Yehudayoff [Yeh19] then gave a super-polynomial lower bound for a polynomial family in mVNP therby separating mVP and mVNP. Srinivasan [Sri20], then gave an exponential lower bound for a different polynomial family in mVNP. Inspired by techniques used in the exponential lower bound, Chattopadhyay, Datta, Utsab and Mukhopadhyay [CDGM22] showed a strongly exponential separation between the classes mVP and VP.

The current class relations in the monotone setting are shown in equation 1.2.

$$\mathsf{mVF} \subseteq \mathsf{mVBP} \subsetneq \mathsf{mVP} \subsetneq \mathsf{mVNP}. \tag{1.2}$$

We refer to a circuit as a *skew circuit* if one of the child of every multiplication gate is a leaf node. It is well known that in the general setting, skew circuits are equivalent to algebraic branching programs with respect to p-projections. However, things change quite a bit in the non-commutative setting. In the non-commutative setting, we make a distinction between left multiplication and right multiplication i.e., in the non-commutative setting $x_1x_2 - x_2x_1 \neq 0$. In this setting, Nisan [Nis91] showed, contrary to the general setting, a strong exponential separation between algebraic branching programs and skew circuits. The technique used by Nisan was subsequently generalised by Limaye, Malod and Srinivasan [LMS16] and used to show an exponential separation between non-commutative *skew circuits* and non-commutative circuits.

Similar to the results in the general setting [Val80] [HY09] [CDGM22] showing the power of negation, it was shown by Hrubes and Yehudayoff [HY13] that negation is exponentially powerful in the non-commutative setting too. To state it explicitly, there are polynomial families that can be efficiently computed by non-commutative circuits but cannot be efficiently computed by circuits that are both monotone and non-commutative.

The current class relations in the non-commutative setting are shown in the equation below.

$$\mathsf{VF}_{\mathsf{nc}} \subseteq \mathsf{VBP}_{\mathsf{nc}} \subsetneq \mathsf{VP}_{\mathsf{nc,skew}} \subsetneq \mathsf{VP}_{\mathsf{nc}} \subseteq \mathsf{VNP}_{\mathsf{nc}}. \tag{1.3}$$

In the multilinear setting, the degree of any variable appearing in a monomial is bounded to be at most 1. One possible motivation for such a restriction is the following. Consider a multilinear polynomial family. It could very well be the case that the polynomial family can be efficiently computed

by a general circuit but the intermediate computations in the circuit involve non-multilinear terms which get cancelled out. Therefore, by restricting ourselves to multilinear circuits, we give ourselves another way to test the power of negation albeit in a weaker way than in the monotone setting. Upon imposing such a restriction, there have been many studies showing the existence of hard polynomial families. One such work was by Dvir, Malod, Perifel and Yehudayoff [DMPY12], which showed a separation between multilinear formulas and multilinear branching programs was shown i.e., ($\mathsf{VF}_{\mathsf{ml}} \subsetneq \mathsf{VBP}_{\mathsf{ml}}$). Later, Mahajan, Saurabh and Tavenas [MST16] showed that the classes $\mathfrak{ml}\mathsf{VP}$ and $\mathfrak{ml}\mathsf{VNP}$ are equivalent. This is shocking since such a collapse is not believed to occur in the general setting.

The current class relations in the multilinear setting are shown in equation 1.4.

$$\mathsf{VF}_{\mathsf{ml}} \subsetneq \mathsf{VBP}_{\mathsf{ml}} \subseteq \mathsf{VP}_{\mathsf{ml}} = \mathsf{VNP}_{\mathsf{ml}}. \tag{1.4}$$

1.2 Motivation

The classes VF and VP are defined based on formulas and circuits respectively. The only difference in definition between formulas and circuits is that the fan-out of nodes is restricted to 1 in formulas and there is no such restriction for circuits. The class VBP is defined on algebraic branching programs. Algebraic branching programs are a very different computational model when compared to formulas and circuits. Furthermore, the class VNP does not have an efficient characterization. To compare and possibly separate these classes, it would be ideal if all of them are characterized by similar computational models which makes them easier to compare. Towards this, it was shown that skew-circuits are equivalent to algebraic branching programs and the polynomial upper bound on degree of polynomial families was handled by showing that the class VP is characterized by multiplicatively disjoint circuits (Circuits where no leaf node appears in both the left subtree and the right subtree of any product gate).

VF	$\subseteq$	VBP	$\subseteq$	VP	$\subseteq$	VNP
Formula		Skew circuit		M.D. circuit		

Natural characterization of a class is often followed by discovery of complete polynomials for that class based on the characterization. It is important to come up with complete polynomials since they are the hardest polynomials in that class and therefore ideal candidates for lower bound and separation results.

One possible approach is to look at variations of branching programs and provide characterizations for each complexity class. One argument for using branching programs is their graph-like structure allows us to come up with hard polynomial families for the corresponding classes easily. An example supporting this argument is the proof by Valiant [Val79] showing the the determinant polynomial family is a hard polynomial family for the class VBP. Ben-Or and Cleave [BOC92] showed that formulas are characterized by width-3 branching programs. While it is possible to characterize the class VP using

quasi-polynomial sized branching programs, the degree needs to be artificially bounded to polynomial in number of variables. Due to this reason, such a characterization is not a natural one.

Malod [Mal11] studied the class VPSPACE (informally, we are allowed to perform operations that duplicate computation and immediately follow it up with partial substitution of one variable) and showed that it is characterized by succinct algebraic branching programs. A succinct branching program is any branching program that has a polynomial sized circuit that outputs the corresponding edge weight when two vertices of the branching program are given as input. Malod further showed that the class VNP is characterized by succinct algebraic branching programs with their length restricted to a polynomial. Unfortunately, a succinct branching program based characterization was not obtained for the class VP.

VF	$\subseteq$	VBP	$\subseteq$	VP	$\subseteq$	VNP	$\subseteq$	VPSPACE
width-3 ABP		ABP				poly-length succABP		succABP

A different variation of algebraic branching programs was considered by Mengel. Mengel [Men13] showed that algebraic branching programs augmented with memory can be used to characterize the classes. To be specific, branching program with a single stack as memory characterizes the class VP and branching program with random-access memory characterized the class VNP. Following up Mengel's results, Chaugule, Limaye and Pandey [CLP21] defined new polynomial families StackDet, CountDet and showed that they are complete for VP and VNP respectively.

VF	$\subseteq$	VBP	$\subseteq$	VP	$\subseteq$	VNP
width-3 ABP		ABP		SBP		RABP

1.3 Our Contribution

The main goal of this thesis is to further refine the computational models introduced by Mengel. We also introduce new restrictions on the computational model and study the computing power of the model under such restrictions.

Class	**Characterization**	**Theorem**
VP	polynomial-sized stack branching programs over binary stack-alphabet and *polylogarithmic stack-height*	Theorem 17
VNP	polynomial-sized 2-stack branching programs over binary stack-alphabet	Theorem 21
VNP	polynomial-sized queue branching programs over binary stack-alphabet	Theorem 26

Using these characterizations, we give alternate proofs for the following

- Any polynomial family in the class VP can be computed by a quasipolynomial-sized ABP (Section 3.1).
- Any polynomial family, defined as exponential sum of a polynomial family in VBP, is in VNP (Corollary 33).
- Any polynomial family, defined as exponential sum of a polynomial family in VP, is in VNP (Section 6.1).
- Any polynomial family, defined as exponential sum of polynomial family in VNP, is in VNP i.e., the class VNP is closed under exponential summation (Corollary 32).

Chapter 2

Preliminaries

2.1 Notation

- We use the notation $[n]$ to refer to the set $\{1, 2, \ldots, n\}$
- We use bold letters $\mathbf{X}$, $\mathbf{Y}$ to denote sets of variables
- The abbreviation ABP is used to denote an algebraic branching program
- The abbreviations SBP and QBP are used to denote stack branching programs and queue branching programs respectively
- The abbreviation $\mathsf{S}^{[k]}\mathsf{BP}$ is used to denote a k-stack branching program

2.2 Definitions

2.2.1 Polynomials of interest

Definition 1. *Permanent polynomial*

The permanent polynomial Perm_n *is defined over a square matrix* $\mathbf{X}$ *of size* n*. Let* $\{x_{ij} \mid \forall i, j \in [n]\}$ *denote the* n^2 *entries of the matrix and* S_n *denote the set of all permutations over* $[n]$*. Then* Perm_n *is defined as follows*

$$\mathsf{Perm}_n(\mathbf{X}) = \sum_{\sigma \in S_n} \prod_{i \in [n]} x_{i\sigma(i)} \tag{2.1}$$

Definition 2. *Hamiltonian cycle polynomial*

The hamiltonian cycle polynomial HC_n *is defined over a square matrix* $\mathbf{X}$ *of size* n*. Let* $\{x_{ij} \mid \forall i, j \in [n]\}$ *denote the* n^2 *entries of the matrix and* H_n *denote the set of all permutations over* $[n]$ *that have exactly one cycle. Then* HC_n *is defined as follows*

$$\mathsf{HC}_n(\mathbf{X}) = \sum_{\sigma \in H_n} \prod_{i \in [n]} x_{i\sigma(i)} \tag{2.2}$$

2.2.2 Computational Models

Definition 3 ([SY09]). *Formulas*

A formula is a DAG with the terminal nodes labelled by variables or constants and the intermediate nodes labelled by addition or multiplication gates. Each addition and multiplication gate has fan-in bounded by 2 *and fan-out bounded by* 1*. An important consequence of bounding the fan-out to* 1 *is that intermediate computations cannot be reused. The size of a formula is given by the number of nodes.*

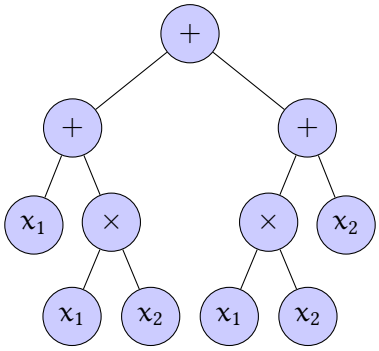

Figure 2.1 Formula computing $x_1 + 2x_1x_2 + x_2$

Definition 4 ([SY09]). *Circuits*

A circuit is a DAG with the terminal nodes labelled by variables or constants and the intermediate nodes labelled by addition or multiplication gates. Each addition and multiplication gate has fan-in bounded by 2*. Unlike formulas, the fan-out is not bounded in circuits. Therefore, the intermediate computations can be reused as many times as required. The size of a circuit is given by the number of nodes in it.*

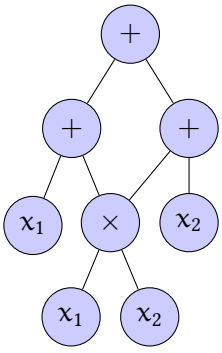

Figure 2.2 Circuit computing $x_1 + 2x_1x_2 + x_2$

Definition 5 ([SY09]). *Algebraic Branching Programs (ABP)*

An ABP *is a single source, single sink DAG where each edge has a weight (variables/constants) and addition and multiplication operations are represented by edge connections (parallel/series). Any* ABP *can*

be described by the tuple (G, s, t) *where* G *is the DAG,* s *is the source and* t *is the sink. The computed polynomial is given by* $\sum \mathrm{wt}(s \text{ to } t \text{ path})$*. The size of an* ABP *is given by the number of vertices in it.*

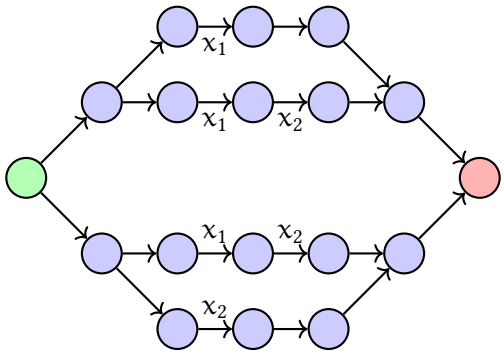

Figure 2.3 ABP computing $x_1 + 2x_1x_2 + x_2$

2.2.3 Algebraic Complexity Classes

Definition 6 ([SY09]). *VF*

VF *is defined as the class containing all polynomial families that can be computed by formulas of size polynomial in number of input variables.*

Definition 7 ([SY09]). *VBP*

VBP *is defined as the class containing all polynomial families that can be computed by algebraic branching programs of size polynomial in number of input variables.*

Definition 8 ([SY09]). *VP*

VP *is defined as the class containing all polynomial families, with degree bounded by a polynomial in number of input variables, that can be computed by circuits of size polynomial in number of input variables.*

Definition 9 ([SY09]). *p-projections*

Let the polynomial families $p_n(x_1, \ldots, x_n)$ *and* $g_m(y_1, \ldots, y_m)$ *belong to the ring of polynomial over the field* $\mathbb{F}$*. We say that* $p_n(x_1, \ldots, x_n)$ *is a p-projection of* $g_m(y_1, \ldots, y_m)$ *if there exists a map* $\sigma : \{y_i\}_{i=1}^m \to \{x_i\}_{i=1}^n \cup \mathbb{F}$ *s.t.*

$$p_n(x_1, \ldots, x_n) = g_m(\sigma(y_1), \ldots, \sigma(y_m)). \tag{2.3}$$

Definition 10 ([SY09]). *VNP*

A polynomial family $p_n(x_1, \ldots, x_n)$ *is in* VNP *if one of the following conditions are satisfied*

- *There exists a polynomial sized function* q *s.t.*

$$p_n(x_1, \ldots, x_n) = \sum_{e \in \{0,1\}^n} q(e) \prod_{i \in [n]; e_i = 1} x_i, \tag{2.4}$$

- *There exists a polynomial family* g_m *in* VNP *s.t.* p_n *is a p-projection of* g_m.

