Branching Programs with Extended Memory: New Insights

Suryajith Chillara and Nithish Raja

Center for Security, Theory and Algorithmic Research (CSTAR),
International Institute of Information Technology - Hyderabad (IIIT-H), India.
{suryajith.chillara@, nithish.raja@research.}iiit.ac.in

Abstract. One of the central themes of *Algebraic Complexity Theory* is to understand the relative computation power of algebraic complexity classes VP and VNP. VP is a class of polynomial families which can be *computed efficiently* by algebraic circuits. VNP is a class of polynomials which can be *efficiently expressed* through polynomial families in VP, but we do not know if they can be computed efficiently. It is a long standing open problem in this area to show that VP is a strict subset of VNP. At this juncture, it is fair to believe that newer characterization of these complexity classes could help us understand these models better (and possibly help us is resolving this question in restricted settings like non-commutativity at the very least).

Recently, Mengel [MFCS, 2013] showed that Algebraic Branching Programs (ABPs)[1] can be extended with memory and the computational models thus obtained can be used to characterize VP and VNP. In particular, Mengel showed that ABPs with a single stack characterize VP, and branching programs with random-access memory characterize VNP.

In this work we show that algebraic branching programs with just 2 stacks efficiently simulates the polynomial families in VNP, and two stack branching programs are VNP-complete.

Further, we observe that k-stack branching programs (for all polynomially bounded k) are no more powerful than 2-stack branching programs (upto a polynomial blowup in size).

This strengthens the work of Mengel in the following way – VBP vs. VP vs. VNP are characterized by algebraic branching programs with 0, 1, and 2 stacks, respectively, or no-memory, a stack as memory, and a queue as memory respectively, which hold even in the non-commutative setting.

We also refine Mengel's characterization of VP through stack-height characterization.

[1] It is well known that algebraic branching programs (ABPs) characterize the class VBP (or equivalently $\mathsf{VP}_{\text{skew}}$).

1 Introduction

Algebraic models of computation An algebraic circuit is the most commonly encountered computational model in algebraic complexity. It is a DAG where the leaf nodes are labelled with variables or constants and the intermediate nodes are labelled with operations such as $+, \times$. Another important computational model is the algebraic branching program (ABP). ABPs are source-to-sink DAGs with the edges labelled with variables or constants. The polynomial computed by an ABP is given by sum of weights of all source-to-sink path where weight of a path is defined as product of weight of edges in it (refer to [7], [6] for a detailed discussion on these computational models).

VP, VBP, VNP, *reductions and completeness* The class VP contains all polynomial families that have polynomially-bounded degree, and are computed by a polynomial-sized circuits. Similarly, the class VBP contains those polynomial families in VP which can also be computed by a polynomial-sized ABPs. Another important class in algebraic complexity is VNP. It contains all polynomial families that can be expressed as a boolean sum of a VP polynomial family. That is, $(p_n)_{n\in\mathbb{N}} \in \mathsf{VNP}$ if $p_n(\bar{x}) = \sum_{\bar{e}\in\{0,1\}^{m-n}} q_m(\bar{x},\bar{e})$ where $m \in \text{poly}(n)$ and $(g_m)_{m\in\mathbb{N}} \in \mathsf{VP}$.

Definition 1 (p-projection). *Let $p_n(x_1,\ldots,x_n)$ and $q_m(y_1,\ldots,y_m)$ be two polynomials. Then p_n is said to be a p-projection of q_n if there exists a simple map $\sigma : \{x_i\}_{i\in[n]} \to \{y_i\}_{i\in[m]} \cup \mathbb{F}$ and $m \in \text{poly}(n)$ s.t. $p_n(x_1,\ldots,x_n) = q_m(\sigma(x_1),\ldots,\sigma(x_n))$.*

Valiant [8] showed that polynomial families (Perm_n) and (HC_n) are complete for VNP under p-projections. In other words, a polynomial family is in VNP if it can be expressed as a p-projection of either Permanent or Hamiltonian cycle familyes. Valiant further showed that a polynomial family is in VNP if it's coefficients can be computed in $\#\mathsf{P}/\text{poly}$ (Valiant's criterion). We refer the readers to [4,8,6] for finer details.

1.1 Branching programs with auxiliary memory

Let S be a stack, and Σ be its alphabet. For each letter $s \in \Sigma$, we can define stack operations $\texttt{push}(s)$ and $\texttt{pop}(s)$. We use $\texttt{nop}$ to denote no operation on the stack. A sequence of operations on the stack S is a sequence $\texttt{op}_1, \texttt{op}_2, \ldots, \texttt{op}_r$ where (for all $i \in [r]$) either $\texttt{op}_i$ is of the form $\texttt{push}(s)$ or $\texttt{pop}(s)$, for some letter $s \in \Sigma$, or of the form $\texttt{nop}$. Realizable sequences of stack operations are defined inductively.

1. Empty sequence is realizable.
2. A realizable sequence starts with an empty stack and ends with an empty stack.
3. If P is a sequence of realizable stack operations, then for all $s \in \Sigma$, $\texttt{push}(s) \cdot P \cdot \texttt{pop}(s)$ is also a realizable sequence. Further, $\texttt{nop} \cdot P$ and $P \cdot \texttt{nop}$ are also realizable sequences.
4. If P and Q are two realizable sequences of stack operations, then $P \cdot Q$ is also a realizable sequence of stack operations.

Definition 2 (Definition 3.1, [5]). *Let S be a stack over the alphabet Σ. A stack branching program (SBP) G is an algebraic branching program with an additional edge labeling σ where $\sigma : E \mapsto \{\texttt{op}(s) \mid \texttt{op} \in \{\texttt{push},\texttt{pop}\}, s \in \Sigma\} \cup \{\texttt{nop}\}$. A source-to-sink path $P = e_1, e_2, \ldots, e_r$ has the sequence of stack operations $\sigma(P) = \sigma(e_1), \sigma(e_2), \ldots, \sigma(e_r)$. We call P a stack realizable path if $\sigma(P)$ is a stack realizable sequence. Polynomial $f(x_1,\ldots,x_n)$ thus computed is given by the sum of weights of all stack realizable source-to-sink paths.*

$$f(x_1,\ldots,x_n) = \sum_{P:P\in s\rightsquigarrow t \text{ stack realizable paths}} \text{wt}(P).$$

Using this model of computation, Mengel proved the following theorem.

Theorem 1 (Theorem 3.3, [5]). *A polynomial family $\{f_n\}_{n\in\mathbb{N}}$ is in* VP *if and only if a family of polynomial-sized stack branching programs computes it.*

We make this theorem of [5] a bit more precise by adapting the proof of depth reduction of [10] and show the following.

Theorem 2. *A polynomial family $\{f_n\}_{n\in\mathbb{N}}$ is in* VP *if and only if a family of polynomial-sized stack branching programs of stack-height at most $O(\log^2 n)$ computes it.*

This additional characterization (along with Lemma 2) gives us an alternative proof to the fact that polynomial families in VP can be simulated by quasi-polynomial sized algebraic branching programs.

k-stack branching programs In this work we also introduce k-stack branching programs, a generalization of stack branching programs obtained by allowing the number of stacks to be a parameter.

Definition 3. *Let $S_1, \ldots, S_k$ be k many stacks over the alphabet Σ. A k-stack branching program G is an algebraic branching program with an additional edge labeling σ where*

$$\sigma : E \mapsto \{(op_1, \ldots, op_k) \mid \forall\, j \in [k], op_j \in \{\mathtt{push}_j(s_j), \mathtt{pop}_j(s'_j), \mathtt{nop}\} \text{ for } s_j, s'_j \in \Sigma\}$$

and, for each $j \in [k]$, $\mathtt{push}_j$ and $\mathtt{pop}_j$ are push *and* pop *operations on stack S_j, respectively. For each $i \in [k]$, let π_i be the projection of a k-tuple to its ith element. A source-to-sink path $P = e_1 e_2 \ldots e_r$ has the sequence of stack operations $\sigma(P) = \sigma(e_1), \sigma(e_2), \ldots, \sigma(e_r)$. For each $i \in [k]$, let $\sigma_i(P) = \pi_i(\sigma(e_1)), \pi_i(\sigma(e_2)), \ldots, \pi_i(\sigma(e_r))$.*

P is a k-stack realizable path if for each $i \in [k]$, $\sigma_i(P)$ is a stack realizable sequence. Polynomial $f(x_1, \ldots, x_n)$ thus computed is given by the sum of weights of all k-stack-realizable $s \rightsquigarrow t$ paths.

$$f(x_1, \ldots, x_n) = \sum_{\pi : \pi \in s \rightsquigarrow t \ \text{k-stack-realizable paths}} \mathrm{wt}(\pi).$$

Using this definition, we prove that k-stack branching programs are VNP-complete for $2 \leqslant k \leqslant \mathrm{poly}(n)$.

