Trading Time and Space in Catalytic Branching Programs

James Cook
Independent Researcher[1], Toronto, Canada

Ian Mertz
University of Toronto, Canada

Abstract

An m*-catalytic branching program* (Girard, Koucký, McKenzie 2015) is a set of m distinct branching programs for f which are permitted to share internal (i.e. non-source non-sink) nodes. While originally introduced as a non-uniform analogue to catalytic space, this also gives a natural notion of amortized non-uniform space complexity for f, namely the smallest value $|G|/m$ for an m-catalytic branching program G for f (Potechin 2017).

Potechin (2017) showed that every function f has amortized size $O(n)$, witnessed by an m-catalytic branching program where $m = 2^{2^n-1}$. We recreate this result by defining a catalytic algorithm for evaluating polynomials using a large amount of space but $O(n)$ time. This allows us to balance this with previously known algorithms which are efficient with respect to space at the cost of time (Cook, Mertz 2020, 2021). We show that for any $\epsilon \geq 2n^{-1}$, every function f has an m-catalytic branching program of size $O_\epsilon(mn)$, where $m = 2^{2^{\epsilon n}}$. We similarly recreate an improved result due to Robere and Zuiddam (2021), and show that for $d \leq n$ and $\epsilon \geq 2d^{-1}$, the same result holds for $m = 2^{\binom{n}{\leq \epsilon d}}$ as long as f is a degree-d polynomial over $\mathbb{F}_2$. We also show that for certain classes of functions, m can be reduced to $2^{\text{poly}\, n}$ while still maintaining linear or quasi-linear amortized size.

In the other direction, we bound the necessary length, and by extension the amortized size, of any *permutation branching program* for an arbitrary function between $3n$ and $4n - 4$.

2012 ACM Subject Classification Theory of computation → Computational complexity and cryptography

Keywords and phrases complexity theory, branching programs, amortized, space complexity, catalytic computation

Digital Object Identifier 10.4230/LIPIcs.CCC.2022.8

Related Version *Full Version*: `https://eccc.weizmann.ac.il/report/2022/026/`

Funding *Ian Mertz*: Partially funded by NSERC.

Acknowledgements The authors would like to thank Toniann Pitassi, Robert Robere, Jeroen Zuiddam, and Aaron Potechin for many helpful discussions.

1 Introduction

In computational complexity, there is often a focus on analyzing the *worst-case* scenario for a given computation model C and a given function f, but there are other natural cases to consider. One such case is *amortized* computation, where our C algorithm computes many copies of f in such a way that the average cost per copy may be much less than the worst-case cost of computing f a single time.

[1] Now at Amazon, Toronto, Canada.

Amortized analysis has typically been used in the context of (time-bounded) Turing Machines, but studying the amortized complexity of syntactic – and in particular non-uniform – computation models goes back just as far, such as Uhlig's results on circuits computing f on m different inputs. The study of amortized analysis for *branching programs*, a model corresponding to non-uniform space-bounded complexity, was initiated by Potechin [11] and later standardized by Robere and Zuiddam [12] for branching programs and other syntactic models.

The amortized model was introduced by [8] in a different context, namely as a non-uniform version of *catalytic space*, originally introduced in the uniform setting by Buhrman et al. [4] as a new type of space-bounded complexity class. In a catalytic Turing Machine, there are four tapes: as in a traditional space-bounded Turing Machine there is a read-only input tape of length n, a write-only output tape of length 1, and a read-write work tape of length $s(n)$, but additionally we have a read-write *catalytic tape* of length $2^{O(s(n))}$. Ordinarily a tape of this length would allow us to capture $SPACE(2^{O(s(n))})$ rather than $SPACE(s(n))$, but the catalytic tape comes with a catch: the entire tape is initialized to an arbitrary string τ, and at the end of the computation it must contain that same string τ. Since the algorithm must work for every string τ (so, for example, τ cannot be compressed), it seems as if we should get no additional power over $SPACE(s(n))$. Buhrman et al. defied this intuition, showing that when $s(n) = O(\log n)$, the catalytic tape allows us to compute any function in TC^1, a class which as far as we know is larger even than $NL = NSPACE(s(n))$.

Going to the non-uniform setting, an m-catalytic branching program for f [8] is defined as having m start nodes, m 1-end nodes, and m 0-end nodes, where if we restrict to any particular start node we get an ordinary branching program for f with a single start, 1-end, and 0-end node, all of which are distinct for each different choice of the start node. To see the connection to catalytic space, we can think of each start node as corresponding to a different setting of a $\log m$-length catalytic tape, where the "restoration" of the catalytic tape is captured by the fact that the two end nodes corresponding to any start node are unique to that start node. After defining this model and showing multiple results extending the uniform arguments in [4], Girard, Koucký, and McKenzie [8] left open the question of how large of an m-catalytic branching program is required to compute an *arbitrary* function f.

As observed by Potechin [11], this definition is also a natural interpretation of branching programs in the amortized world, as our m-catalytic branching program can be thought of as computing the function f m times. Thus in approaching the question left open by [8], they also settled the question of the amortized space required for an arbitrary function f; they showed that *every* function f has an m-catalytic branching program of size $O(mn)$, or in other words amortized size $O(n)$. The only catch is that the number of copies is doubly exponential; specifically, there exists a (layered) m-catalytic branching program of width $2m$ and length $4n$, where $m = 2^{2^n-1}$. The branching program also has the nice property of reading each input variable exactly 4 times, and thus this also has implications for *read-k branching programs* for $k = O(1)$.