Another equivalent way to define the class VNP is as follows. The polynomial family p_n is in VNP if and only if there exists a polynomial family $g_n \in \mathsf{VP}$ and $m \in \mathrm{poly}(n)$ s.t.

$$p_n(x_1, \ldots, x_n) = \sum_{e \in \{0,1\}^m} g_{n+m}(x_1, \ldots, x_n, e_1, \ldots, e_m). \tag{2.5}$$

It is easy to see that $\mathsf{VP} \subseteq \mathsf{VNP}$ since setting $m = 0$ causes polynomial p_n to be inside VP.

2.2.4 Computational Models with Memory

Definition 11 ([Men13]). *Stack Branching Program*

A SBP *is an* ABP *where the edges are additionally labelled with operations from the set* $\{\mathrm{push}, \mathrm{pop}, \mathrm{nop}\}$. SBP *can be described by the tuple* (G, s, t, T) *where* G *is the DAG,* s *source vertex,* t *is the sink vertex and* T *is the stack alphabet set. The polynomial computed by an* SBP *is given by sum of weights of realizable* s *to* t *paths. The size of a* SBP *is given by the number of vertices in the underlying graph.*

A s *to* t *path is said to be realizable if and only if it satisfies the following conditions*

- *Starting with an empty stack, we perform all the stack operations along the path and end with an empty stack*
- *There are no faults while performing the stack operations along the path i.e., we do not try to pop an element that is not at the top of the stack*

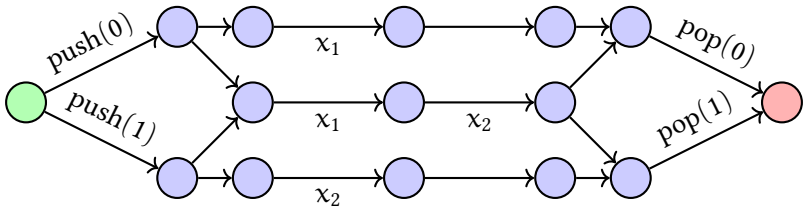

Figure 2.4 SBP computing $x_1 + 2x_1x_2 + x_2$

Definition 12. *k-Stack Branching Program*

A $\mathsf{S}^{[k]}\mathsf{BP}$ *is a generalization of* SBP *where the additional labels are now from the set* $\{\mathrm{push}_i, \mathrm{pop}_i\}_{i=1}^{k} \cup \{\mathrm{nop}\}$. *The polynomial computed by* $\mathsf{S}^{[k]}\mathsf{BP}$ *is given by sum of weights of realizable* s *to* t *paths and the size of a* $\mathsf{S}^{[k]}\mathsf{BP}$ *is given by the number of vertices in the underlying graph.*

A s *to* t *path is said to be realizable if and only if it satisfies the following conditions*

- *Starting with* k *empty stacks, we perform all the stack operations along the path and end with* k *empty stacks*

- *There are no faults while performing the stack operations along the path i.e., we do not try to pop an element that is not at the top of the corresponding stack*

Definition 13. *Queue Branching Program*

A QBP *is an* ABP *where the edges have operations from the set* $\{\text{insert}, \text{remove}, \text{nop}\}$ *as additional labels.* QBP *can be described by the tuple* (G, s, t, T) *where* G *is the DAG,* s *is the source vertex,* t *is the sink vertex and* T *is the queue alphabet set. The polynomial computed by an* QBP *is given by sum of weights of realizable* s *to* t *paths. The size of a* QBP *is given by the number of vertices in the underlying graph.*

A s *to* t *path is said to be realizable if and only if it satisfies the following conditions*

- *Starting with an empty queue, we perform all the queue operations along the path and end with an queue stack*
- *There are no faults while performing the queue operations along the path i.e., we do not try to remove an element that is not at the front of the queue*

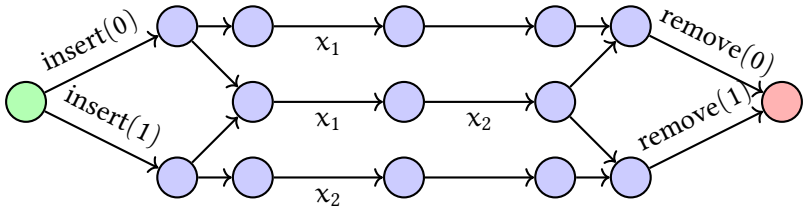

Figure 2.5 QBP computing $x_1 + 2x_1x_2 + x_2$

Lemma 14 (Valiant's criterion [Val79])**.** *A polynomial* $p_n(x_1, \ldots, x_n)$ *is said to belong to the class* VNP *if computing its coefficients is in* #P/poly.

For a polynomial with coefficients being only 0 or 1, the above lemma can be simplified to the following. A polynomial $p_n(x_1, \ldots, x_n)$ with the coefficients being only 0 or 1 is said to belong to the class VNP if computing its coefficients is in P/poly.

Chapter 3

Improved characterization of VP

3.1 Stack Branching Programs with polylogarithmic stack height

One of the well known results when it comes to characterizing the class VP is the result by Valiant, Skyum, Berkowitz and Rackoff [VSBR83]. Informally, they showed that any polynomial family in the class VP can be computed by an algebraic circuit of depth $O(\log(n)^2)$. In this section, we show a similar result for stack branching programs. We show that any polynomial family in the class VP can be computed by a polynomial-sized SBP with the stack height bounded by $O(\log(n)^2)$.

Before showing the result, we first borrow some notation from [VSBR83]. For a gate w in a given circuit C, we use $f(w)$ to denote the polynomial computed at w, and $d(w)$ to denote degree of polynomial computed at w. Further, we have the following.

Definition 15. *Let C be a circuit and v, w are gates in the circuit. $f(v)$ and $f(w)$ denote the polynomial computed at gates v and w respectively. Then, the function $f(v;w)$ is defined as follows.*

- $f(v;w) = 1$, *if* $v = w$
- $f(v;w) = 0$, *if* $v \neq w$ *and* $f(w) \in \mathbb{F} \cup \{x_i\}_{i=1}^{n}$ *i.e., w is a leaf node*
- $f(v;w) = f(v;w') + f(v;w'')$, *if* $w = w' + w''$
- $f(v;w) = f(w'')f(v;w')$, *if* $w = w' \times w''$ *s.t.* $d(w'') \leqslant d(w')$

Definition 16. *Let C be a circuit computing an arbitrary polynomial. For any value of a, the set V_a is defined as follows.*

$$V_a = \{t |\ t \text{ is a gate in } C,\ d(t) > a,\ t = t' \times t'',\ d(t') < a\}. \tag{3.1}$$

Theorem 17. *Let P_n be a degree d polynomial computed by a circuit C of size $|C|$ and depth $O(\log(|C|)\log(d))$. Then, there exists a SBP of polynomial size with stack height $O(\log(|C|)\log(d))$ computing P_n.*

Proof. The idea is to show the following.

1. Every gate w in C has a corresponding pair of vertices $\{s_w, t_w\}$ in a relaxed SBP s.t. $\sum \text{wt}(\text{realizable } s_w \text{ to } t_w \text{ walks of length } m_w) = f(w)$, $m_w = O(d(w)^2)$ and stack height $O(\log(d(w)))$, and

2. Every pair of gates v, w s.t. $f(v;w)$ is non-zero, there exists a pair of vertices $\{s_{vw}, t_{vw}\}$ in the relaxed SBP s.t. $\sum \text{wt}(\text{length } m_{vw} \text{ realizable } s_{vw} \text{ to } t_{vw} \text{ walks}) = f(v;w)$, $m_{vw} = O(d(f(v;w)^2))$ and stack height $O(\log(d(f(v;w))))$.

The above statements are proven by induction on the degree of the polynomial computed at each gate. At any stage i, we consider all gates w and pairs of gates v, w s.t. $d(w), d(f(v;w)) \in (2^{i-1}, 2^i]$.

Base case: Consider gates w and pairs of gates v, w s.t. $d(w)$ and $d(f(v;w))$ is 1. Clearly, $f(w)$ and $f(v;w)$ compute variables or constants or linear forms. Edges with weights being variables and constants can be added to the relaxed SBP. Any linear form $\sum_{j=1}^{n} c_j x_j$ can be computed as shown in Figure 3.1. Note that the stack symbols used in Figure 3.1 will change to $\langle w, v, i \rangle$ if the linear form is computed by $f(v;w)$.

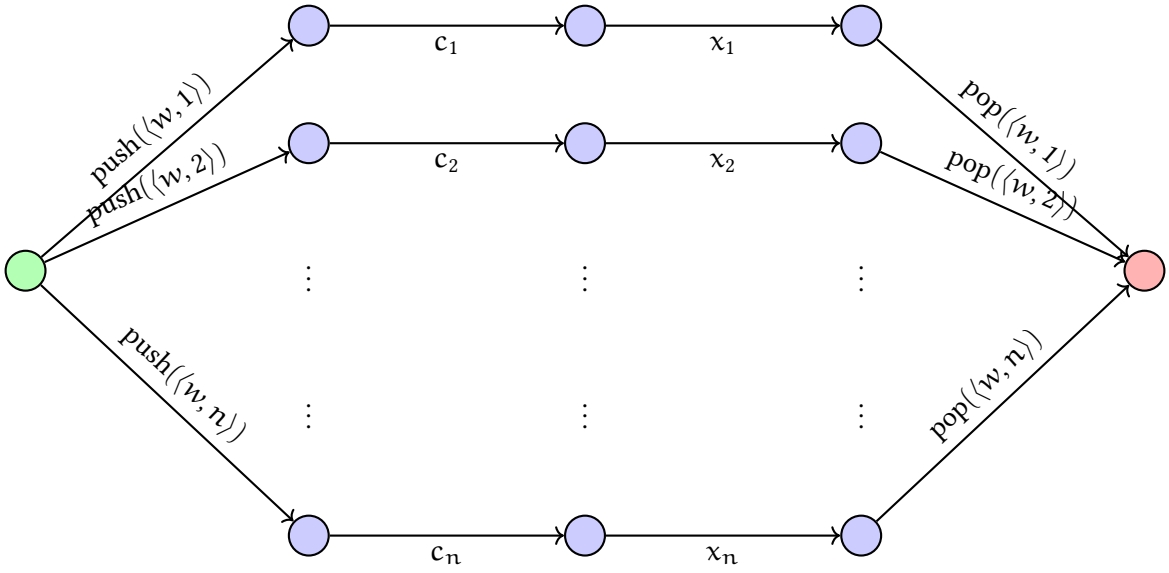

Figure 3.1 Gadget computing linear form

Clearly, the stack height is 1 and path length $\leqslant 1 + 3^{(0+1)} \leqslant 4$.

Inductive assumption:

1. Consider all gates w s.t. $d(w) \in (2^{i-1}, 2^i]$. Assume that there exists pairs of vertices $\{s_w, t_w\}$ in the polynomial-sized relaxed SBP s.t. $\sum \text{wt}(\text{length } m_w \text{ realizable } s_w \text{ to } t_w \text{ walks}) = f(w)$ satisfying $m_w \leqslant d(w) + 3^{(i+1)}$ and stack height is $O(i)$.