Theorem 3. *Let n, k be natural numbers such that n is large, and $2 \leqslant k \leqslant \mathrm{poly}(n)$. A polynomial family $\{f_n\}_{n\in\mathbb{N}} \in$ VNP if and only it can be computed by a family of polynomial-sized k-stack branching programs.*

A generic approach to prove Theorem 3 is the following.

1. Show that Hamiltonian cycle polynomial family (or Permanent polynomial family resp.) can be computed by polynomial-sized 2-stack branching programs. This can be done via gadget constructions (see Lemma 4 (Lemma 3 resp.)). That is, for all $n \in \mathbb{N}$, there is a polynomial-sized 2-stack branching program that computes HC_n (Perm_n resp.), and
2. Show that for any natural number k that is polynomially-bounded, families of polynomials computed by k-stack branching programs are in VNP (see Lemma 1).

However for the sake of this article, we would like to replace the approach in Item 1 with the following (more) insightful theorem.

Theorem 4. *Given a k-stack branching program ($k \geqslant 1$) of size s computing the polynomial $q_n(x_1, \ldots, x_n)$, there exists a $(k+1)$-stack branching program of size $\mathrm{poly}(s, n)$ computing*

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

We also provide efficient constructions of 2-stack branching program computing the Perm_n and HC_n polynomials (see Appendix A and Appendix B).

By putting Lemma 1, Lemma 4 and Theorem 4 together, we also get alternate proofs for the following well-known statements.

1. Families of polynomials that are obtained through *Exponential sum* of families of polynomials from either VBP or VP, are in VNP, and
2. VNP is closed under exponential summation (cf. [2, Theorem 2.19]).

2 Improved Characterization of VP

It was shown by Mengel [5] that a polynomial p_n computed by a multiplicatively disjoint circuit (see [4] for a detailed discussion on multiplicatively disjoint circuits) of size s can also be computed by a stack branching program of size $\text{poly}(s)$ and any polynomial p_n computed by a stack branching program of size s' can also be computed by a circuit of size $\text{poly}(s')$. These two statements combined gives us a characterization of VP i.e., a polynomial family $(p_n)_{n\in\mathbb{N}}$ is in VP if and only if there exists a $\text{poly}(n)$-sized stack branching program computing it. In this section, we improve upon this characterization by showing that any degree d polynomial computed by a circuit of size s can also be computed by a stack branching program of size $\text{poly}(n, s, d)$ and stack height $O(\log(s)\log(d))$. Setting $s, d \in \text{poly}(n)$, we get that any polynomial family $(p_n)_{n\in\mathbb{N}}$ is in VP if and only if there exists a $\text{poly}(n)$-sized stack branching program computing it and the stack height of such a stack branching program is bound by $O(\log^2(n))$. This can be seen as a stack height reduction result for stack branching program analogous to the depth reduction result for circuits shown in [10].

Before we describe the technical details, we first borrow some notation from [10]. 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 4. *Let C be a circuit and v, w are gates in the circuit. $f(v)$, $f(w)$ denote the polynomial computed at gates v, w respectively and $d(v)$, $d(w)$ denote the degree of the polynomial computed at gates v, w respectively. Then, the function $f(v; w)$ is defined as shown below and $d(f(v; w)) = d(w) - d(v)$.*

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

Definition 5 (Frontier gates). *Let C be a circuit computing an arbitrary polynomial. For any value of a, the set V_a is defined as follows.*

$$V_a = \{t |\ t \text{ is a gate in } C,\ d(t) > a,\ t = t' \times t'',\ d(t') \leqslant a\}. \tag{1}$$

Theorem 5. *Let P_n be a degree d polynomial computed by a circuit C of size $|C|$. Then, there exists a $\text{poly}(|C|)$-sized stack branching program with stack height $O(\log|C|\log d)$ computing P_n.*

Proof. We will now adapt and build on the proof of [5, proposition 3.6] and thus we recommend the readers to look at it. Further, 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 stack branching program (see [5] for a detailed discussion on relaxed stack branching program) 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 at most $O(\log(d(w)))$, 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 stack branching program s.t. $\sum$ wt(length m_{vw} realizable s_{vw} to t_{vw} 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 the gate w individually and then the pair 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 as variables and constants can be added to the relaxed stack branching program. Any linear form $\sum_{j=1}^{n} c_j x_j$ can be computed as shown in Figure 1. Note that the stack alphabet used here will be $\{\langle w, i\rangle \mid \forall w \in C, i \in [n]\} \cup \{\langle w, v, i\rangle \mid \forall v, w \in C, i \in [n]\} \cup \{u_k, v_k, u_{k'}, v_{k'} \mid u_k, v_k, u_{k'}, v_{k'} \in C\}$ and the stack symbols used in Figure 1 will change to $\langle w, v, i\rangle$ if the linear form is computed by $f(v;w)$.

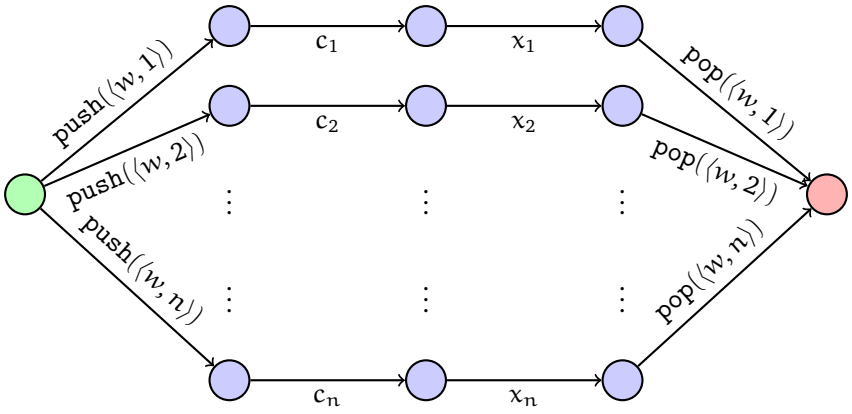


Fig. 1: Gadget computing linear form

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

Inductive hypothesis

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 stack branching program s.t. $\sum$ wt(length m_w realizable s_w to t_w 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 stack branching program s.t. $\sum$ wt(length m_{vw} realizable s_{vw} to t_{vw} walks) is equal to $f(v;w)$ 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 [10], 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 stack branching program exists with pairs of vertices $\{s_{t'}, t_{t'}\}$, $\{s_{t''}, t_{t''}\}$ and $\{s_{tw}, t_{tw}\}$ computing each $f(t')$, $f(t'')$ and $f(t;w)$ respectively.
   Next, recall from [5, Proposition 3.6] that we can construct a relaxed stack branching program with source and sink vertices s_w, t_w computing $f(w)$. For each product term we add 3 vertices, 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 $(3|C| + m_w + 2)|C|$. Since, we push one stack alphabet for addition and one for multiplication, the overall stack height increases by 2.
We know that $d(t) = d(t') + d(t'')$, $d(f(t;w)) = d(w) - d(t)$ and due to the inductive assumption, we get $m_{t'} \leqslant d(t') + 3^{(i+1)}$, $m_{t''} \leqslant d(t'') + 3^{(i+1)}$ and $m_{tw} \leqslant d(f(t;w)) + 3^{(i+1)}$.

$$\begin{aligned} m_w &\leqslant \max_t\{m_{t'} + m_{t''} + m_{tw}\} && (2)\\ &\leqslant \max_t\{d(t') + d(t'') + d(f(t;w)) + 3^{(i+2)}\} && (3)\\ &\leqslant \max_t\{d(t) + d(w) - d(t) + 3^{(i+2)}\} && (4)\\ &\leqslant \max_t\{d(w) + 3^{(i+2)}\} && (5)\\ &\leqslant d(w) + 3^{(i+2)}. && (6) \end{aligned}$$