In terms of amortized size, the result of [11] is clearly optimal up to constant factors, and so following [8] the central open question they posed is to understand whether or not m can be improved while maintaining linear amortized size, and what the implications of this result may be. Taking up this challenge, Robere and Zuiddam [12] showed that any function f can be computed by an m-catalytic branching program with the same parameters as [11] even when $m = 2^{\binom{n}{\leq d}-1}$, where d is the degree of f as an $\mathbb{F}_2$ polynomial. Unfortunately this doesn't allow us to go beyond [11] for most functions, but it provides a much sharper analysis for many functions that still appear quite difficult. The proof uses properties of $\mathbb{F}_2$ polynomials under permutations of the input variables.

1.1 Our results

1.1.1 Main result

While the m-catalytic branching programs of [11, 12] can be viewed as catalytic algorithms by definition, our initial aim is to restate these algorithms using the basic catalytic tools derived in [4] and follow-up works. In particular we reprove their results using the natural non-uniform variant of *register programs*, which were defined by Ben-Or and Cleve [3] as space-bounded machines for computing simple arithmetic operations and were also the model used in [4] for their results. Our non-uniform register program follows by extending previous work on catalytic algorithms for the *tree evaluation problem* [5, 6], by adapting their register program to optimize time rather than space.

More importantly, as a result of this connection, we can also exploit a trade-off between space and time – here corresponding to m and length, respectively – in order to strongly break the 2^{2^n-1} barrier for arbitrary functions. In Section 3, we show:

▶ **Theorem 1.** *For any function $f : \{0,1\}^n \to \{0,1\}$ and any $\epsilon \geq 2n^{-1}$, there exists an m-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2\epsilon n$ computing f, where $m = 2^{n + \frac{1}{\epsilon} \cdot 2^{\epsilon n}}$.*

Furthermore, if f is a degree-d polynomial over $\mathbb{F}_2$ and $\epsilon \geq 2d^{-1}$, there is an m-catalytic branching program of width $2m$ and length $2^{1/\epsilon} \cdot 2n$ and computing f, where $m = 2^{n + \frac{1}{\epsilon}\binom{n}{\leq \epsilon d}}$.

Focusing on the case when $\epsilon = \Omega(1)$, this gives us a read-$O(1)$ m-catalytic branching program with m significantly less than 2^{2^n-1} for every function, as well as significantly less than $2^{\binom{n}{\leq d}-1}$ for degree d functions.

1.1.2 Other results

As a bonus, this interpretation also allows us to show significant improvements on m (while still maintaining linear amortized size) for some functions not covered by [12], in particular all functions in TC^0, the class of functions computable by low-depth threshold circuits of polynomial size. By allowing the amortized size to increase to quasilinear, we can capture the much larger class VP, which is the class of all polynomials computable by poly-size arithmetic circuits. See the full version of the paper for proofs and discussions of these results.

▶ **Theorem 2.** *Let f be a function which can be computed by a Boolean circuit C which has depth $O(1)$, size $\mathrm{poly}(n)$, and consists of unbounded fan-in MAJ gates. Then f can be computed by an m-catalytic branching program of length $O(n)$ and width $3m$, where $m = 2^{\mathrm{poly}(n)}$.*

Let $\mathbb{F} \in \{\mathbb{F}_{p \in [\mathrm{poly}\, n]}, \mathbb{Z}, \mathbb{Q}\}$, and let f be a function which can be computed by an arithmetic circuit[2] C over $\mathbb{F}$ which has depth $O(\log n)$, size $\mathrm{poly}(n)$, and consists of unbounded fan-in $+$ gates and fan-in 2 $\times$ gates. Then for any $\epsilon > 0$, f can be computed by an m-catalytic branching program of length $O(n^\epsilon) \cdot n$ and width $3m$, where $m = 2^{n^{O(1/\epsilon)}}$.

In Section 4, we study whether tradeoffs can be made in the other direction, namely whether length $4n$ is optimal for m-catalytic branching programs of width $O(m)$. We show that a few modifications can bring the length down even while slightly improving the original

[2] Our notion of an arithmetic circuit computing a Boolean function is the *Boolean part* of the circuit class, meaning that for all inputs coming from $\{0,1\}^n$, the polynomial the circuit computes will either evaluate to 0 or 1 over $\mathbb{F}$. See e.g. [1] for more discussion of such models.

parameter $m = 2^{2^n-1}$. As with the results of [11, 12] and our improvements, this program is not only layered with optimal width $2m$, but in fact can be made a *permutation branching program*, meaning that each layer has exactly $2m$ nodes and the transition between layers is restricted to be a permutation.[3]

▶ **Theorem 3.** *For every function f, there is a read-4 permutation branching program of width $2^{n+2^{n-1}}$ and length $4n-4$ computing f.*

To complement this result, we show that for such restricted programs and *any* m, even the AND function cannot be computed by permutation programs of length less than $3n$ once $n \geq 4$. In fact our lower bound is very suggestive; it shows that any attempt to read any variable less than three times results in a program exactly matching Theorem 3.

▶ **Theorem 4.** *Any permutation branching program computing AND$(x_1, \ldots, x_n)$ which reads some variable at most twice must have length at least $4n-4$.*

Together these two results leave three questions: 1) what, between $3n$ and $4n-4$, is the shortest length of a permutation branching program for an arbitrary function?; 2) can m can be improved for programs of length $3n$ – or in general any fixed length?; and 3) can we get any improvements by moving to more general programs?

2 Preliminaries

We introduce two space-bounded models of computation. Our first model is a variation of *branching programs*, which are the standard syntactic (i.e. non-uniform) notion of space-bounded computation (see [7]).