2. Consider all pairs of gates v, w s.t. $f(v;w)$ is non-zero and $d(f(v;w)) \in (2^{i-1}, 2^i]$. Assume that there exists pairs of vertices $\{s_{vw}, t_{vw}\}$ in the polynomial-sized relaxed SBP s.t. $\sum$ wt(length m_{vw} realizable s_{vw} to t_{vw} walks) $=$ $f(v;w)$ satisfying $m_{vw} \leqslant d(f(v;w)) + 3^{(i+1)}$ and stack height is $O(i)$.

Increment step:

1. Consider all gates w s.t. $d(w) \in (2^i, 2^{i+1}]$. Due to [VSBR83], we write $f(w) = \sum_{t \in V_a} f(t')f(t'')f(t;w)$ s.t. $d(t')$, $d(t'')$, $d(f(t;w)) \in (2^{i-1}, 2^i]$ and $a = 2^i$. Due to the inductive assumption, we get that a relaxed SBP exists with pairs of vertices computing each $f(t')$, $f(t'')$ and $f(t;w)$.

   Next, by following the steps detailed in [Men13, Proposition 3.6], we construct a relaxed SBP with source and sink vertices s_w, t_w computing $f(w)$. For each product term, we add 3 vertices and for addition, we add 2 vertices and at most m_w vertices to make the walk lengths equal. Trivially, $V_a \leqslant |C|$ and number of gates w is also bounded by $|C|$. Therefore, the total number of vertices added can be at most $(3|C| + m_w + 2)|C|$. This also causes the stack height to increase by 2.

   Due to the inductive assumption, $m_{t'} \leqslant d(t') + 3^{(i+1)}$, $m_{t''} \leqslant d(t'') + 3^{(i+1)}$ and $m_{tw} \leqslant d(f(t;w)) + 3^{(i+1)}$.

$$\begin{aligned} m_w &\leqslant \max_t\{m_{t'} + m_{t''} + m_{tw}\} && (3.2)\\ &\leqslant \max_t\{d(t') + d(t'') + d(f(t;w)) + 3^{(i+2)}\} && (3.3)\\ &\leqslant \max_t\{d(t) + d(w) - d(t) + 3^{(i+2)}\} && (3.4)\\ &\leqslant \max_t\{d(w) + 3^{(i+2)}\} && (3.5)\\ &\leqslant d(w) + 3^{(i+2)}. && (3.6) \end{aligned}$$

2. Consider all pairs of gates v, w s.t. $d(f(v;w)) \in (2^i, 2^{i+1}]$. Due to [VSBR83], we write $f(v;w) = \sum_{t \in V_a} f(t'')f(v;t')f(t;w)$ s.t. $d(t'')$, $d(f(v;t'))$, $d(f(t;w)) \leqslant (2^{i-1}, 2^i]$ and $a = 2^i + d(v)$. Due to the inductive assumption, we get that a relaxed SBP exists with pairs of vertices computing $f(t'')$, $f(v;t')$ and $f(t;w)$.

   Once again, by following the steps in [Men13, Proposition 3.6], we construct a relaxed SBP with source and sink vertices s_{vw}, t_{vw} computing $f(v;w)$. Since, we look at pairs of gates, the total number of vertices added can be atmost $(3|C| + m_{vw} + 2)|C|^2$. This also causes the stack height to increase by 2.

   This causes the stack height to increase by 2.

Due to the inductive assumption, $m_{t''} \leqslant d(t'') + 3^{(i+1)}$, $m_{vt'} \leqslant d(f(v;t')) + 3^{(i+1)}$ and $m_{tw} \leqslant d(f(t;w)) + 3^{(i+1)}$.

$$\begin{aligned} m_{vw} &\leqslant \max_t\{m_{t''} + m_{vt'} + m_{tw}\} && (3.7)\\ &\leqslant \max_t\{d(t'') + d(t') - d(v) + d(w) - d(t) + 3^{(i+2)}\} && (3.8)\\ &\leqslant \max_t\{d(w) - d(v) + 3^{(i+2)}\} && (3.9)\\ &\leqslant d(f(v;w)) + 3^{(i+2)}. && (3.10) \end{aligned}$$

The maximum walk length becomes $d + 3^{\lceil \log(d) \rceil + 1} \in O(d^2)$. Now, we can convert the relaxed SBP to a normal SBP as shown in [Men13, lemma 3.5] and the size of the SBP remain polynomial.

The overall stack height is atmost $2\log(d)$. However, every alphabet in the stack is of the form $\langle u_k, v_k, u_{k'}, v_{k'} \rangle$ where u_k, v_k, $u_{k'}$ and $v_{k'}$ are gates in C. Converting to binary, the stack height becomes $O(\log(|C|)\log(d))$.

□

Using this characterization, we can now convert the stack branching program with polylogarithmic stack height to an ABP as shown in 27. This immediately gives us that any polynomial family in VP can be computed by a quasipolynomial-sized ABP.

Chapter 4

Characterizing VNP using branching programs with memory

4.1 2-Stack Branching Programs

In this section, we show that the class VNP is exactly characterized by $\mathsf{S}^{[2]}\mathsf{BP}$ i.e., any polynomial computed by a polynomial-sized $\mathsf{S}^{[2]}\mathsf{BP}$ is inside the class VNP and for every polynomial in class VNP, there exists a polynomial sized $\mathsf{S}^{[2]}\mathsf{BP}$ computing it (Theorem 21).

Lemma 18. *Any polynomial family computed by a polynomial sized* $\mathsf{S}^{[2]}\mathsf{BP}$ *is inside* VNP.

Proof. The weight of a source-to-sink path is given by product of weights of edges in that path. Therefore, each source to sink path computes a monomial (multiple paths may compute the same monomial). The coefficient of a monomial is non-zero only if there exists at least one source-to-sink realizable path that computes the monomial. It is easy to see that verifying if a given path is realizable or not can be done in polynomial time. Therefore, due to Lemma 14, the polynomial is in VNP. □

Lemma 19. *For all* $n \in \mathbb{N}$*, there exists a* $O(n^3)$*-sized* $\mathsf{S}^{[2]}\mathsf{BP}$ *that computes the permanent polynomial* Perm_n.

Proof. We shall prove the statement of the theorem by constructing a 2-stack branching program for the permanent polynomial. Let S_1 and S_2 be two stacks. Let T be equal to $\{\langle b, i\rangle \mid b \in \{0,1\}, i \in [n]\}$. For each stack S_u (for $u \in \{1,2\}$), let the push and pop operations be denoted by $\mathrm{push}_u(s)$ and $\mathrm{pop}_u(s)$ for some letter $s \in T$. As before, nop denotes no operation on the stacks. In the figures below, for the sake brevity, edge weights are only mentioned when they are different from 1, and stack operations on edges are mentioned only when they are different from nop. The default edge weight is 1, and the default operation is nop.

Let G' and G'' be directed path graphs as shown in Figure 4.1 and Figure 4.2 respectively. Every edge has a weight 1 in both of these graphs.

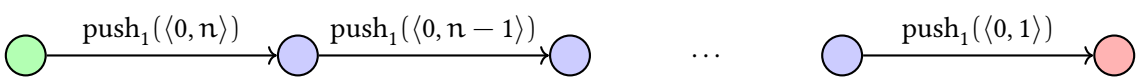

Figure 4.1 Gadget G'

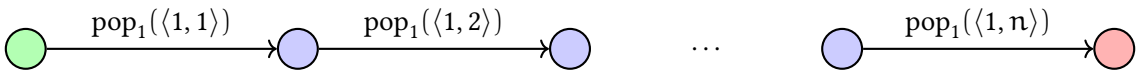

Figure 4.2 Gadget G''

For each $j \in [n]$, let the source-to-sink graph $G_{(j)}$ be as shown in Figure 4.3. Every edge has weight 1 in this graph.

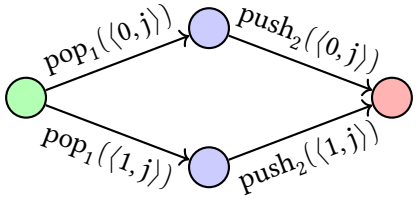

Figure 4.3 Gadget $G_{(j)}$

For each $i \in [n]$, let the source-to-sink graph $G_{pop(i)}$ be constructed by placing the graphs $G_{(1)}$, $G_{(2)}$, ..., $G_{(i-1)}$ in series (see Figure 4.4). That is, for all $1 \leqslant j \leqslant i-2$, the sink of $G_{(j)}$ is connected to the source of $G_{(j+1)}$ (with edges of weight 1), and the source and sink of $G_{pop(i)}$ are the source of $G_{(1)}$, and the sink of $G_{(i-1)}$.

Figure 4.4 Gadget $G_{pop(i)}$

For each $j \in [n]$, let the source-to-sink graph $G'_{(j)}$ be as shown in Figure 4.5. Every edge has a weight 1 in this graph.

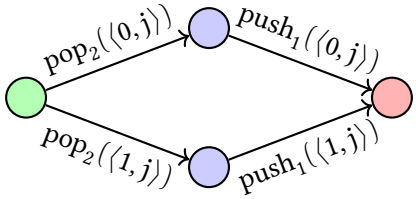

Figure 4.5 Gadget $G'_{(j)}$

For each $i \in [n]$, let the source-to-sink graph $G_{push(i)}$ be constructed by placing the graphs $G'_{(i-1)}, G'_{(i-2)}, \ldots, G'_{(1)}$ in series (see Figure 4.6). That is, for all $2 \leqslant j \leqslant i-1$, the sink of $G'_{(j)}$ is connected to the source of $G'_{(j-1)}$ (with edges of weight 1), and the source and sink of $G_{push(i)}$ are the source of $G'_{(i-1)}$, and the sink of $G'_{(1)}$.

Figure 4.6 Gadget $G_{\mathrm{push}(i)}$

For each $i \in [n]$, let the source-to-sink graph $G_{\mathrm{map}(i)}$ be constructed as shown in Figure 4.7. That is, the source and sink vertices of $G_{\mathrm{map}(i)}$ are connected by n disjoint paths such that j^{th} such path (for all $1 \leqslant j \leqslant n$) is obtained by connecting source, graph gadget $G_{\mathrm{pop}(j)}$, a single vertex (say $v_{\mathrm{map}(i),j}$) and graph gadget $G_{\mathrm{push}(j)}$ in series such that all new edges except for the edge between $G_{\mathrm{pop}(j)}$ and $v_{\mathrm{map}(i),j}$, and the edge between $v_{\mathrm{map}(i),j}$ and $G_{\mathrm{push}(j)}$ have weight 1 and no associated stack operations. The edge between $G_{\mathrm{pop}(j)}$ and $v_{\mathrm{map}(i),j}$ has weight $x_{i,j}$ and the associated stack operation $\mathrm{pop}_1(\langle 0, j\rangle)$, and the edge between $v_{\mathrm{map}(i),j}$ and $G_{\mathrm{push}(j)}$ has weight 1 and the associated stack operation $\mathrm{push}_1(\langle 1, j\rangle)$.

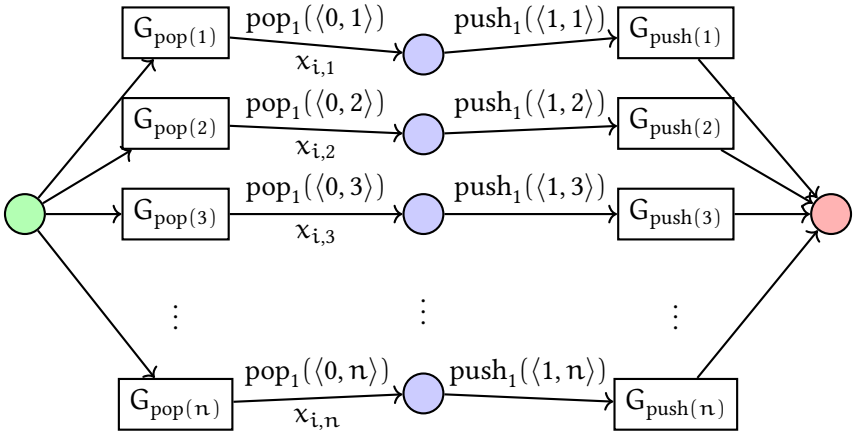

Figure 4.7 Gadget $G_{\mathrm{map}(i)}$

We now get the 2-stack branching program G for the permanent polynomial Perm_n (that remains to be proved) by placing the graph gadgets G', $G_{\mathrm{map}(1)}$, …, $G_{\mathrm{map}(n)}$, G'' in series (with connecting edges of weight 1 and no associated stack operations) as shown in Figure 4.8.