2. Consider all pairs of gates v, w s.t. $d(f(v;w)) \in (2^i, 2^{i+1}]$. Due to [10], 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 stack branching program exists with pairs of vertices $\{s_{t''}, t_{t''}\}$, $\{s_{vt'}, t_{vt'}\}$ and $\{s_{tw}, t_{tw}\}$ computing $f(t'')$, $f(v;t')$ and $f(t;w)$ respectively.
Once again, by following the steps in [5, Proposition 3.6], we construct a relaxed stack branching program 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 at most $(3|C| + m_{vw} + 2)|C|^2$. Similar to the previous case, we push one stack alphabet for addition and one for multiplication. This causes the overall stack height to increase by 2.
We know that $d(f(v;t')) = d(t') - d(v)$, $d(f(t;w)) = d(w) - d(t)$ and due to the inductive assumption, we get $m_{t''} \leqslant d(t'') + 3^{(i+1)}$, $m_{vt'} \leqslant d(f(v;t')) + 3^{(i+1)}$ and $m_{tw} \leqslant d(f(t;w)) + 3^{(i+1)}$.

$$\begin{aligned} m_{vw} &\leqslant \max_t\{m_{t''} + m_{vt'} + m_{tw}\} && (7)\\ &\leqslant \max_t\{d(t'') + d(t') - d(v) + d(w) - d(t) + 3^{(i+2)}\} && (8)\\ &\leqslant \max_t\{d(w) - d(v) + 3^{(i+2)}\} && (9)\\ &\leqslant d(f(v;w)) + 3^{(i+2)}. && (10) \end{aligned}$$

The maximum walk length becomes $d + 3^{\lceil \log(d)\rceil + 1} \in O(d^2)$. Now, we can convert the relaxed stack branching program to a normal stack branching program as shown in [5, lemma 3.5] and the size of the stack branching program remain polynomial.

The overall stack height is atmost $2\log(d)$. However, every alphabet in the stack is from the set $\{\langle w, i\rangle \mid \forall w \in C, i \in [n]\} \cup \{\langle w, v, i\rangle \mid \forall v, w \in C, i \in [n]\} \cup \{u_k, v_k, u_{k'}, v_{k'} \mid u_k, v_k, u_{k'}, v_{k'} \in C\}$. Converting the stack alphabets to binary, the overall stack height becomes $O((\log|C|)(\log d))$.

□

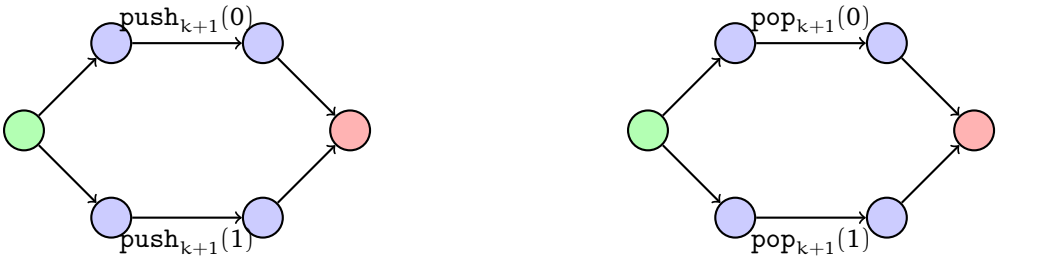
3 Computational power of k-stack branching programs

In this section, we first provide a proof for Theorem 4 and through this, we explore the computational power of k-stack branching programs.

Proof of Theorem 4. In the figures below (and henceforth in this document), for the sake brevity, edge weights, and stack operations on edges are only mentioned when they are different from 1, and when they are different from nop respectively. The default edge weight is 1, and the default operation is nop.

Let G_Q be the k-stack branching program that computes the polynomial $Q_{n+m}(x_1, \ldots, x_n, y_1, \ldots, y_m)$, and $|G_Q| = \text{poly}(n)$.

Let the graph gadgets $G_{\texttt{push}(0,1)}$, $G_{\texttt{pop}(0,1)}$, $G_{\texttt{init}}$ and G_{reset} be constructed as shown in Figures 2a, 2b, 3 and 4 respectively.



(a) Gadget $G_{\texttt{push}(0,1)}$ (b) Gadget $G_{\texttt{pop}(0,1)}$

Fig. 2: Gadget $G_{\texttt{push}(0,1)}$ and $G_{\texttt{pop}(0,1)}$

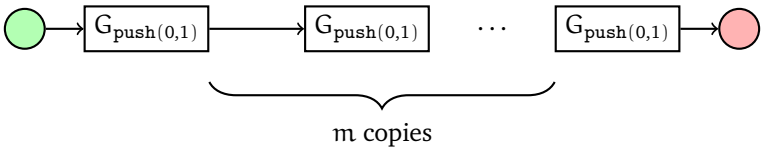


Fig. 3: Gadget G_{init}

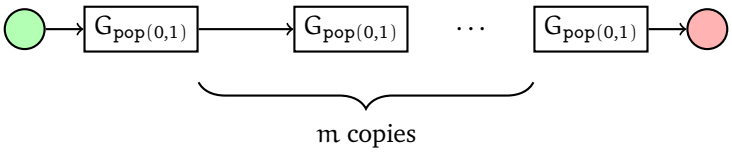


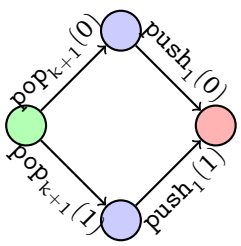
Fig. 4: Gadget $G_{\texttt{reset}}$

Let the helper gadgets $G_{k+1\to1}$, $G_{1\to k+1}$, $G_{(i)}$ and $G'_{(i)}$ be as described in Figures 5a, 5b, 6 and 7. Note that these gadgets make use of the $(k+1)$th stack.

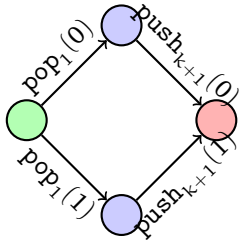
Any edge in G_Q with edge label y_i $(\forall\, i \in [m])$ is now replaced by a corresponding graph gadget H_i (shown in Figure 8).

We now construct a $(k+1)$-stack branching program G_P by putting together $G_{\texttt{init}}$, modified G_Q and $G_{\texttt{reset}}$ as shown in Figure 9.

Note that each source to sink path in the gadget $G_{\texttt{init}}$ fills m many $\{0,1\}$ values into the $(k+1)^{\text{th}}$ stack. We now associate i^{th} entry of this stack with the assignment of the variable y_i. The top of the stack corresponds to the assignment for y_1 and the bottom most element, to y_m. Thus, a path from source to sink in $G_{\texttt{init}}$ fixes the variables $y_1, \ldots, y_m$.



(a) Gadget $G_{k+1\to 1}$



(b) Gadget $G_{1\to k+1}$

Fig. 5: Gadgets $G_{k+1\to 1}$ and $G_{1\to k+1}$

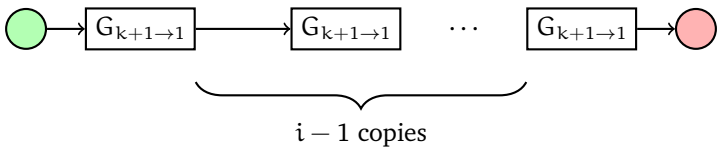


Fig. 6: Gadget $G_{(i)}$

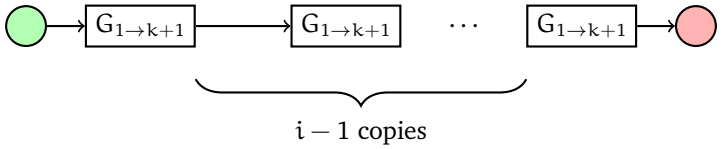


Fig. 7: Gadget $G'_{(i)}$

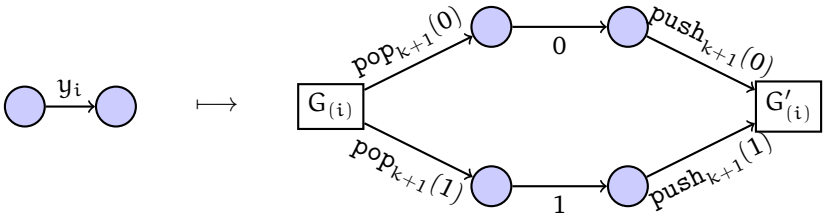


Fig. 8: Gadget H_i (for each $i \in [m]$)

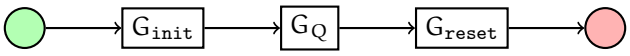


Fig. 9: $(k+1)$-stack branching program G_P

To access the assignment for a variable y_i, we need to access the i^{th} element from the top, in the $(k+1)$th stack. Towards that we use the helper gadgets $G_{(i)}$ and $G'_{(i)}$. $G_{(i)}$ moves the $i-1$ elements at the top of the $(k+1)^{\text{th}}$ stack 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 with weight 1 is added. Finally, $G'_{(i)}$ moves back all the elements moved by $G_{(i)}$ to $(k+1)^{\text{th}}$ stack.