▶ **Definition 5** (m-catalytic branching program [8])**.** *Let $n \in \mathbb{N}$ and let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary function. An m-*catalytic branching program *is a directed acyclic graph G with the following properties:*

- *There are m source nodes and $2m$ sink nodes.*
- *Every non-sink node is labeled with an input variable x_i for $i \in [n]$, and has two outgoing edges labeled 0 and 1.*
- *For every source node v there is one sink node labeled with $(v, 0)$ and one with $(v, 1)$.*

We say that G computes *f if for every $x \in \{0,1\}^n$ and source node v, the path defined by starting at v and following the edges labeled by the value of the x_i labeling each node ends at the sink labeled by $(v, f(x))$. The* size *of G is the number of non-sink nodes[4] in G.*

We also consider m-catalytic branching programs with standard restrictions:

- *layered branching programs*: for some $\ell \in \mathbb{N}$, there exists a function $\sigma : G \to [\ell+1]$ such that for all $u \in G$, the outgoing edges of u go to nodes v, w such that $\sigma(v) = \sigma(w) = \sigma(u) + 1$; we call the set of nodes $\{v \in \sigma^{-1}(j)\}$ the jth *layer*. Furthermore for each $j \in [\ell]$ there exists an input variable x_{j_i} which is the variable labeling every $v \in \sigma^{-1}(j)$.[5] The *width* of G is $\max_{j \in [\ell]} |\sigma^{-1}(j)|$ and the *length* of G is ℓ. Note that the size of G is at most the product of the length and width of G.

[3] It is likely m can be improved to $2^{2^{n-1}}$ by arguing directly about the branching program. E.g. our Theorem 9 loses a similar factor compared to Potechin's Theorem 3.1 [11], by using register programs.

[4] This is somewhat non-standard, but when talking about layered branching programs this simplifies things by defining the length as the number of times we read variables, which will in turn be connected to the number of instructions in our register program model. This choice does not affect the asymptotics of any results.

[5] By construction layer $\ell + 1$ will contain exactly the sink nodes of G. See the previous footnote for an explanation of this somewhat non-standard convention.

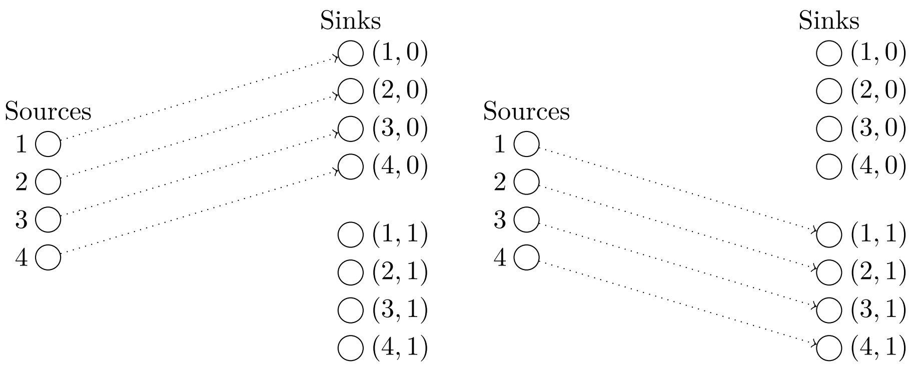


Figure 1 The computation of an m-catalytic branching program starting at source node i ends at sink node $(i, 0)$ if $f(x) = 0$ (left) or $(i, 1)$ if $f(x) = 1$ (right). In these diagrams, $m = 4$.

- *read-k branching programs*: for any start node v and any input x, each variable x_i is read at most k times during the computation starting at v. We note that for layered branching programs, this corresponds to each variable x_j labeling at most k layers.
- *permutation branching programs*: consider a layered branching program of width $2m$ with $2m$ source nodes $S = \{s_1 \dots s_{2m}\}$ and $2m$ sink nodes $T = \{t_1 \dots t_{2m}\}$, and let $P(x) : [2m] \to [2m]$ be the function such that $P(x)(i) = i'$ where the computation path starting from s_i on input x goes to $t_{i'}$. Then there exists a permutation $\sigma^* : [2m] \to [2m]$ such that $P(x)$ is the identity function when $f(x) = 0$ and $P(x) = \sigma^*$ when $f(x) = 1$. This is not an m-catalytic branching program in the strict sense, and we will address this model more precisely in Section 4.

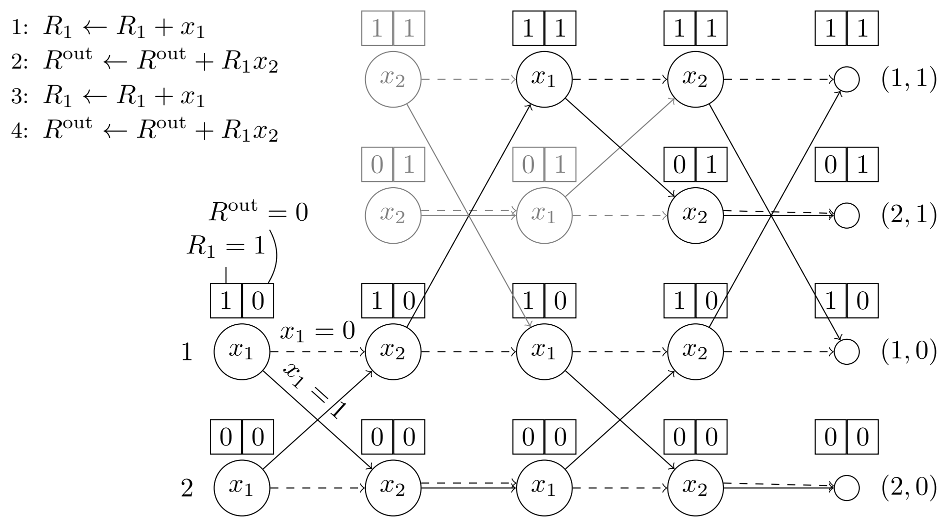
Our second model comes from a line of work starting with [3], more recently fleshed out in [4] and used in many follow-up works on catalytic computation [5, 6].