Figure 4.8 Gadget connections to obtain a 2-stack branching program G

Given the construction of the 2-stack branching program G given above, we now need to show that the polynomial computed by G is, in fact, the permanent polynomial. Towards that we shall show that there is a bijection between the 2-stack realizable paths in G and monomials corresponding to permutations, and thus the sum of weights of 2-stack realizable paths in G is the permanent polynomial Perm_n.

Every 2-stack realizable path P maps to a unique permutation π_P: For every $i \in [n]$, the weight of j^{th} source-to-sink path in the graph gadget $G_{map(i)}$ is $x_{i,j}$, that is, this path maps i to j (let us call this path $P_{i,j}$). Thus from the construction described above, source-to-sink paths in G have weights of the following form.

$$\prod_{i=1}^{n} x_{i,j_i} \text{ where } j_i \in [n] \text{ for all } i \in [n].$$

Note that not every source-to-sink path in G is 2-stack realizable. We will now argue that for any 2-stack realizable path P, it cannot happen that there exist $i < i' \in [n]$, such that weights of path P restricted to $G_{map(i)}$ and $G_{map(i')}$ be equal to $x_{i,k}$ and $x_{i',k}$ respectively. For the sake of contradiction, let us suppose that such an event occurs.

Let us first focus our attention on sub-paths P_i and $P_{i'}$ obtained by restricting path P to $G_{map(i)}$ and $G_{map(i')}$ respectively.

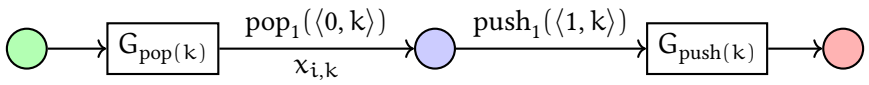

Figure 4.9 Path P_i where i maps to k in $G_{map(i)}$

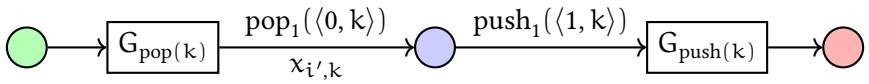

Figure 4.10 Path $P_{i'}$ where i' maps to k in $G_{map(i')}$

Note that the operations $pop_1(\langle 0, k\rangle), push_1(\langle 1, k\rangle)$ in $G_{map(i)}$ precede the operations $pop_1(\langle 0, k\rangle), push_1(\langle 1, k\rangle)$ in $G_{map(i')}$, in the sequence of stack operations associated with path P. As i is getting mapped to k in $G_{map(i)}$, $\langle 0, k\rangle$ is popped from the first stack and $\langle 1, k\rangle$ is pushed into it (along with other operations, cf. Figure 4.3 and Figure 4.4). Hereafter, the symbol $\langle 0, k\rangle$ no longer exists inside the first stack. Therefore for the operation $\langle 0, k\rangle$ in $P_{i'}$, the stack throws a fault and thus such a sequence of stack operations is not 2-stack realizable. Hence, through $G_{map(1)}$ till $G_{map(n)}$ each of the n elements are mapped to distinct n elements in $[n]$ and this is a permutation.

Assume for the sake of contradiction that two different 2-stack realizable paths map to the same permutation. Let $G_{map(i)}$ be the graph gadget where the paths diverge. Let one path take the j^{th} edge and the other path take the j'^{th} edge as shown in figure 4.11.

One path clearly gives the monomial with the variable $x_{i,j}$ in it, and the other gives the monomial with the variable $x_{i,j'}$. Therefore, they do not output the same permutations. By compiling all the discussion above, we get that each 2-stack realizable path gets mapped to a unique permutation.

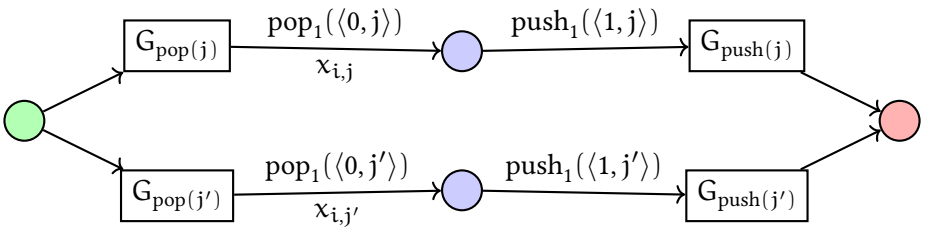

Figure 4.11 Alternate paths

For every permutation, there exists a unique 2-stack realizable path: Let $\pi : [n] \mapsto [n]$ be an arbitrary permutation. That is, for all $i \in [n]$, $\pi(i)$ denotes the value i gets mapped to. First, we will identify a source-to-sink path that generates the monomial corresponding to π. Towards that, in each gadget $G_{map(i)}$ (for all $i \in [n]$), choose the $\pi(i)^{th}$ source-to-sink path within the gadget (let us call this $P_{i,\pi}$, which itself is a path graph). Now the path G', $P_{1,\pi}$, $P_{2,\pi}$, ..., $P_{n,\pi}$, G'' gives us a source-to-sink path of G (let us call this P_π) and the product of weights of its edges gives us the monomial $\prod_{i\in[n]} x_{i,\pi(i)}$. This follows from the fact that the weights of source-to-sink paths in graph gadgets G' and G'' is 1, and each path $P_{i,\pi}$ has a weight of $x_{i,\pi(i)}$. By careful observation, we can infer that P_π is 2-stack realizable.

Now for the sake of contradiction, assume there are two different permutations π and π' that get mapped to the same 2-stack realizable path.

$$\prod_{i\in[n]} x_{i,\pi(i)} \neq \prod_{i\in[n]} x_{i,\pi'(n)}$$

Let j be the first index at which π and π' are distinct, i.e., $\forall 1 \leqslant i < j \leqslant n$, $x_{i\pi(i)} = x_{i\pi'(i)}$ and $x_{j\pi(j)} \neq x_{j\pi'(j)}$. According to our construction, in the gadget $G_{map(j)}$, we need to select the $\pi(j)^{th}$ path to get $x_{j,\pi(j)}$ and $\pi'(j)^{th}$ path to get the variable $x_{j,\pi'(j)}$. But each source-to-sink path in G picks precisely one of the n possible paths in each gadget $G_{map(i)}$ (for all $i \in [n]$). By putting together all the aforementioned discussion, we get that every permutation maps to a unique 2-stack realizable path.

□

Lemma 20. *For all $n \in \mathbb{N}$, there exists a $O(n^3)$-sized $S^{[2]}BP$ that computes the Hamiltonian cycle polynomial HC_n.*

Proof. we shall prove the statement of the theorem by constructing a 2-stack branching program for the Hamiltonian cycle polynomial. Let S_1 and S_2 be two stacks. Let T be equal to $\{\langle b, i\rangle \mid b \in \{0, 1\}, i \in [n]\} \cup \{\langle i\rangle \mid i \in [n]\}$. For each stack S_u (for $u \in \{1, 2\}$), let the push and pop operations be denoted by $push_u(s)$ and $pop_u(s)$ for some letter $s \in T$. As before, nop denotes no operation on the stacks.

The gadgets $G_{pop(i)}$, $G_{push(i)}$ (for each $i \in [n]$), G' and G'' are as they were in the proof of theorem 19. We will now construct new graph gadgets $G_{out(i)}$ (for each i) as shown in Figure 4.12, and G_{in} as shown in Figure 4.13.

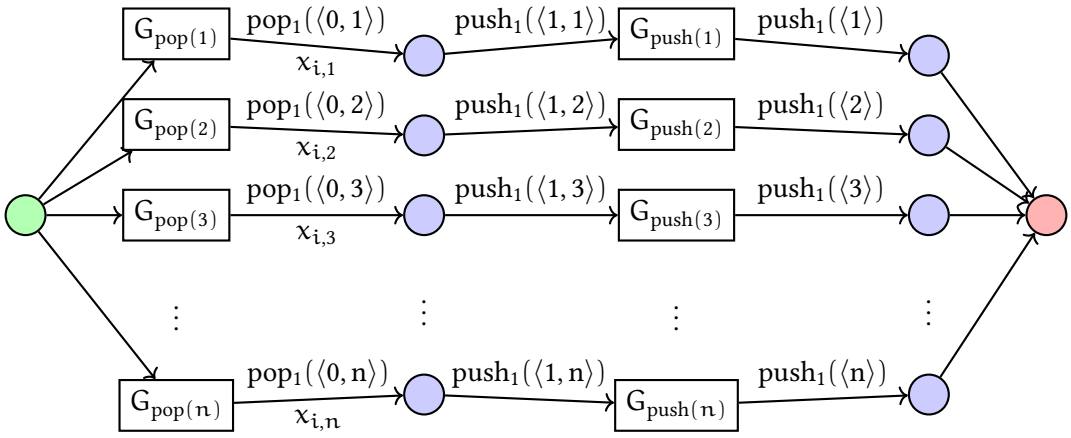

Figure 4.12 Gadget $G_{out(i)}$

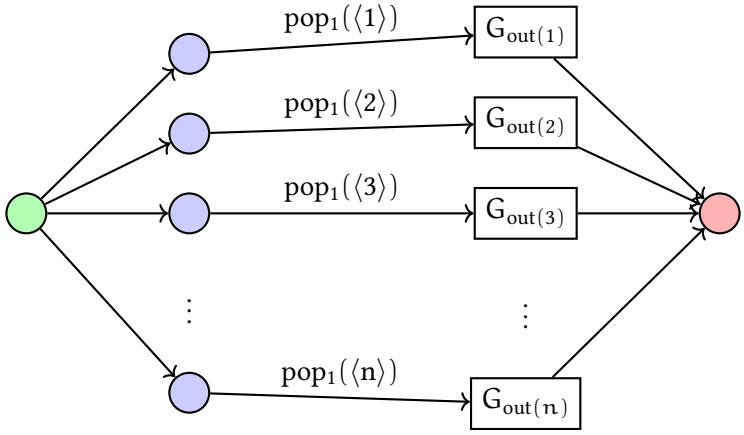

Figure 4.13 Gadget G_{in}

We now get the 2-stack branching program G by placing the graph gadgets G', n many copies of G_{in}, and G'' in series (with connecting edges of weight 1 and no associated stack operations) as shown in Figure 4.14.

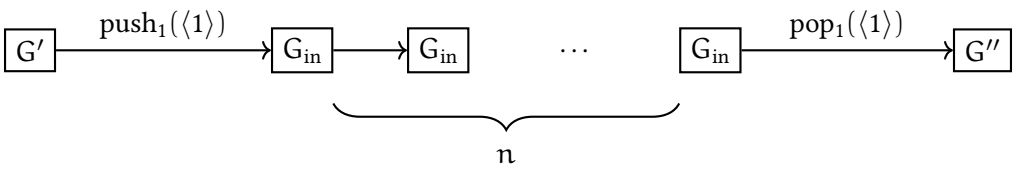

Figure 4.14 Gadget connections for hamiltonian cycle

For every cyclic permutation, there exists a unique 2-stack realizable path: For an arbitrary cyclic permutation $\pi : [n] \mapsto [n]$, let the corresponding monomial be $x_{i_1,i_2}x_{i_2,i_3}x_{i_3,i_4}\dots x_{i_n,i_1}$

where $i_1 = 1$ and $i_2, \ldots, i_{n-1} \in [n] \setminus \{1\}$ are distinct. Note that $\pi(i_j) = i_{j+1}$ (for all $1 \leqslant j \leqslant n-1$) and $\pi(i_{n-1}) = 1$. We will first identify a 2-stack realizable path corresponding to this cyclic permutation.