It is easy to see that in any source to sink path of G_P, its subsection corresponding to $G_{\texttt{init}}$ determines the path taken in each instance of graph gadgets H_i ($i \in [m]$).

Putting it all together, we get that G_P computes the polynomial $P(x_1, \ldots, x_n)$.

$$\begin{aligned}
P(x_1, \ldots, x_n) &= \sum_{(e_1,\ldots,e_m)\in\{0,1\}^m} Q(x_1, \ldots, x_n, e_1, \ldots, e_m) \\
&= \sum_{\mathbf{e}=(e_1,\ldots,e_m)\in\{0,1\}^m} \left(\sum_{\pi:\pi\in s\rightsquigarrow t \text{ k-stack-realizable paths in } G_Q} \mathrm{wt}(\pi|_{\mathbf{e}}) \right) \\
&= \sum_{\pi:\pi\in s\rightsquigarrow t \text{ k-stack-realizable paths in } G_Q} \left(\sum_{\mathbf{e}=(e_1,\ldots,e_m)\in\{0,1\}^m} \mathrm{wt}(\pi|_{\mathbf{e}}) \right) \\
&= \sum_{\pi':\pi'\in s\rightsquigarrow t \ (k+1)\text{-stack-realizable paths in } G_P} \mathrm{wt}(\pi') \,.
\end{aligned}$$

In the above math block, the last equality follows from the fact that each k-stack realizable path is associated with each of the 2^m many paths of $G_{\mathtt{init}}$ which in turn correspond to the assignment $\mathbf{y} \leftarrow \mathbf{e}$, and $\mathrm{wt}(\pi|_{\mathbf{e}})$ corresponds to the weight of the path π after substituting for $y_1, \ldots, y_m$ by $e_1, \ldots, e_m$. □

Corollary 1. *Every polynomial family in* VNP *is computed by a polynomial-sized* 2*-stack branching programs.*

Proof. By definition, we know that every polynomial family $(p_n)_{n\in\mathbb{N}}$ in VNP can be expressed as $\sum_{\bar{e}\in\{0,1\}^{m-n}} g_m(\bar{x}, \bar{e})$ where the polynomial family $(g_m)_{m\in\mathbb{N}}$ is in VP and $m = \mathrm{poly}(n)$. Due to Theorem 5, we know that every polynomial family in VP has an efficient stack branching program computing it. Combining it with the above theorem, we see that every polynomial family in VNP has a 2-stack branching program computing it. □

The above corollary tells us that every polynomial family in VNP can be computed efficiently by 2-stack branching programs. However, is every polynomial family computed by a 2-stack branching program inside VNP? We give an affirmative answer to this question using the following lemma. In fact, the lemma proves a much stronger result. We prove that statement for k-stack branching program for every $k \in [1, \mathrm{poly}(n)]$.

Lemma 1. *Let* $(p_n)_{n\in\mathbb{N}}$ *be a polynomial family such that for each* n*,* p_n *is computed by a polynomial-sized* k*-stack branching program where* $0 \leqslant k \leqslant \mathrm{poly}(n)$*. Then* $(p_n)_{n\in\mathbb{N}}$ *is in* VNP*.*

Proof. Note that each k-stack realizable, source-to-sink path computes a monomial. Multiple such paths may compute the same monomial and together they would contribute to the coefficient of the monomial. Given a monomial, we can compute all the 2-stack realizable paths contributing to this in $\#\mathsf{P}/\mathrm{poly}$ – over a sequence of all non-deterministic choices at each node to go from the source to sink, we can accumulate the edge weights (including the stack operations) of all the edges along the traversal and check if – the product of weights of edges equals the given monomial, and if the sequence of stack operations are 2-stack realizable. Putting this together with Valiant's criterion completes the proof. □

Combining Theorem 4 with Lemma 1, we get the promised characterization 2-stack branching program of VNP. However, notice that we have proven Theorem 4 and Lemma 1 for an arbitrary k. Consider the following scenario, a polynomial p_n, in VNP, is computed by a 2-stack branching program and the boolean sum of p_n is computed by a 3-stack branching program (given by Theorem 4). However, Lemma 1 tells us that any polynomial computed by 3-stack branching program is also inside VNP. Therefore, we have also given a stack branching program based proof showing the closure of VNP under boolean sum (this was originally proven by Valiant [9, Theorem 5]). Another detail to note from the proof of Theorem 4 is that it can be modified slightly and used when p_n is computed by an ABP instead. To be more precise, using the proof of Theorem 4 with 2 additional stacks instead of 1, we see that if a polynomial is computed by a size s ABP, then its boolean sum is computed by 2-stack branching program i.e., the boolean sum of any polynomial computed by an efficient ABP lies inside VNP. This gives an alternate proof for boolean sum of polynomial families in VBP lying inside VNP (originally obtained as a consequence of [9, proposition 1] and [9, proposition 2]).

4 Relevance of Stack Height

Mengel proves that any 1-stack branching program with a larger alphabet size ($|\Sigma| > 2$) can be simulated by a 1-stack branching program (over binary alphabet) with poly-logarithmic blow-up in length and polynomial blow-up in size. Thus, it is sufficient to work with binary alphabet. This observation further implies that stack branching programs with constant-sized alphabet, logarithmic stack-height, and of size s can be simulated by algebraic branching programs of size $\mathrm{poly}(s)$. We extend this line of thought and state the following.

Lemma 2. *Let $P(x_1, \ldots, x_n)$ be a polynomial computed by a k-stack branching program (over binary alphabet) of size s, and stack-height at most H (for every stack). Then $P(x_1, \ldots, x_n)$ can also be computed by an algebraic branching program of size at most $O(2^{k(H+1)} \cdot s)$.*

Proof. Let G be the k-stack branching program. Let V and E be the vertex and edge sets of G. Let $\mathscr{S}$ be the set of all possible heights of stack configurations.

$$\mathscr{S} = \{(h_1, \ldots, h_k) \mid \forall\, i \in [k], 0 \leqslant h_i \leqslant H\}.$$

For a stack-height vector $(h_1, \ldots, h_k)$ in $\mathscr{S}$, there are $\prod_{i=1}^{k} 2^{h_i}$ many different stack configurations. In total, there are at most

$$\sum_{0 \leqslant h_1, \ldots, h_k \leqslant H} \left(\prod_{i=1}^{k} 2^{h_i} \right) \leqslant \left(\sum_{j=0}^{H} 2^j \right)^k \leqslant 2^{(H+1)k}.$$

Let V' and E' be initialized to empty sets. Let us use t to denote the quantity $2^{(H+1)k}$. For each vertex $v \in V$, we make t many copies of this vertex and add them to set V'. v_C be the copy of v that corresponds to the configuration C of k-stacks. For each edge $e = (u, v) \in E$ with weight $\mathrm{wt}(e)$ and the associated stack operations $(\mathsf{op}_1, \ldots, \mathsf{op}_k)$, we add an edge between u_{C_1} and v_{C_2} (with edge weight $\mathrm{wt}(e)$), for all stack configurations C_1 and C_2 such that C_2 is obtained from C_1 through the stack operations $(\mathsf{op}_1, \ldots, \mathsf{op}_k)$. Let G' be the algebraic branching program obtained at the end of the above process with V' and E' as its vertex and edge sets.

It is now straightforward to form a bijection between realizable source-to-sink paths in the G and source-to-sink paths in G'. □

Observe that we can trade-off H for k and vice versa and get the following statements. The set of families of polynomials computed by polynomial-sized k-stack branching programs with $k = O(1)$ and H logarithmic, is equivalent to VBP.

5 Discussion

5.1 What about VF

In this paper we mainly talk about branching programs with memory and their characterizations of VP and VNP in general and non-commutative settings. So far we did not make any observations about the formulas. Ben-Or and Cleve [1] showed that algebraic formulas are computationally equivalent to width-3 branching programs. Mengel [5, Lemma 3.9] showed that any polynomial family computed by polynomial-sized algebraic circuits can be computed with a family of width-2 stack branching programs over the binary alphabet. By analysing [5, Lemma 3.9] carefully, we can get that any polynomial family that is computed by polynomial-sized algebraic branching programs can also be computed by polynomial-sized width-2 stack branching programs over the binary alphabet, with stack-height at most logarithmic. In other words, adding a logarithmically height-bounded stack to a constant-width algebraic branching program adds computational power (assuming VF $\neq$ VBP) and this is in contrast to the fact that general 1-stack branching programs with logarithmic stack-height are computationally equivalent to algebraic branching programs.