▶ **Definition 6** (Transparent register program). *Let $\mathcal{R}$ be a ring and $f \in \{0,1\}^n \to \{0,1\}$ a function. A* register program *P is defined by a set of registers $\mathcal{S} = \{R_1 \dots R_s, R^{\text{out}}\}$, each storing a value in $\mathcal{R}$, plus an ordered list of t instructions where for every $j \in [t]$ the jth instruction is $R \leftarrow R + p_j(x_i, \mathcal{S} \setminus \{R\})$ for some $i \in [n]$, $R \in \mathcal{S}$, and polynomial $p_j \in \mathcal{R}[x, y_0 \dots y_s]$.*

We say that P computes f if for every $x \in \{0,1\}^n$, after initializing $R = 0$ for all registers R and then executing all instructions in order, the value stored in R^{out} is $f(x)$. We say that P transparently *computes f if instead of initializing all R to 0, each R begins in an arbitrary initial state τ_i, and at the end of the program we have $R^{\text{out}} = \tau^{\text{out}} + f(x)$ and $R_i = \tau_i$ for all i. The* size *of P is the number of registers $s + 1$ and the* time *of P is the number of instructions t.*

We use register programs as an abstraction for understanding m-catalytic branching programs, which will be the model our main results will refer to. The two models are connected through the following observation (see Figure 2 for an example of our construction).

▶ **Observation 7.** *Let $f_n : \{0,1\}^n \to \{0,1\}$ be a family of functions and let P_n be a family of register programs over a family of rings $\mathcal{R}_n$ with size $s(n)+1$ and time $t(n)$ each transparently computing f_n. Then f_n can be computed by a family of m-catalytic branching programs of width $|\mathcal{R}_n|m$ and length $t(n)$, where $m = |\mathcal{R}_n|^{s(n)}$.*



```
1: $R_1 \leftarrow R_1 + x_1$
2: $R^{\text{out}} \leftarrow R^{\text{out}} + R_1 x_2$
3: $R_1 \leftarrow R_1 + x_1$
4: $R^{\text{out}} \leftarrow R^{\text{out}} + R_1 x_2$
```

Figure 2 A transparent register program computing AND(x_1, x_2), and its realization as a 2-catalytic branching program using Observation 7. Each node is annotated (above, in boxes) with the register values it stores, and each non-sink node is labelled (inside the circle) with the input to be read. Dashed lines are transitions taken when that input is zero, and solid lines are taken when the input is one. Finally, the source nodes are assigned the numbers 1 and 2 (since $m = 2$), and the sink nodes are assigned pairs of numbers, so that like in the more abstract Figure 1, a computation starting at source node v will end at end at sink node $(v, f(x))$. Note that nodes/edges that appear in gray are unreachable from the start states and thus would not appear in the final branching program; they appear only for reference as to how the register program translates to a branching program.

Proof. We will define a branching program with width $|\mathcal{R}_n|^{s(n)+1}$ and length $t(n)$. Each layer will represent a stage in the register program and each node in a layer will represent a setting to the registers at that time. Since each register program step only requires us to read one input variable, at layer k we query the variable associated with step k in the register program and create edges from each node in layer k to the nodes at layer $k+1$ representing the state of our memory after the step has completed. We label each input node as τ for some distinct initial configuration $\tau = (\tau_1 \dots \tau_{s(n)})$ to all registers except R^{out}, and we treat R^{out} as being initialized to 0. Then by the fact that P transparently computes f, starting at node τ we are guaranteed to reach a node $(\tau, f(x))$. Since in each layer we have a node for every setting of the $s(n)+1$ registers, the width of our branching program is $|\mathcal{R}_n|^{s(n)+1}$, and since each non-output layer corresponds to a unique instruction from our register program, the length of our branching program is $t(n)$. ◀

▶ Note 8. In our computation we often include instructions of the form $R \leftarrow R + p_j(\mathcal{R}/\{R\})$, i.e. instructions that do not require reading a variable x_i. By observing the above proof, it is clear that such instructions do not add any length to the branching program, as they can be computed at the same time as an adjacent instruction.

3 Saving Space

In this section we show that every function can be computed by an m-catalytic branching program with size $O(mn)$ for $m \ll 2^{2^n - 1}$ (improving on [11]) and $m \ll 2^{\binom{n}{\leq d} - 1}$ (improving on [12]). We present our algorithm in three steps:

- In Section 3.1 we show a simpler version of our algorithm which is sufficient to reproduce – with a negligible loss in parameters – Potechin's result [11] that any function can be computed with a linear-amortized-size m-catalytic branching program. Our program has length $4n$ and width $2m$, where $m = 2^{2^n+n}$.
- In Section 3.2 we show how to trade off between m and amortized size, yielding for every $k \in [d]$ an m-catalytic branching program of length $2^k \cdot 4\lceil n/k \rceil$ and width $2m$, where $m = 2^{k \cdot 2^{\lceil n/k \rceil}+n}$.
- In Section 3.3 we show a simple modification of our first algorithm which reproduces – again with a negligible loss – the result of Robere and Zuiddam [12] that m can be made as small as $2^{\binom{n}{\leq d}-1}$, where d is the degree of f as an $\mathbb{F}_2$ polynomial, with no cost to the length. Our program has length $4n$ and width $2m$, where $m = 2^{\binom{n}{\leq d}+n}$. We then show that the tradeoff algorithm gives us an m-catalytic branching program of length $2^k \cdot 2n$ and width $2m$, where $m = 2^{k \cdot \binom{n}{\leq \lceil d/k \rceil}+n}$.[6]

In these sections, our register programs will all operate over the field of two elements: $\mathcal{R}_n = \mathbb{F}_2$ for all n.