Note that the construction of G uses n many copies of G_{in} in series. We use the notation $G_{in}^{(j)}$ to refer to the j^{th} copy of G_{in} in series. For each $j \in [n]$ choose a source-to-sink path in the graph gadget $G_{in}^{(j)}$ that goes through the edge with stack operation $\text{pop}(\langle i_j \rangle)$ and in the graph gadget $G_{\text{out}(i_j)}$ choose the 2-stack realizable path that goes through the edge with weight $x_{i_j,\pi(i_j)}$. Let us call the 2-stack realizable path in this graph gadget $G_{in}^{(j)}$ as P_j. Note that at the end of each graph gadget $G_{in}^{(j)}$, the letter $\langle \pi(i_j) \rangle$ is at the top of stack S_1. Thus, in the next gadget $G_{in}^{(j+1)}$ a 2-stack realizable path is the one that pops the head $\langle \pi(i_j) \rangle$ from the stack and takes the stack realizable paths in the corresponding copies of G_{push} and G_{pop}. By taking all these paths $G', P_1, \ldots, P_n, G''$ in series gives us a 2-stack realizable source-to-sink path in G whose weight is equal to $x_{i_1,i_2} x_{i_2,i_3} x_{i_3,i_4} \ldots x_{i_n,i_1}$. It is easy to see that the uniqueness of a source-to-sink path for a given cyclic permutation also follows from the same arguments.

Every 2-stack realizable path maps to a unique cyclic permutation: Gadgets G_{in} and G_{out} ensure that each monomial corresponds to a path. The number of G_{in} gadgets is n; therefore, the monomials correspond to paths of length n. The $\text{push}_1(\langle 1 \rangle)$ operation at the start and the $\text{pop}_1(\langle 1 \rangle)$ operation at the end ensure that the path begins and ends at vertex 1. Gadgets G' and G'' ensure that each vertex is visited exactly once. Putting it all together, it is clear that each 2-stack realizable path corresponds to a Hamiltonian cycle. □

Theorem 21. *A polynomial family (f_n) is in* VNP *if and only if there exists a polynomial sized* $\mathsf{S}^{[2]}\mathsf{BP}$ *that computes it.*

Proof. By combining Lemma 18 with Lemma 19, it is easy to see that the class VNP is exactly characterized by polynomial sized 2-stack branching programs. □

4.2 k-Stack Branching Programs

In the previous section, it was shown that the class VNP is exactly characterized by branching programs with 2 stacks. A natural question that stems from that is the computation power of branching programs with more than 2 stacks. In this section, we show that any $\mathsf{S}^{[k]}\mathsf{BP}$ can be converted to a $\mathsf{S}^{[2]}\mathsf{BP}$ with only a polynomial blowup in size (Theorem 22). As a corollary, we get that any polynomial computed by a polynomial sized $\mathsf{S}^{[k]}\mathsf{BP}$, where $k > 2$, can also be computed by a polynomial sized $\mathsf{S}^{[2]}\mathsf{BP}$ (Corollary 23).

Theorem 22. *Let $p(x_1, \ldots, x_n)$ be a polynomial computed by an $\mathsf{S}^{[k]}\mathsf{BP}$ of size s, path length bounded by m and stack alphabet set $T = \{0, 1\}$. Then, $p(x_1, \ldots, x_n)$ can also be computed by an $\mathsf{S}^{[2]}\mathsf{BP}$ of size $O(mks \log(k))$ and path length $O(m^2 \log(k))$.*

Proof. Given a $S^{[k]}BP$ G, we convert it into a $S^{[2]}BP$ by performing the steps detailed below.

The stack alphabet set T is updated to the set $\cup_{i=1}^{k}\{\langle 0,i\rangle, \langle 1,i\rangle\}$ and $\forall i \in [k], op_i(b)$, where $op_i \in \{push_i, pop_i\}$ & $b \in \{0,1\}$ is replaced by $op_i(\langle b,i\rangle)$. Next we replace $push_i(\langle b,i\rangle)$ with $push_2(\langle b,i\rangle)$, $\forall i \in \{3,4,\ldots,k\}$ & $b \in \{0,1\}$. It is easy to see that the pop_i operations cannot be replaced with pop_2 operations similarly. To enable this, we make use of the gadgets $G_{2\to 1/i}$ (Figure 4.16) and $G_{1\to 2/i}$ (Figure 4.18).

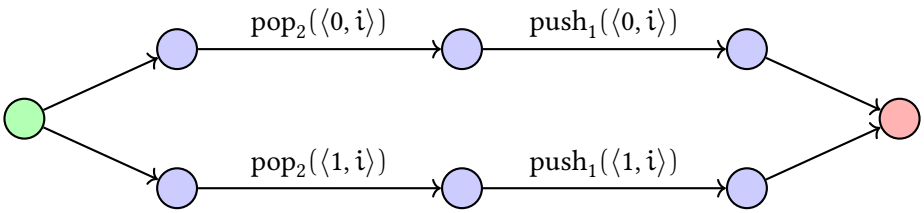

Figure 4.15 Gadget $G_{2\to 1(i)}$

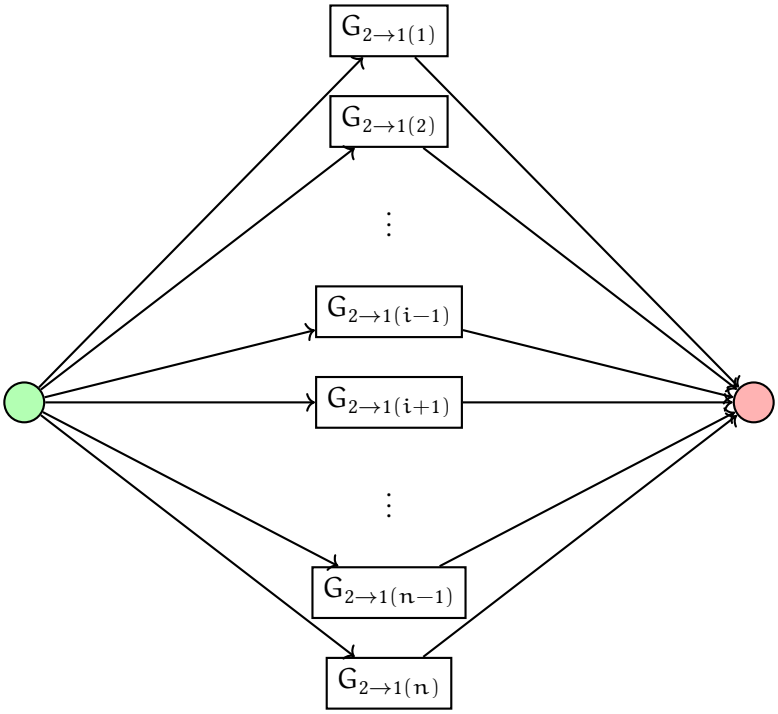

Figure 4.16 Gadget $G_{2\to 1/i}$

Let there be an edge from vertex in layer m' to vertex in layer $m'+1$ with weight $wt(e)$ and stack operation $pop_i(\langle b,i\rangle)$ where $b \in \{0,1\}$. Let the height of the second stack, at this stage, be h and let the stack symbol $\langle b,i\rangle$ appear in the second stack at some height $h' \leqslant h$. The edge is then replaced by the gadget in Figure 4.19.

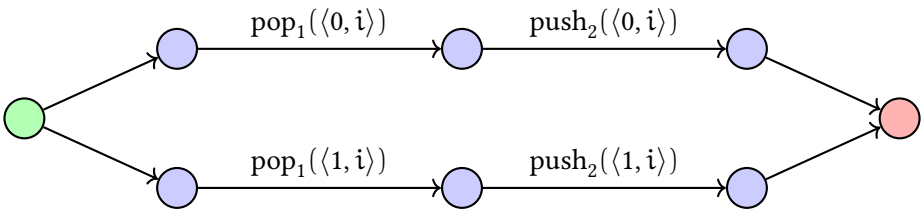

Figure 4.17 Gadget $G_{1\to 2(i)}$

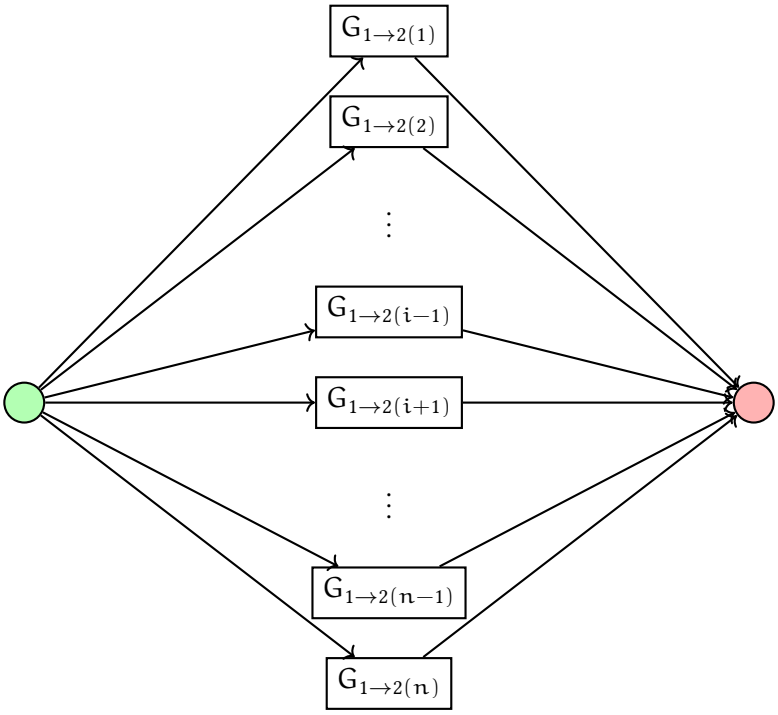

Figure 4.18 Gadget $G_{1\to 2/i}$

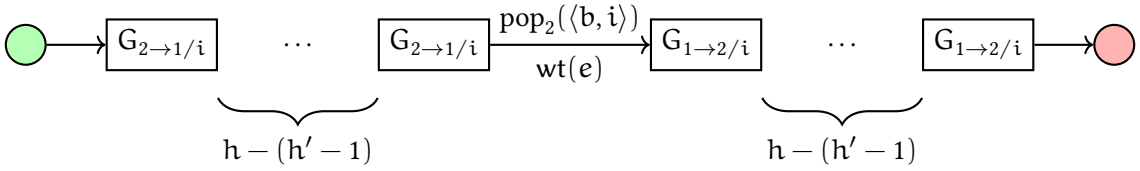

Figure 4.19 Gadget connections

We know that $h \leqslant m'$, therefore $2[h-(h'-1)] \leqslant 2m'$. An upper bound on the new path length is given by $2\sum_{i=1}^{m} i = m(m+1)$. The upperbound on size of the $S^{[2]}BP$ becomes $8mks$.

Next we reduce size of the stack alphabet set from $2k$ to 2 by using binary representation for each stack alphabet. To do this, we replace every edge with push or pop stack operations with $\log(2k)$

many edges connected in series. This causes the bound on path length to become $m(m+1)\log(2k)$ and bound on size of the $S^{[2]}BP$ to become $8mks\log(2k)$.

□

Corollary 23. *Any polynomial family* $p(x_1,\ldots,x_n)$ *computed by a polynomial sized* $S^{[k]}BP$ *where* $k \in \mathsf{poly}(n)$ *can also be computed by a polynomial sized* $S^{[2]}BP$.

4.3 Queue Branching Programs

The study by Mengel [Men13] and the above sections together give a complete picture of the computational power of k-stack branching programs ($\forall k \in \mathbb{N}$). We know that any set of operations performed on two stacks can be efficiently simulated by a queue data structure and vice versa. Therefore, an analogous question to ask in this setting would be whether a queue branching program exactly characterizes the class VNP. In this section we answer this question in the affirmative.

Lemma 24. *Any polynomial family computed by a polynomial sized* $S^{[2]}BP$ *is inside* VNP.

The proof of this lemma is along the same lines of the proof for Lemma 18.

Lemma 25. *For all* $n \in \mathbb{N}$, *there exists a* $O(n^3)$ *sized* QBP *that computes the permanent polynomial* Perm_n.

Proof. Let Q be the queue. Let T be equal to $\{\langle b,i\rangle \mid b \in \{0,1\}, i \in [n]\}$. Let the insert and remove operations be denoted by insert(s) and remove(s) for some letter $s \in T$. As before, nop denotes no operation on the queue. Recall that each edge of the underlying algebraic branching program has a weight which is a linear polynomial, and associated queue operations.

Let G' and G'' be directed path graphs as shown in Figure 4.20 and Figure 4.21 respectively. Every edge has a weight 1 in both of these graphs.