5.2 Future directions

- In this work, we showed that boolean summation of a polynomial can be computed using exactly one additional stack Theorem 4. To be precise, given a k-stack branching program that efficiently computes polynomial $q(\bar{x})$, we can construct a $(k+1)$-stack branching program that computes the polynomial $p = \sum_{e \in \{0,1\}^{|x|}} q(\bar{x}, \bar{e})$. The next step would be to perform the reverse. Given a k-stack branching program computing the polynomial $p(\bar{x})$ and the guarantee that polynomial p is of the form $p = \sum_{e \in \{0,1\}^{|x|}} q(\bar{x}, \bar{e})$, can we construct a $(k-1)$-stack branching program or k-stack branching program that computes $q(\bar{x})$?
- Due to [5], we know that VP is characterized by stack branching programs. Does this hold in the monotone setting as well? We answer this for mVNP and 2-stack branching programs by showing that the permanent polynomial family, which is known to be not in mVNP, can be computed efficiently by a monotone 2-stack branching program Lemma 3.
- Are k-stack branching programs strictly more powerful than monotone k-stack branching programs? In all the well known computational models, the only way to perform cancellations is via negation. However, k-stack branching programs have an alternate way of performing negations (by making the corresponding paths non-realizable). Therefore, it is not immediately clear if negations grant any additional computational power. This is not known even with the restriction of $k = 1$.
- Since monotone 2-stack branching programs are able to compute polynomial families that are outside mVNP, it would be interesting to figure out the exact computational power of monotone 2-stack branching programs. Do they characterize any of the known classes beyond monotone VNP (see [3] for details on classes beyond mVNP).
- Can we use the tools from weighted pushdown automata theory to answer the VP vs VNP question in the non-commutative setting.

References

1. Ben-Or, M., Cleve, R.: Computing algebraic formulas using a constant number of registers. SIAM J. Comput. **21**(1), 54–58 (1992). https://doi.org/10.1137/0221006, https://doi.org/10.1137/0221006 10
2. Bürgisser, P.: Completeness and Reduction in Algebraic Complexity Theory, Algorithms and computation in mathematics, vol. 7. Springer (2000) 4
3. Chatterjee, P., Gajjar, K., Tengse, A.: Monotone Classes Beyond VNP. In: 43rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 284. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.FSTTCS.2023.11, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSTTCS.2023.11 11
4. Malod, G., Portier, N.: Characterizing valiant's algebraic complexity classes. J. Complex. **24**(1), 16–38 (2008). https://doi.org/10.1016/j.jco.2006.09.006, https://doi.org/10.1016/j.jco.2006.09.006 2, 4
5. Mengel, S.: Arithmetic branching programs with memory. In: Mathematical foundations of computer science 2013, Lecture Notes in Comput. Sci., vol. 8087, pp. 667–678. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40313-2_59, https://doi.org/10.1007/978-3-642-40313-2_59 2, 3, 4, 5, 6, 10, 11, 19
6. Saptharishi, R.: A survey of lower bounds in arithmetic circuit complexity, version 9.0.3 (2021), https://github.com/dasarpmar/lowerbounds-survey/releases/tag/v9.0.3, github survey 2
7. Shpilka, A., Yehudayoff, A.: Arithmetic circuits: a survey of recent results and open questions. Found. Trends Theor. Comput. Sci. **5**(3-4), 207–388 (2010) (2009). https://doi.org/10.1561/0400000039, https://doi.org/10.1561/0400000039 2
8. Valiant, L.G.: Completeness classes in algebra. In: Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing (Atlanta, Ga., 1979), pp. 249–261. ACM, New York (1979) 2
9. Valiant, L.G.: Reducibility by algebraic projections. Enseign. Math. (2) **28**(3-4), 253–268 (1982) 9
10. Valiant, L.G., Skyum, S., Berkowitz, S., Rackoff, C.: Fast parallel computation of polynomials using few processors. SIAM J. Comput. **12**(4), 641–644 (1983). https://doi.org/10.1137/0212043, https://doi.org/10.1137/0212043 3, 4, 5, 6

A 2-Stack Branching Program Computing Permanent Polynomial

Lemma 3. *For all $n \in \mathbb{N}$, there exists a $O(n^3)$-sized 2-stack branching program that computes* 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, S_2 denote the two stacks and the stack alphabet is given by $T = \{\langle b, i\rangle \mid b \in \{0,1\}, i \in [n]\}$. Perm_n is defined over variables $\{x_{ij} \mid i \in [n], j \in [n]\}$.

Before we present the construction, let us try to see how to compute a single monomial, that corresponds to a permutation, using the two stacks and the stack alphabet. We start by having only the alphabets $\{\langle 0, i\rangle \mid i \in [n]\}$ in S_1 and S_2 being empty. When a variable x_i is multiplied to the monomial, we remove $\langle 0, i\rangle$ from S_1 and replace it with $\langle 1, i\rangle$. Notice that $\langle 0, i\rangle$ is never added back to S_1 again. Therefore, if x_i is multiplied to the monomial a second time, we try to remove $\langle 0, i\rangle$ from the stack and encounter a fault. After n such multiplications to 1, we end up with a monomial corresponding to a permutation. An important detail is that most times, $\langle 0, i\rangle$ may not be at the top of S_1. This is where we make use of the second stack. We pop elements from S_1 and push them into S_2 until the required element is at the top and when the operation is done, we pop all the elements from S_2 and push them back into S_1.

To generalize this technique to compute Perm_n over variables $\{x_{ij} \mid i, j \in [n]\}$, we first observe that directly using the above technique with the stack alphabets $\{\langle 0, j\rangle \mid j \in [n]\}$ already ensures that each monomial contains exactly one variable from each column. If the construction ensures that the first variable is from the first row, second variable is from the second row and so on, then the monomial computed will be a monomial in Perm_n.

The construction is given below. We start with gadgets G' (Figure 10) and G'' (Figure 11). These gadgets are used to push the stack alphabets $\{\langle 0, j\rangle \mid j \in [n]\}$ into S_1 and pop the stack alphabets $\{\langle 1, j\rangle \mid j \in [n]\}$ from S_1.

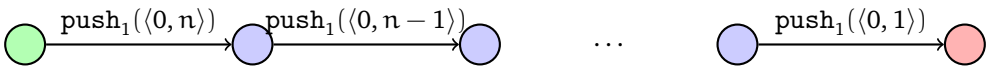


Fig. 10: Gadget G'

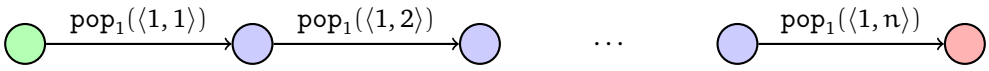


Fig. 11: Gadget G''

The gadget $G_{\mathtt{pop}(i)}$ (Figure 13), constructed by connecting gadgets $G_{(1)}$, $G_{(2)}$, ..., $G_{(i-1)}$ in series, is used to move the top $i-1$ elements from S_1 to S_2 and bringing $\langle b, i\rangle$ ($b \in \{0,1\}$) to the top of S_1. The gadget $G_{\mathtt{push}(i)}$ (Figure 15), constructed by connecting gadgets $G_{(i-1)}$, $G_{(i-2)}$, ..., $G_{(1)}$ in series, is used to move the elements from S_2 back to S_1.

For each $i \in [n]$, let the source-to-sink graph $G_{\mathtt{map}(i)}$ be constructed as shown in Figure 16. That is, the source and sink vertices of $G_{\mathtt{map}(i)}$ 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 $G_{\mathtt{pop}(j)}$, a single vertex (say $v_{\mathtt{map}(i),j}$) and graph gadget $G_{\mathtt{push}(j)}$ in series such that all new edges except for the edge between $G_{\mathtt{pop}(j)}$ and $v_{\mathtt{map}(i),j}$, and the edge between $v_{\mathtt{map}(i),j}$ and $G_{\mathtt{push}(j)}$ have weight 1 and no associated stack operations. The edge between $G_{\mathtt{pop}(j)}$ and $v_{\mathtt{map}(i),j}$ has weight $x_{i,j}$ and the associated stack operation $\mathtt{pop}_1(\langle 0, j\rangle)$, and the edge between $v_{\mathtt{map}(i),j}$ and $G_{\mathtt{push}(j)}$ has weight 1 and the associated stack operation $\mathtt{push}_1(\langle 1, j\rangle)$. An important detail to note is that the construction of $G_{\mathtt{map}(i)}$ ensures that only variables from the i^{th} row get multiplied to the monomial.