3.1 Basic Algorithm

In this section, we will prove:

▶ **Theorem 9.** *For any function $f : \{0,1\}^n \to \{0,1\}$ there is an m-catalytic branching program with length $4n$ and width $2m$ that computes f, where $m = 2^{n+2^n}$.*

This is proved by Algorithm 1 below. This nearly reproduces Potechin's Theorem 3.1 [11], but with a worse value $m = 2^{n+2^n}$ instead of 2^{2^n-1}. Nonetheless, we will find it a useful starting point for our algorithm that trades space for time in Section 3.2.

Proof. We will design a program that uses $n+2^n$ work registers plus one output register R^{out}, which is sufficient by Observation 7. First, we have n registers $R_1^{\text{in}}, \ldots, R_n^{\text{in}}$, corresponding to the n input bits. This correspondence is given by the following subroutine:

```
1: procedure ToggleInput
2:     for $i = 1, \ldots, n$ do
3:         $R_i^{\text{in}} \leftarrow R_i^{\text{in}} + x_i$
4:     end for
5: end procedure
```

After ToggleInput runs, the registers have values $R_i^{\text{in}} = \tau_i^{\text{in}} + x_i$, where τ_i^{in} stands for the initial value of R_i^{in}. If we run it a second time, the registers are restored to their original values: $R_i^{\text{in}} = \tau_i^{\text{in}}$. Since we query all n variables once, ToggleInput requires time n to run once.

Before defining our other 2^n registers, we introduce an algebraic view of f, which will be our main focus. We can view f as a multilinear polynomial $p_f \in \mathbb{F}_2[x_1, \ldots, x_n]$ using basic interpolation:

▶ **Lemma 10.** *For any function $f : \{0,1\}^n \to \{0,1\}$ there is a multilinear polynomial $p_f \in \mathbb{F}_2[x_1, \ldots, x_n]$ such that $p_f(\vec{x}) = f(\vec{x})$ for all $\vec{x} \in \{0,1\}^n$.*

[6] While the program of [12] matches or beats [11] for all d, our improved version of [12] is worse than our improved version of [11] when $d = \Omega(n)$ (although still an improvement over the original results of both papers), and thus we state and prove both results separately rather than subsuming our improved version of [11].

Proof. For any $\vec{y} \in \{0,1\}^n$, we can algebraically define the indicator function $[\vec{x} = \vec{y}]$ as $\prod_{i=1}^{n}(x_i + y_i + 1) \in \mathbb{F}_2[x_1, \ldots, x_n]$. Then we can set

$$p_f = \sum_{\vec{y} \in \{0,1\}^n : f(\vec{y})=1} [\vec{x} = \vec{y}] \qquad \blacktriangleleft$$

Now define $y_i = \tau_i^{\text{in}} + x_i$; in other words, y_i is the value of R_i^{in} after running ToggleInput. Define $q_f(y_1, \ldots, y_n, \tau_1^{\text{in}}, \ldots, \tau_n^{\text{in}}) = p_f(y_1 - \tau_1^{\text{in}}, \ldots, y_n - \tau_n^{\text{in}})$ to be the result of rewriting p_f using the y_i and τ_i variables.[7] For $S, S' \subseteq [n]$, let $c_{S,S'}$ be the coefficient of $\left(\prod_{i \in S} \tau_i^{\text{in}}\right) \cdot \left(\prod_{i \in S'} y_i\right)$ in q_f, so that

$$q_f(\overrightarrow{\tau^{\text{in}}}, \vec{y}) = \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau_i^{\text{in}}\right)\left(\prod_{i \in S'} y_i\right)$$

Note that S and S' are disjoint whenever $c_{S,S'} \neq 0$, because p_f is multilinear. We now introduce our other registers: we have 2^n registers R_S indexed by subsets $S \subseteq [n]$. Our next subroutine prepares us to use these registers to compute q_f:

```
1: procedure ToggleMonomials
2:     ToggleInput
3:     for $S' \subseteq [n]$ do
4:         $R_{S'} \leftarrow R_{S'} + \prod_{i \in S'} R_i^{\text{in}}$
5:     end for
6:     ToggleInput
7: end procedure
```

After ToggleMonomials runs, we have $R_S = \tau_S + \prod_{i \in S} y_i$ for each $S \subseteq [n]$, where τ_S stands for the register's initial value. The R^{in} registers have their initial values $R_i^{\text{in}} = \tau_i^{\text{in}}$. We run ToggleInput twice and have 2^n additional instructions, but since the additional instructions do not query any x variables they can be computed in the last x query of ToggleInput, for a total runtime of $2n$.

Our final algorithm for computing f is:

Algorithm 1 Transparently compute f in time $4n$ with $n + 2^n + 1$ registers.

```
1: ToggleMonomials
2: $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} R_i^{\text{in}}\right) R_{S'}$
3: ToggleMonomials
4: $R^{\text{out}} \leftarrow R^{\text{out}} - \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} R_i^{\text{in}}\right) R_{S'}$
```

When Line 2 executes, we have $R_{S'} = \tau_{S'} + \prod_{i \in S'} y_i$, so after that line,

$$R^{\text{out}} = \tau^{\text{out}} + \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau_i^{\text{in}}\right)\left(\tau_{S'} + \prod_{i \in S'} y_i\right).$$

Then when Line 4 executes, we have $R_{S'} = \tau_{S'}$, so the final value is

$$R^{\text{out}} = \tau^{\text{out}} + \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau_i^{\text{in}}\right)\left(\prod_{i \in S'} y_i\right) = \tau^{\text{out}} + f(x_1, \ldots, x_n).$$

[7] For example, if f is the OR function $f(x_1, x_2) = x_1 \vee x_2$, we have $p_f(x_1, x_2) = x_1 + x_2 + x_1 x_2$ and $q_f(\tau_1^{\text{in}}, \tau_2^{\text{in}}, y_1, y_2) = y_1 + \tau_1^{\text{in}} + y_2 + \tau_2^{\text{in}} + y_1 y_2 + y_1 \tau_2^{\text{in}} + \tau_1^{\text{in}} y_2 + \tau_1^{\text{in}} \tau_2^{\text{in}}$, and both are equal to $f(x_1, x_2)$ so long as $\vec{y}$ have the correct values.