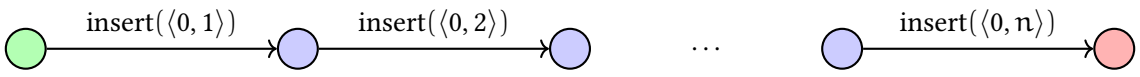

Figure 4.20 Gadget G'

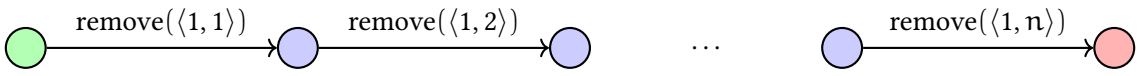

Figure 4.21 Gadget G''

For each $j \in [n]$, the source-to-sink graph $G_{(j)}$ is as shown in Figure 4.22 with each edge having weight 1.

For each $i \in [n]$, the gadget $G_{\mathsf{remove}(i)}$ is constructed by connecting the sink of $G_{(1)}$ with the source of $G_{(2)}$, sink of $G_{(2)}$ with source of $G_{(3)}$ and so on till the sink of $G_{(i-2)}$ is connected to the source

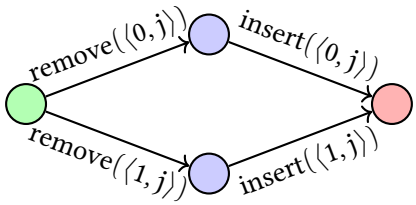

Figure 4.22 Gadget $G_{(j)}$

of $G_{(i-1)}$ as shown in Figure 4.23. All edges used for these connections have weight 1 and queue operation nop.

Figure 4.23 Gadget $G_{\text{remove}(i)}$

For each $i \in [n]$, the gadget $G'_{\text{remove}(i)}$ is constructed by connecting the sink of $G_{(i+1)}$ to the source of $G_{(i+2)}$, sink of $G_{(i+2)}$ with source of $G_{(i+3)}$ and so on till the sink of $G_{(n-1)}$ is connected to the source of $G_{(n)}$ as shown in Figure 4.24. All edges used for these connections have weight 1 and queue operation nop.

Figure 4.24 Gadget $G'_{\text{remove}(i)}$

For each $i \in [n]$, let the source-to-sink graph $G_{\text{map}(i)}$ be constructed as shown in Figure 4.25. The source and sink vertices of $G_{\text{map}(i)}$ are connected by n disjoint paths such that the j^{th} path (for all $1 \leqslant j \leqslant n$) is obtained by connecting the source vertex, gadget $G_{\text{remove}(j)}$, a single vertex (say $v_{\text{map}(i),j}$), gadget $G'_{\text{remove}(j)}$ and sink vertex in series. All edges have weight 1 and queue operation nop except for the edge connecting sink of gadget $G_{\text{remove}(i)}$ to $v_{\text{map}(i),j}$ and the edge connecting $v_{\text{map}(i),j}$ to source of $G'_{\text{remove}(i)}$. The edge connecting sink of $G_{\text{remove}(i)}$ to $v_{\text{map}(i),j}$ has weight x_{ij} and queue operation remove$(\langle 0, j\rangle)$. The edge connecting $v_{\text{map}(i),j}$ to source of $G'_{\text{remove}(i)}$ has weight 1 and queue operation insert$(\langle 1, j\rangle)$.

We now claim that the QBP G computes the polynomial Perm_n by placing the gadgets G', $G_{\text{map}(1)}$, …, $G_{\text{map}(n)}$, G'' in series with edges having weight 1 and queue operation nop as shown in Figure 4.26.

To prove afore mentioned claim, it is sufficient to prove that there exists a bijection between queue realizable paths in G and monomials corresponding to permutations.

Every queue realizable path maps to a unique permutation: For all $i \in [n]$, the weight of the j^{th} source-to-sink path in graph gadget $G_{\text{map}(i)}$ is $x_{i,j}$. Therefore, any source-to-sink path in graph

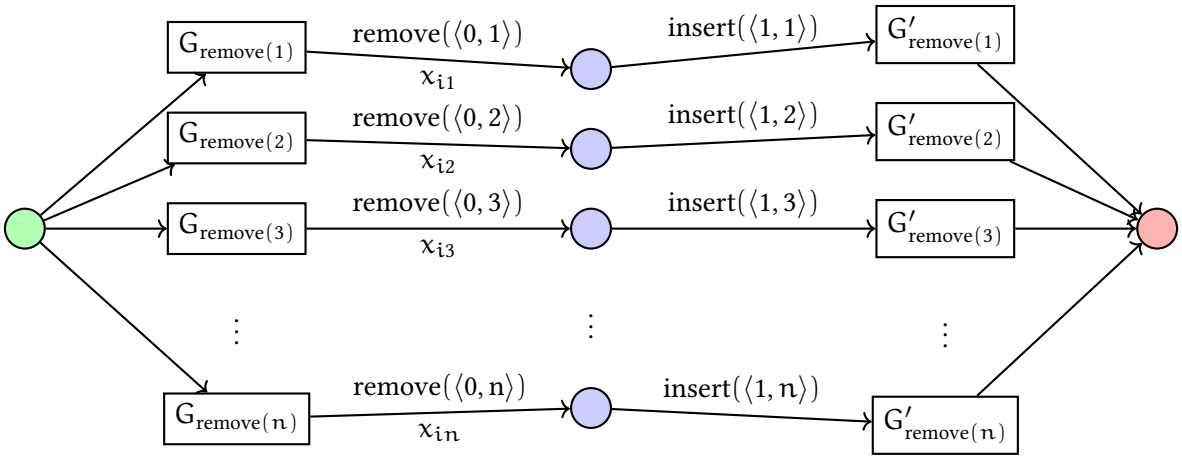

Figure 4.25 Gadget $G_{map(i)}$

Figure 4.26 Gadget connections for QBP computing permanent

G will compute a monomial of the following form.

$$\prod_i x_{i,j_i} \text{ where } j_i \in [n] \text{ for all } i \in [n].$$

We now show that for any queue realizable path, the monomial computed corresponds to a permutation. Let this monomial contain both x_{ik} and $x_{ik'}$ ($i \neq i'$). This implies that the path with remove($\langle 0,k\rangle$) queue operation in graph gadget $G_{map(i)}$ and the path with remove($\langle 0,k\rangle$) queue operation in graph gadget $G_{map(i')}$ are chosen. Once $\langle 0,k\rangle$ is removed from the queue, it is never inserted back in. Therefore, any such source-to-sink path in G is not queue realizable. Assume that the monomial computed contains both x_{ik} and $x_{ik'}$. This is possible only if the path containing x_{ik} and the path containing $x_{ik'}$ in $G_{map(i)}$ are chosen. Since this is clearly not possible, such a source-to-sink path is not queue realizable.

For the sake of contradiction, assume that two different queue realizable paths map to the same permutation. Let the graph gadget $G_{map(i)}$ be where the paths diverge. Let one path take the j^{th} edge and the other path take the j'^{th} edge. Clearly, one path outputs a monomial with x_{ij} in it and the other path outputs a monomial with $x_{ij'}$ in it. Both paths do not output the same monomial. Putting everything together, we see that every queue realizable path gets mapped to a unique permutation.

For every permutation, there exists a unique queue realizable path: Let $\pi : [n] \mapsto [n]$ be an arbitrary permutation and $\pi(i)$ denote the value i gets mapped to. Firstly, given a permutation, we identify a queue realizable path that computes the monomial corresponding to the permutation. For all $i \in [n]$, in graph gadget $G_{map(i)}$ select the π^{th} path within the graph gadget (let this path be

$P_{i,\pi}$). Now we get G', $P_{1,\pi}$, …, $P_{n,\pi}$, G'' as the source-to-sink path P_π in graph G. Every edge in each $P_{i,\pi}$ contributes a weight of $x_{i,\pi(i)}$. All other edges in P_π have weight 1. Therefore, the monomial computed by source-to-sink path P_π is $\prod_{i\in\pi} x_{i,\pi(i)}$. By careful observation, we can infer that P_π is queue realizable.

For the sake of contradiction, assume that two permutations π and π' get mapped to the same queue realizable path.

$$\prod_{i\in[n]} x_{i,\pi(i)} \neq \prod_{i\in[n]} x_{i,\pi'(n)}$$

Let j be the first index at which π and π' are distinct i.e., $\forall 1 \leqslant i < j \leqslant n$, $x_{i\pi(i)} = x_{i\pi'(i)}$ and $x_{j\pi(j)} \neq x_{j\pi'(j)}$. Given a permutation, to get the corresponding queue realizable path, we need to select the path $P_{j,\pi}$ in $G_{\text{map}(j)}$ to get $x_{j,\pi(j)}$ and the path $P_{j,\pi'}$ in $G_{\text{map}(j)}$ to get $x_{j,\pi'(j)}$. Since we select exactly one path in graph gadget $G_{\text{map}(i)}$ (for all $i \in [n]$), we cannot get the same queue realizable path for both permutations.

□

Theorem 26. *A polynomial family* (f_n) *is in* VNP *if and only if there exists a polynomial sized* QBP *that computes it.*

Proof. By combining Lemma 24 with Lemma 25, we get that the class VNP is exactly characterized by polynomial sized queue branching programs. □

Chapter 5

A closer look at stack branching programs

5.1 Restricting stack height

In this section, we place asymptotic bounds on the height of the stacks used and look at the computing power of k-stack branching programs. We show that $\mathsf{S}^{[\text{const}]}\mathsf{BP}$ with logarithmic bound on stack height and $\mathsf{S}^{[\log]}\mathsf{BP}$ with constant bound on stack height are equivalent and exactly characterize the class VNP (Corollary 28). We also improve the characterization given by [Men13] and show that the class VP is characterized by polynomial-sized stack branching programs with polylogarithmic bound on stack height ().

Theorem 27. *Let* $p(x_1, \ldots, x_n)$ *be a polynomial computed by a* $\mathsf{S}^{[k]}\mathsf{BP}$ *of size* s *with the stack height bounded by* H*, path length bounded by* m *and* $T = \{0, 1\}$*. Then the polynomial* $p(x_1, \ldots, x_n)$ *can also be computed by an* ABP *of size* $2^{kH}s$.

Proof. The ABP is constructed by enumerating all possible stack configurations of the $\mathsf{S}^{[k]}\mathsf{BP}$ and adding a vertex for each one. Then edges are added to make sure that there is a bijection between source-to-sink paths in the ABP and realizable source-to-sink paths in the $\mathsf{S}^{[k]}\mathsf{BP}$.

Since each stack's height is bounded by H, the total number of stack configurations possible are $\sum_{i=0}^{H}(2^i)^k = (2^{k(H+1)} - 1)/(2^k - 1)$. Let this value be denoted by γ. For each vertex v_i in the $\mathsf{S}^{[k]}\mathsf{BP}$ add the set of vertices $\{v_{i,j}\}_{j=1}^{\gamma}$.

Since there are γ many possible stack configurations, we assign a unique integer from $[\gamma]$ to each configuration. Let the configuration where all stacks are empty be represented by S_0.

Let P_i be a realizable source-to-sink path in the $\mathsf{S}^{[k]}\mathsf{BP}$ given by the set of edges $(e_1, \ldots, e_m)$. The stack configuration at each vertex in the path is given by the set $(S_0, S_1, \ldots, S_m = S_0)$. For each edge $e_r = (v_j, v_{j'})$, add an edge to the ABP connecting the vertex $v_{j,S_{r-1}}$ and v_{j',S_i} where S_{r-1} is the stack configuration at vertex v_j and S_r is the stack configuration at vertex $v_{j'}$. This is repeated for all realizable source-to-sink paths in the $\mathsf{S}^{[k]}\mathsf{BP}$.

It is straightforward to see the bijection between realizable source-to-sink paths in the $\mathsf{S}^{[k]}\mathsf{BP}$ and source-to-sink paths in the constructed ABP. Since we place γ many vertices in the ABP for each vertex in the $\mathsf{S}^{[k]}\mathsf{BP}$, the size of the ABP is $O(2^{kH}s)$.

□

Corollary 28.

1. *Let* $p(x_1, \ldots, x_n)$ *be a polynomial that can be computed by a polynomial sized* $S^{[k]}BP$, $k \in \mathbb{N}$ *and stack height is bounded by a function in* $O(\log(n))$. *Then* $p(x_1, \ldots, x_n)$ *can also be computed by a polynomial sized* ABP.