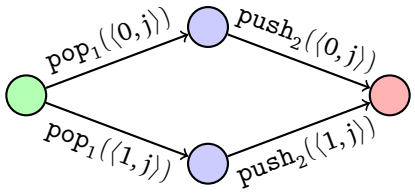


Fig. 12: Gadget $G_{(j)}$

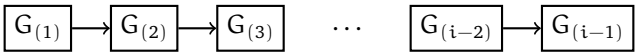


Fig. 13: Gadget $G_{\mathtt{pop}(i)}$

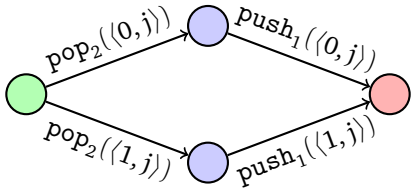


Fig. 14: Gadget $G'_{(j)}$

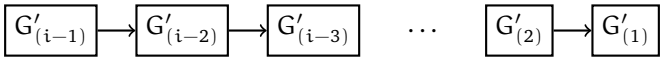


Fig. 15: Gadget $G_{\mathtt{push}(i)}$

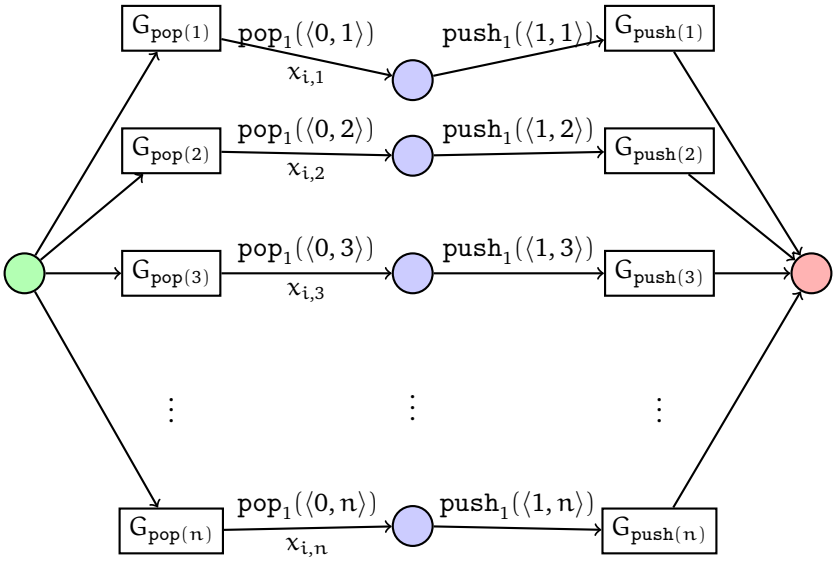


Fig. 16: Gadget $G_{\mathtt{map}(i)}$

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', $G_{\mathtt{map}(1)}$, …, $G_{\mathtt{map}(n)}$, G'' in series (with connecting edges of weight 1 and no associated stack operations) as shown in Figure 17.

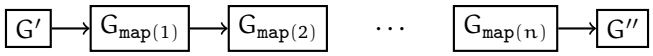


Fig. 17: Gadget connections to obtain a 2-stack branching program G

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 $G_{\mathtt{map}(i)}$ 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 $G_{\mathtt{map}(i)}$ and $G_{\mathtt{map}(i')}$ 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 $G_{\mathtt{map}(i)}$ and $G_{\mathtt{map}(i')}$ respectively.

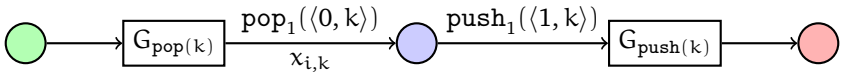


Fig. 18: Path P_i where i maps to k in $G_{\mathtt{map}(i)}$

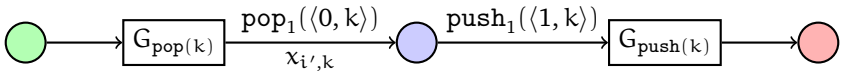


Fig. 19: Path $P_{i'}$ where i' maps to k in $G_{\mathtt{map}(i')}$

Note that the operations $\mathtt{pop}_1(\langle 0, k\rangle), \mathtt{push}_1(\langle 1, k\rangle)$ in $G_{\mathtt{map}(i)}$ precede the operations $\mathtt{pop}_1(\langle 0, k\rangle), \mathtt{push}_1(\langle 1, k\rangle)$ in $G_{\mathtt{map}(i')}$, in the sequence of stack operations associated with path P. As i is getting mapped to k in $G_{\mathtt{map}(i)}$, $\langle 0, k\rangle$ is popped from the first stack and $\langle 1, k\rangle$ is pushed into it (along with other operations, cf. Figure 12 and Figure 13). 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 $G_{\mathtt{map}(1)}$ till $G_{\mathtt{map}(n)}$ 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 $G_{\mathtt{map}(i)}$ 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 20.

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.

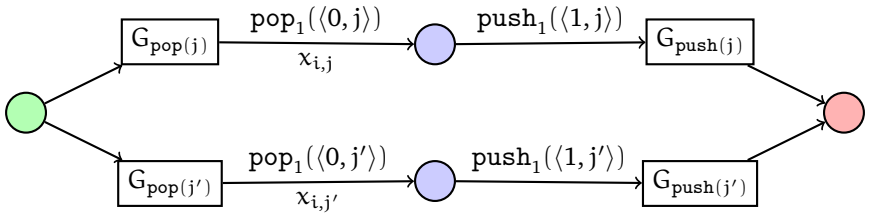


Fig. 20: Alternate paths

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 $G_{\mathtt{map}(i)}$ (for all $i \in [n]$), choose the $\pi(i)^{\text{th}}$ 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 $G_{\mathtt{map}(j)}$, we need to select the $\pi(j)^{\text{th}}$ path to get $x_{j,\pi(j)}$ and $\pi'(j)^{\text{th}}$ 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 $G_{\mathtt{map}(i)}$ (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.

□

B 2-stack Branching Program Computing Hamiltonian Cycle Polynomial

Definition 6 (Hamiltonian cycle polynomial). *Let* $X = \{x_{i,j} \mid 1 \leqslant i, j \leqslant n\}$. *For all* $n \in \mathbb{N}$, $\mathsf{HC}_n(X)$ *is defined as follows.*

$$\mathsf{HC}_n(X) = \sum_{\sigma \text{ is a cyclic permutation in } S_n} \prod_{i=1}^{n} x_{i,\sigma(i)}\,.$$

Lemma 4. *For all* $n \in \mathbb{N}$, *Hamiltonian cycle polynomial* $\mathsf{HC}_n(X)$ *can be computed by a* $O(n^3)$*-sized 2-stack branching program.*

Proof. Let S_1 and S_2 denote the two stacks. Let the stack alphabet be $\{\langle b, i\rangle \mid b \in \{0,1\}, i \in [n]\} \cup \{\langle i\rangle \mid i \in [n]\}$. For each stack S_u (for $u \in \{1,2\}$), let the push and pop operations be denoted by $\mathtt{push}_u(s)$ and $\mathtt{pop}_u(s)$ for some letter $s \in \Sigma$. As before, $\mathtt{nop}$ denotes no operation on the stacks.

The idea behind the construction is the following. We start with $\{\langle 0, i\rangle \mid i \in [n]\}$ in S_1 and S_2 being empty. A hamiltonian cycle of a graph with n vertices is n edges forming a cycle i.e., each variable x_{ij} represents the edge from vertex i to j. When the edge corresponding to x_{ij} is traversed, we mark the edge j as visited by replacing $\langle 0, j\rangle$ with $\langle 1, j\rangle$ in S_1. Next we push the alphabet $\langle j\rangle$ onto S_1. This is done to ensure that the next edge chosen will

be of the form $x_{jj'}$. After n such steps, we pop $\{\langle 1, i\rangle \mid i \in [n]\}$ from S_1. This ensures that every vertex was visited once and since only n vertices were visited, each vertex was visited exactly once. The role of S_2 is the following. When $\langle 0, j\rangle$ is replaced with $\langle 1, j\rangle$ in S_1, all the elements above $\langle 0, j\rangle$ are first popped from S_1 and pushed into S_2. Once the replacement is done, the elements are popped from S_2 and pushed back into S_1.

Let G' and G'' be directed path graphs as shown in Figures 21 and 22 respectively. Every edge has weight 1 in both of these graphs.