So Algorithm 1 correctly computes f. The space of the program is $n + 2^n$ by construction, and as before we can ignore the instructions on lines 2 and 4 since they do not use x, giving us a total runtime of $4n$ from the two calls to ToggleMonomials. ◀

3.2 Trading Space for Time

In this section, we will modify Algorithm 1 to make m dramatically smaller, in exchange for making the branching program longer.

▶ **Theorem 11.** *For any $k \in \mathbb{N}$ and any function $f : \{0,1\}^n \to \{0,1\}$ there is an m-catalytic branching program with length $2^k \cdot 2\lceil n/k \rceil$ and width $2m$ that computes f, where $m = 2^{n+k \cdot 2^{\lceil n/k \rceil}}$.*

Before jumping into the proof of Theorem 11, we will address the main innovation of our work, namely trading off time for space. Namely we begin by building a register program that takes time $n2^{n+1}$ but uses only the $n+1$ registers $R^{\text{in}}_1 \dots R^{\text{in}}_n, R^{\text{out}}$. This is similar to what Theorem 11 guarantees when $k = n$.

As in Sections 3.1 and 3.3, let $p_f \in \mathbb{F}_2[x_1, \dots, x_n]$ be f as a polynomial, let $q_f \in \mathbb{F}_2[\tau^{\text{in}}_1, \dots, \tau^{\text{in}}_n, y_1, \dots, y_n]$ be equal to p_f so long as $y_i = \tau^{\text{in}}_i + x_i$ for all i, and let $c_{S,S'}$ be the coefficient of $\left(\prod_{i \in S} \tau^{\text{in}}_i\right)\left(\prod_{i \in S'} y_i\right)$ in q_f. We define a small generalization of ToggleInput, where we can choose to toggle only a subset of our inputs:

```
1: procedure ToggleSomeInputs(S')
2:     for $i \in S'$ do
3:         $R^{\text{in}}_i \leftarrow R^{\text{in}}_i + x_i$
4:     end for
5: end procedure
```

Using ToggleSomeInputs(S'), we can replace the register $R_{S'}$ in Algorithm 1 with a separate set of instructions that computes the corresponding term of q_f:

Algorithm 2 A slow algorithm for computing q_f.

```
1: for $S' \subseteq [n]$ do
2:     ToggleSomeInputs(S')
3:     $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S \subseteq [n]} c_{S,S'} \cdot \prod_{i \in S \cup S'} R^{\text{in}}_i$
4:     ToggleSomeInputs(S')
5: end for
```

Whenever Line 3 is executed, $R^{\text{in}}_i = y_i$ for $i \in S'$, and $R^{\text{in}}_i = \tau^{\text{in}}_i$ for $i \notin S'$. By construction of q_f, S and S' are disjoint whenever $c_{S,S'} \neq 0$, from which it follows that $R^{\text{in}}_i = \tau^{\text{in}}_i$ for $i \in S$. Thus the effect of Line 3 is to add $\sum_{S \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau^{\text{in}}_i\right)\left(\prod_{i \in S'} y_i\right)$ to R^{out}. Since this is run for every subset S', the overall effect of the program is to add

$$\sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau^{\text{in}}_i\right)\left(\prod_{i \in S'} y_i\right) = p_f(x_1 \dots x_n)$$

to R^{out}.

We now give an overview of the full proof. Our goal is to balance Algorithms 1 and 2 by removing the registers R_S corresponding to large subsets S and using slow multiplication to build the polynomial q_f from the remaining small subsets. In particular, if we divide the input bits into k groups each of size $\lceil n/k \rceil$, and only store all subsets within each group, then any monomial $c_{S,S'}\prod_{i\in S}\tau_i^{\text{in}}\prod_{i\in S'}y_i$ can be computed by multiplying together one subset from each group, namely the restriction of S to the group. Instead of 2^n registers for all subsets, we use only $k\cdot 2^{\lceil n/k\rceil}$ registers corresponding to subsets in the k groups, and we can compute all the corresponding monomials into these registers in time $2n$ using the first half of Algorithm 1. Then since we are only multiplying k monomials together, we can compute q_f using Algorithm 2 in time $2^k\cdot 2\cdot 2n$, since each call to ToggleSomeInputs is replaced with our $2n$ time execution of Algorithm 1.

As a slight last speedup, we use a Gray code to order our subsets in Algorithm 2, replacing two executions of ToggleSomeInputs with a subroutine toggling a single group on or off and only spending $2\lceil n/k\rceil$ time to do so. This allows us to compute q_f in $2^k\cdot 2\lceil n/k\rceil$ time rather than $2^k\cdot 4n$ time.