2. *Let* $p(x_1, \ldots, x_n)$ *be a polynomial that can be computed by a polynomial sized* $S^{[k]}BP$, $k \in O(\log(n))$ *and stack height is bounded by a constant. Then* $p(x_1, \ldots, x_n)$ *can also be computed by a polynomial sized* ABP.

The above corollary shows a neat relationship between the bound on stack height and the number of stacks needed to characterize VBP.

5.2 Restricting branching program width

In this section, we study the computational power of width-2 $S^{[k]}BP$ with additional restrictions on stack height. We give a fine-grained analysis of the blowup that occurs in converting a width-t ABP to a width-2 SBP (Lemma 29).

Lemma 29. *Let* (G, u_s, u_t) *be an* ABP *with* k *layers and atmost* t *vertices in each layer. Let the* ABP *compute the polynomial* $p(x_1, x_2, \ldots, x_n)$. *Then the polynomial* $p(x_1, x_2, \ldots, x_n)$ *can also be computed by a width-*2 SBP $(G', v_s, v_t, T = \{0, 1\})$ *of size bounded by* $O(\log(kt^2)kt^3)$ *and logarithmically bounded stack height.*

Proof. The SBP G' is constructed as detailed below. Each edge e in G is replaced by 2 edges with stack labels as shown in Figure 5.1.

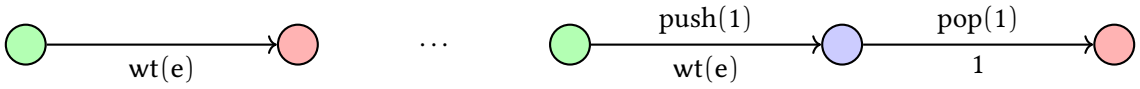

Figure 5.1 Converting ABP to SBP

Now, we have an SBP of arbitrary width over a unary stack alphabet computing the polynomial $p(x_1, x_2, \ldots, x_n)$.

$$\#\text{Edges in the ABP} = (k-1)t^2 + 2t$$
$$\#\text{Edges in the SBP} = 2[(k-1)t^2 + 2t]$$
$$\#\text{Edge pairs sharing a common vertex} \leqslant 2[(k-1)t^2 + 2t]t$$

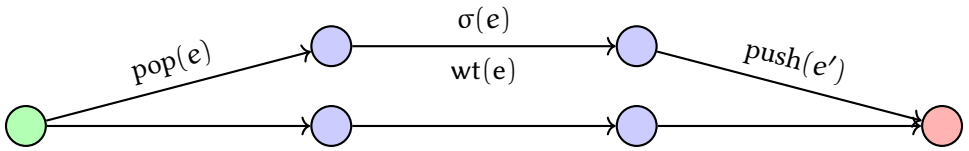

Figure 5.2 Gadget for reducing width [Men13]

Next, we reduce the width of the SBP to 2 by constructing a gadget for each edge pair as shown by [Men13] using the gadget in Figure 5.2. After the width reduction, we end-up with a width-2 SBP of size $O(kt^3)$ over stack alphabet of size $O(\log(kt^2))$.

Finally, we represent each element in the stack alphabet set using a binary string to reduce the stack alphabet set size to 2. This causes the size of the SBP to become $O(\log(kt^2)kt^3)$.

□

Corollary 30. *Let $p(x_1, x_2, \ldots, x_n)$ be a polynomial that is computed by a formula of polynomial size. Then, it can also be computed by a* SBP *of size $O(4^d d)$ over binary alphabet with logarithmic-ally bounded stack height.*

Proof. Due to [BOC92], we know that any polynomial computed by a formula of depth d can also be computed by a width-3 ABP of length 4^d. Setting $t = 3$ and $k = 4^d$ in Lemma 29, we get the size of the SBP to be $O(d^d d)$. Furthermore, due to [Bre74], we know that $d = \log(n)$. Therefore, size of the SBP becomes $O(n^2 \log(n))$. □

Chapter 6

Exponential summation

The class VNP is defined as all polynomial families that are given by exponential summation of polynomial families in class VP (see Definition 10). Since VP is characterized by polynomial-sized SBP and VNP is characterized by polynomial-sized $\mathsf{S}^{[2]}\mathsf{BP}$, an exponential summation of a polynomial-sized SBP should give a polynomial-sized $\mathsf{S}^{[2]}\mathsf{BP}$. In this chapter, we explicitly show how to construct an $\mathsf{S}^{[2]}\mathsf{BP}$ that computes exponential summation of a given SBP.

6.1 Simulating exponential summation using stacks

Lemma 31. *Given a* $\mathsf{S}^{[k]}\mathsf{BP}$ $(k \geqslant 1)$ *of size* $s \in \mathsf{poly}(n)$ *computing the polynomial* $q_n(x_1, \ldots, x_n)$*, there exists a* $\mathsf{S}^{[k+1]}\mathsf{BP}$ *of size* $s' \in \mathsf{poly}(n)$ *computing*

$$p_n(x_1, \ldots, x_n) = \sum_{(e_1, \ldots, e_m) \in \{0,1\}^m} q_{n+m}(x_1, \ldots, x_n, e_1, \ldots, e_m) \qquad (m \in \mathsf{poly}(n))$$

Proof. We prove the lemma by showing how to construct the $\mathsf{S}^{[k+1]}\mathsf{BP}$ computing $p_n(x_1, \ldots, x_n)$. We begin by noticing that the $\mathsf{S}^{[k]}\mathsf{BP}$ computing $q_{n+m}(x_1, \ldots, x_n, e_1, \ldots, e_m)$ is of size $\mathsf{poly}(n, m)$. Since $m \in \mathsf{poly}(n)$, the size can simply be written as $\mathsf{poly}(n)$. Denote the $\mathsf{S}^{[k]}\mathsf{BP}$ computing $q_{n+m}(x_1, \ldots, x_n, e_1, \ldots, e_m)$ by G.

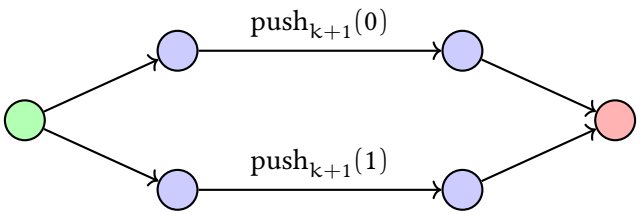

Figure 6.1 Gadget $G_{\mathtt{push}(0,1)}$

We use the gadget $G_{\mathtt{init}}$ to go over all 2^m possible settings of $(e_1, \ldots, e_m)$. Construct a $\mathsf{S}^{[k+1]}\mathsf{BP}$ G' from G, using gadgets G_{init} and G_{reset} as shown in Figure 6.5.

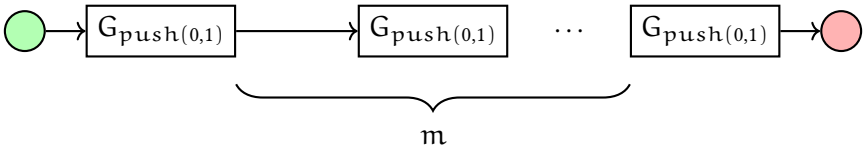

Figure 6.2 Gadget G_{init}

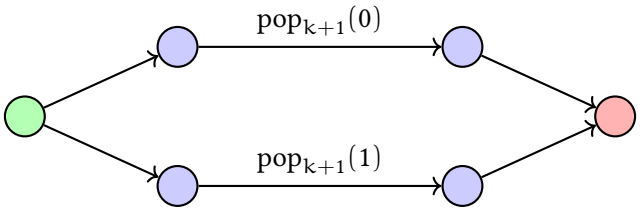

Figure 6.3 Gadget $G_{pop(0,1)}$

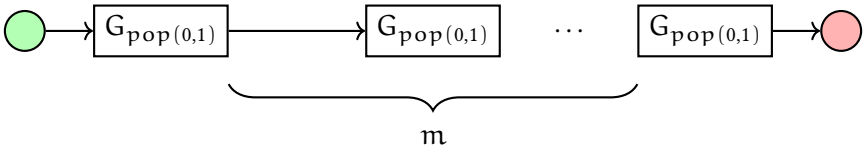

Figure 6.4 Gadget G_{reset}

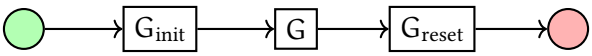

Figure 6.5 $S^{[k+1]}$BP G'

The idea is that the setting of $(e_1, \ldots, e_m)$ is available in the $k+1^{th}$ stack. If we are able to use the information in the stack to assign values to the e_i variables, then we are done. To achieve this, we make use of gadget $G_{(i)}$ in Figure 6.7 and gadget $G'_{(i)}$ in Figure 6.9.

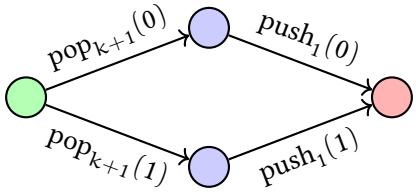

Figure 6.6 Gadget $G_{k+1\to 1}$

Any edge in the $S^{[k+1]}$BP G' with edge weight e_i $(i \in [m])$ is replaced as shown in Figure 6.10.

The $k+1^{th}$ stack contains the binary configuration and by looking at the i^{th} position in the configuration, we can find out the value to be assigned to e_i. The procedure to check the stack element at the i^{th} position is shown in Figure 6.10. The $i-1$ elements at the top of the $k+1^{th}$ stack are moved to the first stack. Next, if a 0 is popped, then we add an edge with weight 0. If a 1 is popped, an edge

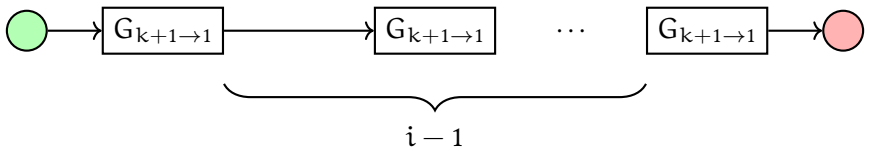

Figure 6.7 Gadget $G_{(i)}$

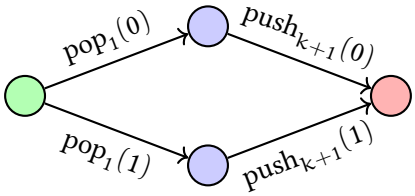

Figure 6.8 Gadget $G_{1\to k+1}$

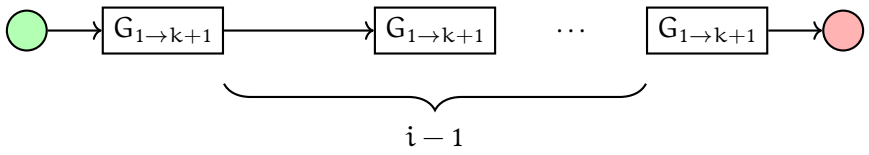

Figure 6.9 Gadget $G'_{(i)}$

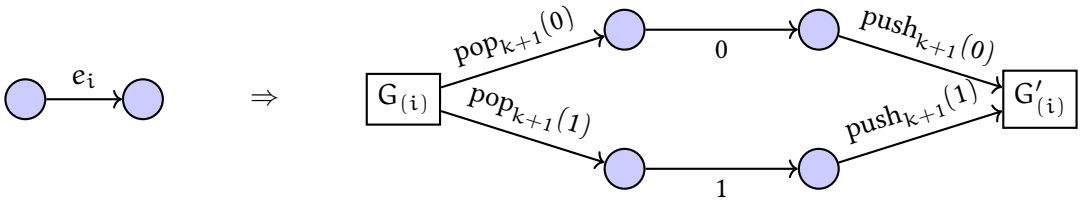

Figure 6.10 Initializing e_i with value

with weight 1 is added. Finally, all the elements removed from $k+1^{th}$ stack are returned back to it. This way, we have effectively substituted either a zero or a one to the variable e_i.

□

By setting the value of k to 1 in Lemma 31, we get the required construction of a polynomial-sized $S^{[2]}BP$ computing exponential sum of a given polynomial-sized SBP.