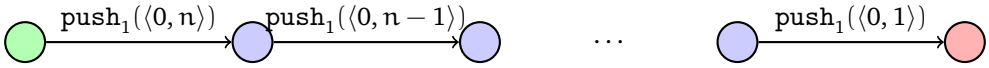


Fig. 21: Gadget G'

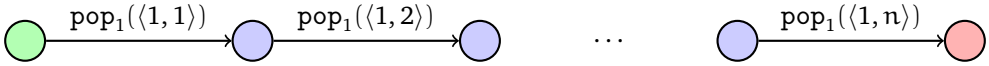


Fig. 22: Gadget G''

For each $j \in [n]$, let the source-to-sink graph $G_{(j)}$ be as shown in Figure 23. Every edge has weight 1 in this graph.

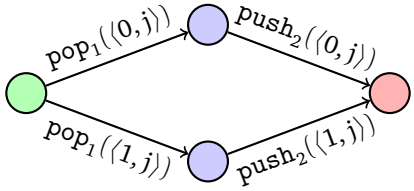


Fig. 23: Gadget $G_{(j)}$

For each $i \in [n]$, let the source-to-sink graph $G_{\mathtt{pop}(i)}$ be constructed by placing the graphs $G_{(1)}$, $G_{(2)}$, …, $G_{(i-1)}$ in series (see Figure 24). 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 $G_{\mathtt{pop}(i)}$ are the source of $G_{(1)}$, and the sink of $G_{(i-1)}$.

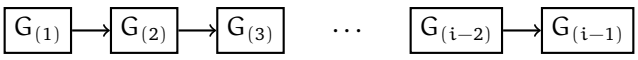


Fig. 24: Gadget $G_{\mathtt{pop}(i)}$

For each $j \in [n]$, let the source-to-sink graph $G'_{(j)}$ be as shown in Figure 25. Every edge has a weight 1 in this graph.

For each $i \in [n]$, let the source-to-sink graph $G_{\mathtt{push}(i)}$ be constructed by placing the graphs $G'_{(i-1)}, G'_{(i-2)}, \ldots, G'_{(1)}$ in series (see Figure 26). 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 $G_{\mathtt{push}(i)}$ are the source of $G'_{(i-1)}$, and the sink of $G'_{(1)}$.

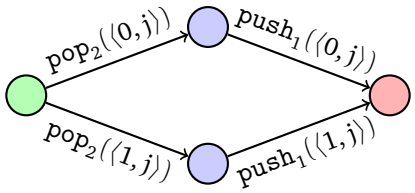


Fig. 25: Gadget $G'_{(j)}$

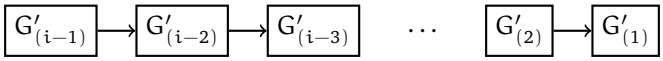


Fig. 26: Gadget $G_{\mathtt{push}(i)}$

We will now construct new graph gadgets $G_{\mathtt{out}(i)}$ (for each $i \in [n]$) as shown in Figure 27, and $G_{\mathtt{in}}$ as shown in Figure 28.

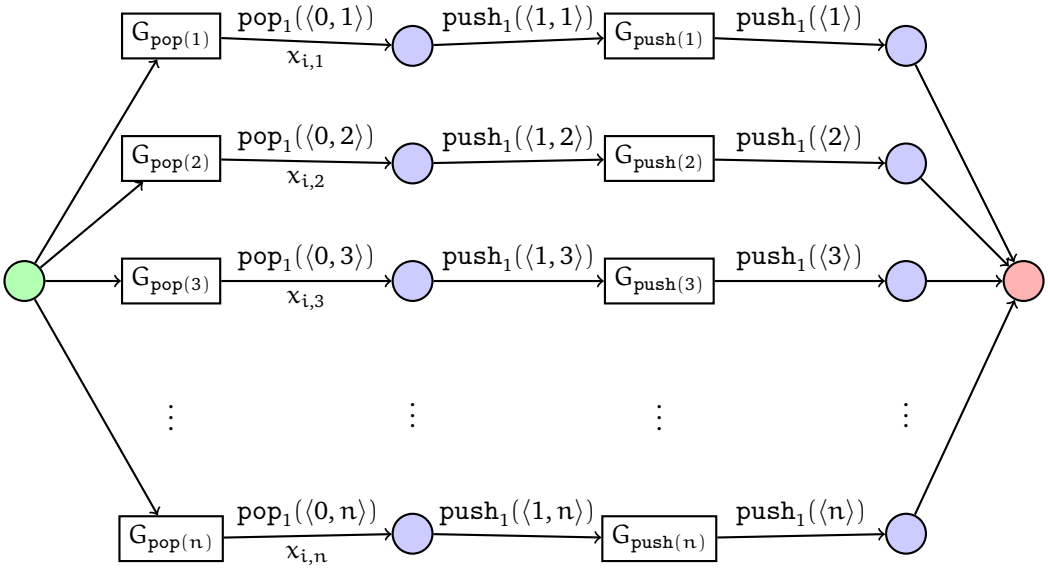


Fig. 27: Gadget $G_{\mathtt{out}(i)}$

We now get the 2-stack branching program G by placing the graph gadgets G', n many copies of $G_{\mathtt{in}}$, and G'' in series (with connecting edges of weight 1 and no associated stack operations) as shown in Figure 29.

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 $i_2, \dots, i_{n-1} \in [n] \setminus \{1\}$ 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 $G_{\mathtt{in}}$ in series. We use the notation $G^{(j)}_{\mathtt{in}}$ to refer to the j^{th} copy of $G_{\mathtt{in}}$ in series. For each $j \in [n]$ choose a source-to-sink path in the graph gadget $G^{(j)}_{\mathtt{in}}$ that goes through the edge with stack operation $\mathtt{pop}(\langle i_j\rangle)$ and in the graph gadget $G_{\mathtt{out}(i_j)}$ 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 $G^{(j)}_{\mathtt{in}}$ as P_j.

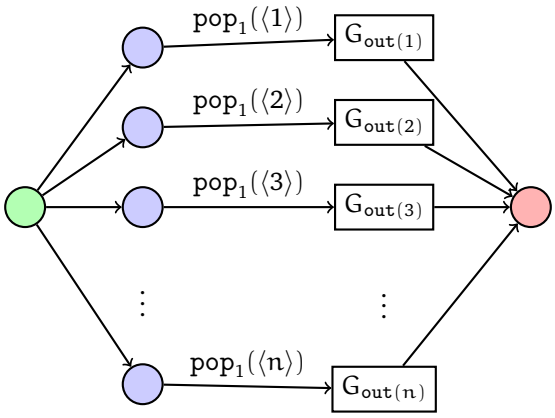


Fig. 28: Gadget $G_{\mathtt{in}}$

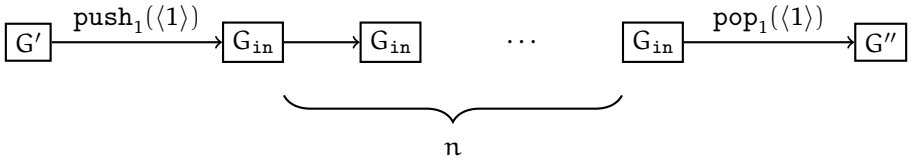


Fig. 29: Gadget connections for hamiltonian cycle

Note that at the end of each graph gadget $G_{\mathtt{in}}^{(j)}$, the letter $\langle \pi(i_j)\rangle$ is at the top of stack S_1. Thus, in the next gadget $G_{\mathtt{in}}^{(j+1)}$ 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 $G_{\mathtt{push}}$ and $G_{\mathtt{pop}}$. 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 $G_{\mathtt{in}}$ and $G_{\mathtt{out}}$ ensure that each monomial corresponds to a path. The number of $G_{\mathtt{in}}$ gadgets is n; therefore, the monomials correspond to paths of length n. The $\mathtt{push}_1(\langle 1\rangle)$ operation at the start and the $\mathtt{pop}_1(\langle 1\rangle)$ 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. □

C Queue Branching Programs

Queue branching programs: With queue the operation associated with it are $\mathtt{insert}(s)$ and $\mathtt{remove}(s)$. Let $P = \mathtt{op}_1, \mathtt{op}_2, \ldots, \mathtt{op}_r$ be a sequence of queue operations. Let countInsert$(P, \mathtt{insert}(s), i)$ denote the count of $\mathtt{insert}$ operations that occur before the i^{th} occurance of the $\mathtt{insert}(s)$ operation, and let countRemove$(P, \mathtt{remove}(s), i)$ denote the count of $\mathtt{remove}$ operations that occur before the i^{th} occurrence of the $\mathtt{remove}(s)$ operation. P is a queue realizable sequence if the following conditions hold.