Proof of Theorem 11. For $j\in[k]$ let $b_j=\lceil nj/k\rceil$, and divide the range $[n]$ into k groups: $G_1=\{1,\ldots,b_1\},G_2=\{b_1+1,\ldots,b_2\},\ldots,G_k=\{b_{k-1}+1,\ldots,b_n=n\}$. For each group G_j, we have $2^{|G_j|}$ registers $R_{j,S}$ indexed by subsets $S\subseteq G_j$. As in all previous algorithms we also use n registers $R_1^{\text{in}},\ldots,R_n^{\text{in}}$, corresponding to the n input bits, for a total of $n+\sum_{j=1}^k 2^{|G_j|}$ registers plus the output register R^{out}.

We'll begin by rewriting the polynomial q_f in a new form. Recall from Section 3.1 that $q_f(\overrightarrow{\tau^{\text{in}}},\vec{y})=f(x)$ so long as $\vec{y}=\vec{x}+\overrightarrow{\tau^{\text{in}}}$, and

$$q_f(\overrightarrow{\tau^{\text{in}}},\vec{y})=\sum_{S,S'\subseteq[n]}c_{S,S'}\left(\prod_{i\in S}\tau_i^{\text{in}}\right)\left(\prod_{i\in S'}y_i\right)$$

Now, for every $S'\subseteq[n]$, define $S'_j:=S'\cap G_j$. For each $j\in[k]$ and $S\subseteq G_j$, let $z_{j,S}=\tau_{j,S}+\prod_{i\in S}y_i$, where $\tau_{j,S}$ is the initial value of register $R_{j,S}$. Now for every monomial in q_f, we split the term $\prod_{i\in S'}y_i$ in the monomial into k different products $\prod_{i\in S'_j}y_i$, each of which we can replace with $z_{j,S'_j}-\tau_{j,S'_j}$. This gives us a new polynomial

$$r_f(\overrightarrow{\tau^{\text{in}}},\vec{\tau},\vec{z})=\sum_{S,S'\subseteq[n]}c_{S,S'}\left(\prod_{i\in S}\tau_i^{\text{in}}\right)\left(\prod_{j=1}^{k}(z_{j,S'}-\tau_{j,S'})\right).$$

As we did with q_f, for $S,S'\subseteq[n]$ and $T\subseteq[k]$, let $d_{S,S',T}$ be the coefficient of $(\prod_{i\in S}\tau_i^{\text{in}})\cdot(\prod_{j\in T}z_{j,S'_j})\cdot(\prod_{j\in[k]\setminus T}\tau_{j,S'_j})$ in r_f, so that

$$r_f(\overrightarrow{\tau^{\text{in}}},\vec{\tau},\vec{z})=\sum_{S\subseteq[n]}\sum_{S'\subseteq[n]}\sum_{T\subseteq[k]}d_{S,S',T}\left(\prod_{i\in S}\tau_i^{\text{in}}\right)\left(\prod_{j\in T}z_{j,S'_j}\right)\left(\prod_{j\in[k]\setminus T}\tau_{j,S'_j}\right)$$

which is equivalent to $f(x_1\ldots f_n)$ as long as $z_{j,S'_j}=\tau_{j,S'_j}+\prod_{i\in S'_j}(x_i+\tau_i^{\text{in}}+1)$.

Following ToggleSomeInputs(S'), we define new versions of ToggleInput and ToggleMonomials from Section 3.1 which focus on some groups and not others. In fact we will only focus on a single group G_j rather than a subset of the groups, as we will order our subsets S' in such a way that we will only ever need to toggle one group at a time:

```
1: procedure ToggleInputForGroup(j)
2:     for $i \in G_j$ do
3:         $R_i^{\text{in}} \leftarrow R_i^{\text{in}} + x_i$
4:     end for
5: end procedure
```

```
1: procedure ToggleMonomialsForGroup(j)
2:     ToggleInputForGroup(j)
3:     for $S \subseteq G_j$ do
4:         $R_{j,S} \leftarrow R_{j,S} + \prod_{i \in S} R_i^{\text{in}}$
5:     end for
6:     ToggleInputForGroup(j)
7: end procedure
```

We are now ready to assemble Algorithm 4, which completes the proof of Theorem 11. To incorporate our improvement using Gray codes [9], let $T_0 = \emptyset, \ldots, T_{2^k-1}$ be an ordering of all subsets of $[k]$ such that each consecutive pair of sets $T_\ell, T_{\ell+1 \bmod 2^k}$ differs by exactly one element $e_\ell \in [k]$; thus we will only need to toggle the group G_{e_ℓ} as claimed:

Algorithm 3 Transparently compute f in time $2^k \cdot 2\lceil n/k \rceil$ with $n + k \cdot 2^{\lceil n/k \rceil}$ registers.

```
1: for $\ell = 0, \ldots, 2^k - 1$ do
2:     $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S \subseteq [n]} \sum_{S' \subseteq [n]} d_{S,S',T_\ell} \left(\prod_{i \in S} R_i^{\text{in}}\right) \left(\prod_{j=1}^{k} R_{j,S'_j}\right)$
3:     ToggleMonomialsForGroup($e_\ell$)
4: end for
```

Each time Line 2 is reached, we have $R_{j,S} = \tau_{j,S} + \prod_{i \in S} y_j$ for $j \in T_\ell$, and $R_{j,S} = \tau_{j,S}$ for $j \in [k] \setminus T_\ell$. We also have $R_i^{\text{in}} = \tau_i^{\text{in}}$ for each $i \in [n]$. So the effect of the line is to add

$$\sum_{S \subseteq [n]} \sum_{S' \subseteq [n]} d_{S,S',T_\ell} \left(\prod_{i \in S} \tau_i^{\text{in}}\right) \left(\prod_{j \in T_\ell} z_{j,S'_j}\right) \left(\prod_{j \in [k] \setminus T_\ell} \tau_{j,S'_j}\right)$$

to R^{out}. Summing this expression over all possible subsets $T_\ell \subseteq [k]$ gives $R^{\text{out}} = \tau^{\text{out}} + r_f(\cdots) = \tau^{\text{out}} + f(x_1, \ldots, x_n)$, and so our algorithm transparently computes f. ◀

▶ Note 12. It is not difficult to save k registers by removing $R_{1,\emptyset}, \ldots, R_{k,\emptyset}$, as we simply add each value $d_{S,\emptyset,T}(\prod_{i \in S} R_i^{\text{in}})$ to our polynomial without concerning ourselves with any x_i (and by extension any y_i or z_i) variables.