Corollary 32. *Let $q_n(x_1,\ldots,x_n)$ be a polynomial family in class* VNP. *Then, the polynomial family given by*

$$p_n(x_1,\ldots,x_n) = \sum_{(e_1,\ldots,e_m)\in\{0,1\}^m} q_{n+m}(x_1,\ldots,x_n,e_1,\ldots,e_m) \qquad (m \in \mathrm{poly}(n))$$

is also in the class VNP *i.e., class* VNP *is closed under exponential summation.*

Proof. Setting $k \in \text{poly}(n)$ in Lemma 31 coupled with Corollary 23 proves that the class VNP is closed under exponential summation. □

Corollary 33. *Let* $q_n(x_1, \ldots, x_n)$ *be a polynomial family in class* VBP. *Then, the polynomial family given by*

$$p_n(x_1, \ldots, x_n) = \sum_{(e_1,\ldots,e_m)\in\{0,1\}^m} q_{n+m}(x_1, \ldots, x_n, e_1, \ldots, e_m) \qquad (m \in \text{poly}(n))$$

is in the class VNP.

Proof. We take the polynomial-sized ABP computing $q_n(x_1, \ldots, x_n)$ and follow the procedure detailed in the proof of Lemma 31. Since the ABP has zero stacks, we need to introduce two new stacks (gadgets $G_{(i)}$ and $G'_{(i)}$ use 2 stacks). At the end of the procedure, we end up with a polynomial-sized $S^{[2]}BP$ computing $p_n(x_1, \ldots, x_n)$. □

While the statements of Corollary 32 and Corollary 33 are well known and have been proved earlier, the proofs provided in this work are stack branching program based. The well known way of showing exponential summation of a polynomial in VBP gives a polynomial in VNP is as follows. We first show that exponential summation of a polynomial in VF gives a polynomial in VNP ([Val82]). Following this, since VF $\subseteq$ VBP $\subseteq$ VP, we get $\sum$VF $=$ $\sum$VBP $=$ $\sum$VP $=$ VNP. To the author's limited knowledge, this work contains the first direct proof of the fact $\sum$VBP $=$ VNP.

Chapter 7

Hardness of computing permanent

7.1 Permanent under row operations

Claim 34. *Let A be a square matrix of size n and a_{ij} represent the $(i,j)^{th}$ entry in A. Let c be a constant. Let A' be the matrix obtained by performing the row operation $R_j \leftarrow cR_i + R_j$ on A and A'' be the matrix obtained by performing the row operation $R_j \leftarrow cR_i$ on A. Then the following holds*

$$\mathsf{Perm}_n(A') = \mathsf{Perm}_n(A) + c\,\mathsf{Perm}_n(A'')$$

Proof. By definition,

$$\mathsf{Perm}_n(A) = \sum_{\sigma\in S_n}\prod_{l\in[n]} a_{l\sigma(l)} \tag{7.1}$$

$$\mathsf{Perm}_n(A'') = \sum_{\sigma\in S_n}\prod_{l\in[n]} a''_{l\sigma(l)} \tag{7.2}$$

$$= c\sum_{\sigma\in S_n} a_{i\sigma(i)}\prod_{l\in[n];l\neq j} a_{l\sigma(l)} \tag{7.3}$$

$$\tag{7.4}$$

$$\mathsf{Perm}_n(A') = \sum_{\sigma\in S_n}\prod_{l\in[n]} a'_{l\sigma(l)} \tag{7.5}$$

$$= \sum_{\sigma\in S_n}(a_{j\sigma(j)} + ca_{i\sigma(i)})\prod_{l\in[n];l\neq j} a_{l\sigma(l)} \tag{7.6}$$

$$= \sum_{\sigma\in S_n}\prod_{l\in[n]} a_{l\sigma(l)} + c\sum_{\sigma\in S_n} a_{i\sigma(i)}\prod_{l\in[n];l\neq j} a_{l\sigma(l)} \tag{7.7}$$

$$= \mathsf{Perm}_n(A) + c\,\mathsf{Perm}_n(A'') \tag{7.8}$$

□

7.2 Reduction

We first start with a simple case where A is a square matrix of size n with entries from the set $S = \{-1, 0, 1\}$. The square matrix A' of size $2n$ is constructed as follows

$$A' = \left[\begin{array}{cccc|cccc} & & & & 0 & 0 & \cdots & 0 \\ & & & & 0 & 0 & \cdots & 0 \\ & A & & & \vdots & \vdots & \ddots & \vdots \\ & & & & 0 & 0 & \cdots & 0 \\ \hline 1 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 1 \end{array}\right]_{2n \times 2n} \tag{7.9}$$

Remove all entries in the top-left block of matrix A' that are from the set $S/\{0, 1\}$ by performing row operations of the form $R_j \leftarrow cR_i + R_j$ where $i \in [2n]/[n]$ and $j \in [n]$. Denote the matrix obtained by A''. It is easy to see that atmost n^2 many row operations are needed to ensure that entries of matrix A'' are only from $\{0, 1\}$. Using Claim 34, $\mathsf{Perm}_{2n}(A'')$ can be written as sum of atmost n^2 many permanents of matrices with entries from $\{-1, 0, 1\}$. If the permanent of a matrix with $\{-1, 0, 1\}$ entries can be computed in time T, then permanent of a matrix with $\{0, 1\}$ entries can be computed in time $O(n^2T)$.

We encounter a problem when we directly use this technique on a matrix with entries from a generic set $S \subseteq \mathbb{Z}$. The row operations cause the top-right block to have entries that are neither 0 nor 1 and the set S may not contain 0. Firstly, we restrict ourselves to set S containing zero. Next, we make the following change to the construction of the matrix A'. Let I be the identity matrix, O be a matrix with all entries being zero and k be the member of set S with the largest absolute value. A' is constructed as follows

$$A' = \left[\begin{array}{c|cccc} A & O & O & \cdots & O \\ \hline I & I & O & \cdots & O \\ I & O & I & \cdots & O \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ I & O & O & \cdots & I \end{array}\right]_{(k+1)n \times (k+1)n} \tag{7.10}$$

Using this matrix, we can now convert all entries to 0 or 1 while also ensuring that the entries in the top-right block do not go above 1. Now, given an algorithm, that runs in time T, for computing permanent of matrices with entries in S, the permanent of matrix with $\{0, 1\}$ entries can be computed in time kn^2T.

Chapter 8

Discussion

General setting

- Section 6.1 gave an explicit way to construct a polynomial sized $\mathsf{S}^{[2]}\mathsf{BP}$ computing the exponential sum of a polynomial computed by a polynomial sized SBP. Giving a construction for the other way i.e., given a polynomial sized $\mathsf{S}^{[2]}\mathsf{BP}$, we construct the corresponding polynomial sized SBP.
- Reinterpreting other existing structural results in terms of stack branching programs might give new insights into the algebraic complexity classes.

Monotone setting

- We believe that monotone SBP is strictly more powerful than mVP since SBPs can get rid of monomials without actually using negations. Can explicit polynomial sized SBP constructions be given for polynomials not in mVP.
- Another interesting line of research is to investigate if there is a separation between SBP and monotone SBP.
- The Perm polynomial can still be computed by a polynomial sized monotone $\mathsf{S}^{[2]}\mathsf{BP}$. This shows that monotone $\mathsf{S}^{[2]}\mathsf{BP}$ is strictly more powerful that mVNP. In fact, among the known monotone algebraic complexity classes, $\mathsf{mVP}_{\mathsf{proj}}$ is the only class known to be able to compute Perm (see [CGT23] for more details). We suspect that $\mathsf{S}^{[2]}\mathsf{BP}$ exactly characterizes $\mathsf{mVP}_{\mathsf{proj}}$. Can it be shown that the class $\mathsf{mVP}_{\mathsf{proj}}$ is exactly characterized by $\mathsf{S}^{[2]}\mathsf{BP}$.

Bibliography

[BOC92] Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, 1992. doi:10.1137/0221006. 3, 30

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.*, 21:201–206, 1974. doi:10.1145/321812.321815. 30

[CDGM22] Arkadev Chattopadhyay, Rajit Datta, Utsab Ghosal, and Partha Mukhopadhyay. Monotone complexity of spanning tree polynomial re-visited. In *13th Innovations in Theoretical Computer Science Conference*, volume 215 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 39, 21. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2022. 2

[CGT23] Prerona Chatterjee, Kshitij Gajjar, and Anamay Tengse. Monotone Classes Beyond VNP. In *43rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2023)*, volume 284 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSTTCS.2023.11, doi:10.4230/LIPIcs.FSTTCS.2023.11. 37

[CLP21] Prasad Chaugule, Nutan Limaye, and Shourya Pandey. Variants of the determinant polynomial and the VP-completeness. In *Computer science—theory and applications*, volume 12730 of *Lecture Notes in Comput. Sci.*, pages 31–55. Springer, Cham, [2021] ©2021. URL: https://doi.org/10.1007/978-3-030-79416-3_3, doi:10.1007/978-3-030-79416-3_3. 4

[DMPY12] Zeev Dvir, Guillaume Malod, Sylvain Perifel, and Amir Yehudayoff. Separating multilinear branching programs and formulas. In *STOC'12—Proceedings of the 2012 ACM Symposium on Theory of Computing*, pages 615–623. ACM, New York, 2012. doi:10.1145/2213977.2214034. 3

[HY09] Pavel Hrubeš and Amir Yehudayoff. Monotone separations for constant degree polynomials. *Inform. Process. Lett.*, 110(1):1–3, 2009. doi:10.1016/j.ipl.2009.09.003. 2

[HY13] Pavel Hrubeš and Amir Yehudayoff. Formulas are exponentially stronger than monotone circuits in non-commutative setting. In *2013 IEEE Conference on Computational*

Complexity—CCC 2013, pages 10–14. IEEE Computer Soc., Los Alamitos, CA, 2013. doi:10.1109/CCC.2013.11. 2

[HY16] Pavel Hrubeš and Amir Yehudayoff. On isoperimetric profiles and computational complexity. In *43rd International Colloquium on Automata, Languages, and Programming*, volume 55 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 89, 12. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2016. 2

[LMS16] Nutan Limaye, Guillaume Malod, and Srikanth Srinivasan. Lower bounds for non-commutative skew circuits. *Theory Comput.*, 12:Paper No. 12, 38, 2016. doi:10.4086/toc.2016.v012a012. 2

[Mal11] Guillaume Malod. Succinct algebraic branching programs characterizing non-uniform complexity classes. In *Fundamentals of computation theory*, volume 6914 of *Lecture Notes in Comput. Sci.*, pages 205–216. Springer, Heidelberg, 2011. URL: https://doi.org/10.1007/978-3-642-22953-4_18, doi:10.1007/978-3-642-22953-4_18. 4

[Men13] Stefan Mengel. Arithmetic branching programs with memory. In *Mathematical foundations of computer science 2013*, volume 8087 of *Lecture Notes in Comput. Sci.*, pages 667–678. Springer, Heidelberg, 2013. URL: https://doi.org/10.1007/978-3-642-40313-2_59, doi:10.1007/978-3-642-40313-2_59. iv, 4, 9, 13, 14, 24, 28, 30

[MST16] Meena Mahajan, Nitin Saurabh, and Sébastien Tavenas. VNP = VP in the multilinear world. *Inform. Process. Lett.*, 116(2):179–182, 2016. doi:10.1016/j.ipl.2015.08.004. 3

[Nis91] Noam Nisan. Lower bounds for non-commutative computation. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 410–418, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/103418.103462. 2

[Sri20] Srikanth Srinivasan. Strongly exponential separation between monotone VP and monotone VNP. *ACM Trans. Comput. Theory*, 12(4):Art. 23, 12, 2020. doi:10.1145/3417758. 2

[SY09] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: a survey of recent results and open questions. *Found. Trends Theor. Comput. Sci.*, 5(3-4):207–388 (2010), 2009. doi:10.1561/0400000039. 7, 8

[Val79] L. G. Valiant. Completeness classes in algebra. In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing (Atlanta, Ga., 1979)*, pages 249–261. ACM, New York, 1979. 1, 3, 10

[Val80] L. G. Valiant. Negation can be exponentially powerful. *Theoret. Comput. Sci.*, 12(3):303–314, 1980. doi:10.1016/0304-3975(80)90060-2. 2

[Val82] L. G. Valiant. Reducibility by algebraic projections. *Enseign. Math. (2)*, 28(3-4):253–268, 1982. iv, 34

[VSBR83] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Comput.*, 12(4):641–644, 1983. doi:10.1137/0212043. 11, 13

[Yeh19] Amir Yehudayoff. Separating monotone VP and VNP. In *STOC'19—Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 425–429. ACM, New York, 2019. 2