- $\forall s \in \Sigma$, occ$(P, \mathtt{insert}(s)) =$ occ$(P, \mathtt{remove}(s))$, and
- $\forall s \in \Sigma$, countInsert$(P, \mathtt{insert}(s), i) =$ countRemove$(P, \mathtt{remove}(s), i)$.

Queue realizable sequences can be visualized as – starting with an empty queue, performing the operations in the sequence onto the queue, and finally ending with an empty queue.

Definition 7. *A queue branching program (QBP)* G *is an algebraic branching program with an additional edge labeling* σ *where*

$$\sigma : E \mapsto \{\mathtt{op}(s) \mid \mathtt{op} \in \{\mathtt{insert}, \mathtt{remove}\}, s \in \Sigma\} \cup \{\mathtt{nop}\}.$$

A source-to-sink path $P = e_1 e_2 \dots e_r$ *has the sequence of queue operations* $\sigma(P) = \sigma(e_1)\sigma(e_2)\dots\sigma(e_r)$*. The source-to-sink path is said to be queue realizable if the sequence of queue operations is realizable. Polynomial* $f(x_1, \dots, x_n)$ *computed by a queue branching program is given by the sum of weights of all queue realizable* $s \rightsquigarrow t$ *paths.*

$$f(x_1, \dots, x_n) = \sum_{P : P \in s \rightsquigarrow t \text{ queue realizable paths}} \mathrm{wt}(P).$$

Definition 8. *The class* QBP *consists of all polynomial families that can be computed by families of branching programs augmented with a queue.*

The study by Mengel [5] 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 5. *Any polynomial family computed by a polynomial-sized queue branching program is inside* VNP.

The proof of this lemma is along the same lines of the proof for Lemma 1.

Lemma 6. *For all* $n \in \mathbb{N}$*, there exists a* $O(n^3)$ *sized queue branching program 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 $\mathtt{insert}(s)$ and $\mathtt{remove}(s)$ for some letter $s \in T$. As before, $\mathtt{nop}$ 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 30 and Figure 31 respectively. Every edge has a weight 1 in both of these graphs.

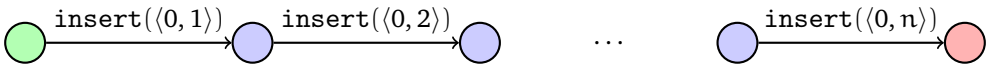


Fig. 30: Gadget G'

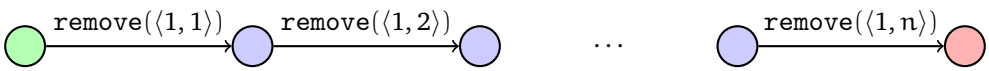


Fig. 31: Gadget G''

For each $j \in [n]$, the source-to-sink graph $G_{(j)}$ is as shown in Figure 32 with each edge having weight 1.

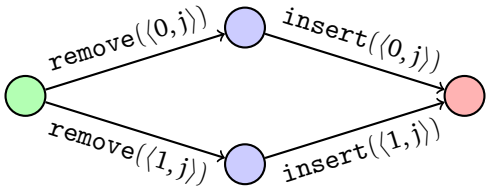


Fig. 32: Gadget $G_{(j)}$

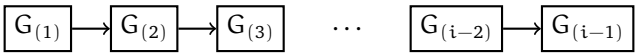


Fig. 33: Gadget $G_{\texttt{remove}(i)}$

For each $i \in [n]$, the gadget $G_{\texttt{remove}(i)}$ 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 of $G_{(i-1)}$ as shown in Figure 33. All edges used for these connections have weight 1 and queue operation `nop`.

For each $i \in [n]$, the gadget $G'_{\texttt{remove}(i)}$ 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 34. All edges used for these connections have weight 1 and queue operation `nop`.

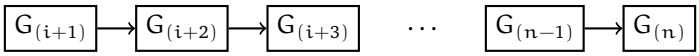


Fig. 34: Gadget $G'_{\texttt{remove}(i)}$

For each $i \in [n]$, let the source-to-sink graph $G_{\texttt{map}(i)}$ be constructed as shown in Figure 35. The source and sink vertices of $G_{\texttt{map}(i)}$ 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 $G_{\texttt{remove}(j)}$, a single vertex (say $v_{\texttt{map}(i),j}$), gadget $G'_{\texttt{remove}(j)}$ and sink vertex in series. All edges have weight 1 and queue operation `nop` except for the edge connecting sink of gadget $G_{\texttt{remove}(i)}$ to $v_{\texttt{map}(i),j}$ and the edge connecting $v_{\texttt{map}(i),j}$ to source of $G'_{\texttt{remove}(i)}$. The edge connecting sink of $G_{\texttt{remove}(i)}$ to $v_{\texttt{map}(i),j}$ has weight x_{ij} and queue operation $\texttt{remove}(\langle 0, j\rangle)$. The edge connecting $v_{\texttt{map}(i),j}$ to source of $G'_{\texttt{remove}(i)}$ has weight 1 and queue operation $\texttt{insert}(\langle 1, j\rangle)$.

We now claim that the queue branching program G computes the polynomial Perm_n by placing the gadgets G', $G_{\texttt{map}(1)}$, ..., $G_{\texttt{map}(n)}$, G'' in series with edges having weight 1 and queue operation `nop` as shown in Figure 36.

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 $G_{\texttt{map}(i)}$ is $x_{i,j}$. Therefore, any source-to-sink path in graph 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 $\texttt{remove}(\langle 0, k\rangle)$ queue operation in graph gadget $G_{\texttt{map}(i)}$ and the path with $\texttt{remove}(\langle 0, k\rangle)$ queue operation in graph gadget $G_{\texttt{map}(i')}$ 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

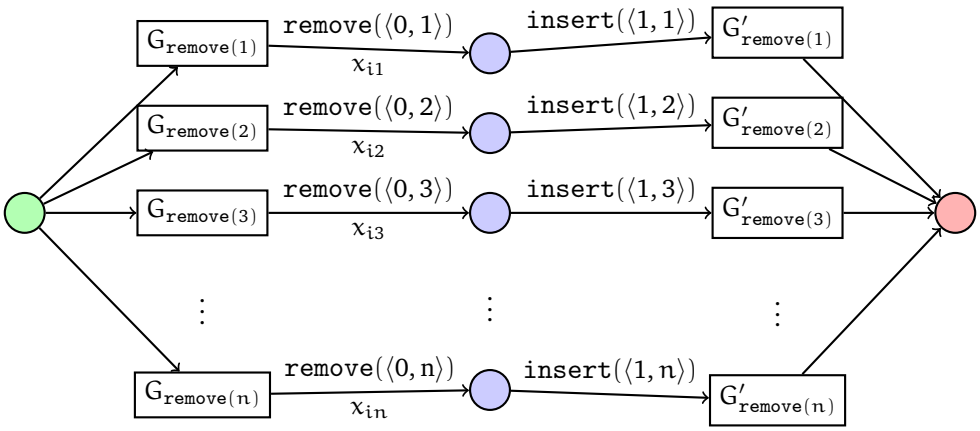


Fig. 35: Gadget $G_{\texttt{map}(i)}$

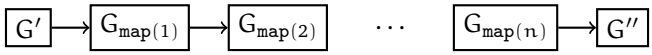


Fig. 36: Gadget connections for QBP computing permanent

path containing x_{ik} and the path containing $x_{ik'}$ in $G_{\texttt{map}(i)}$ 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 $G_{\texttt{map}(i)}$ 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 $G_{\texttt{map}(i)}$ 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 $G_{\texttt{map}(j)}$ to get $x_{j,\pi(j)}$ and the path $P_{j,\pi'}$ in $G_{\texttt{map}(j)}$ to get $x_{j,\pi'(j)}$. Since we select exactly one path in graph gadget $G_{\texttt{map}(i)}$ (for all $i \in [n]$), we cannot get the same queue realizable path for both permutations.

□

Theorem 6. *A polynomial family* (f_n) *is in* VNP *if and only if there exists a polynomial-sized queue branching program that computes it.*

Proof. The Lemma 5 shows that any polynomial computed by a polynomial-sized queue branching program is inside VNP. Lemma 6 shows that VNP-complete polynomials can be computed by polynomial-sized queue branching program. Finally, we notice that given a polynomial p_n and a queue branching program computing it, we can compute any p-projection of the polynomial by replacing every edge in the queue branching program that is labelled by a variable with a ABP computing the corresponding linear form. These statements together tell us that a VNP-complete polynomial can be computed by a queue branching program, queue branching programs can simulate p-projections and any polynomial family computed by efficient queue branching programs are inside VNP. Putting these points together, we get that a polynomial family is in VNP if and only if there exists an efficient queue branching program computing it. □