3.3 Bounded-Degree Polynomials

Robere and Zuiddam [12, Theorem 5.13] showed that if f is a polynomial of degree $d < n$, it is possible to improve on Potechin's theorem by decreasing $m = 2^{2^n - 1}$ down to $m = 2^{\binom{n}{\leq d} - 1}$. Here we show how to adapt Algorithm 1 to get a similar result, and then at the end of the section we build a tradeoff algorithm to improve it.

▶ **Theorem 13.** *For any function $f : \{0,1\}^n \to \{0,1\}$ which is a degree-d polynomial, there is an m-catalytic branching program with length $4n$ and width $2m$ that computes f, where $m \leq 2^{n + \binom{n}{\leq d}}$.*

Again, while this is slightly worse than Robere and Zuiddam's original result, we include it to show the flexibility of our approach and as a stepping stone to our tradeoff result.

Proof. The proof can be summarized as follows. We start with Algorithm 1, and make the following change: *for every $S' \in [n]$ such that $c_{S,S'} = 0$ for all S, remove the register $R_{S'}$.* We then argue that only $n + \binom{n}{\leq d}$ registers remain.

Let $p_f = f \in \mathbb{F}_2[x_1, \dots, x_n]$ – since f is already a polynomial, there's no need to define p_f using interpolation this time. As before, let

$$q_f(\overrightarrow{\tau^{\text{in}}}, \vec{y}) = p_f(\vec{y} - \overrightarrow{\tau^{\text{in}}}) = \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau_i^{\text{in}}\right) \left(\prod_{i \in S'} y_i\right)$$

Let $\mathcal{M}_f \subseteq 2^{[n]}$ be the set of all S' such that there exists an S where $c_{S,S'} \neq 0$. We define the following subroutine:

```
1: procedure ToggleUsefulMonomials
2:     ToggleInput
3:     for $S \in \mathcal{M}_f$ do
4:         $R_S \leftarrow R_S + \prod_{i \in S} R_i^{\text{in}}$
5:     end for
6:     ToggleInput
7: end procedure
```

The only difference from ToggleMonomials is that we ignore subsets S which are not in $\mathcal{M}_f$ (not "useful"). Our final algorithm is

Algorithm 4 Transparently compute a degree-d polynomial f in time $4n$ with $n + \binom{n}{\leq d}$ registers.

```
1: ToggleUsefulMonomials
2: $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S \subseteq [n]} \sum_{S' \in \mathcal{M}_f} c_{S,S'} \left(\prod_{i \in S} R_i^{\text{in}}\right) R_{S'}$
3: ToggleUsefulMonomials
4: $R^{\text{out}} \leftarrow R^{\text{out}} - \sum_{S \subseteq [n]} \sum_{S' \in \mathcal{M}_f} c_{S,S'} \left(\prod_{i \in S} R_i^{\text{in}}\right) R_{S'}$
```

To conclude the proof of Theorem 13, we need to show $|\mathcal{M}_f| \leq \binom{n}{\leq d}$. Indeed, since p_f is a degree-d polynomial, q_f also has degree d, which means $c_{S,S'} = 0$ whenever $|S| + |S'| > d$. So, $\mathcal{M}_f$ only contains sets S' with size at most d, of which there are $\binom{n}{\leq d}$. ◀

▶ Note 14. The original algorithms of [11, 12] rely on the *symmetries* of f as an $\mathbb{F}_2$ polynomial, in essence having each start state represent a possible function g which can be obtained from f by negating input variables or taking $\oplus$ with f itself. [11] takes this set of functions to be the space of all n-variable functions, while [12] analyzes these rules and obtains a more exact characterization. While this characterization is phrased in terms of orbit closures, it can also be described in terms of polynomials as the span of the set of all monomials appearing in f as an $\mathbb{F}_2$ polynomial along with all *submonomials* of this set; this exactly coincides with our notion as $\prod_{i \in S} y_i$ generates all submonomials $\prod_{i \in S' \subseteq S} x_i$ for $y_i := x_i + \tau_i$, which leads to the quantitative results being essentially the same despite taking two completely different approaches.

Now we state our tradeoff algorithm, which goes much in the same way as Theorem 11 but without breaking the variables into groups.

▶ **Theorem 15.** *For any function $f : \{0,1\}^n \to \{0,1\}$ which is a degree-d polynomial and any $k \in \mathbb{N}$, there is an m-catalytic branching program with length $2^k \cdot 2n$ and width $2m$ that computes f, where $m = 2^{n + k \cdot \binom{n}{\leq \lceil d/k \rceil}}$.*

Proof. For any $\Delta \in \mathbb{N}$, let $\mathcal{M}_f^{\Delta} \subseteq \binom{n}{\leq \Delta}$ be the set of all S'' of size at most Δ such that there exists an $S \subseteq [n]$ and $S' \supseteq S''$ where $c_{S,S'} \neq 0$. We will have k registers $R_{j,S''}$ for every $S'' \in \mathcal{M}_f^{\lceil d/k \rceil}$, as well as the usual registers $R_1^{\text{in}} \dots R_n^{\text{in}}, R^{\text{out}}$. Note that this gives us our target space, as $|\mathcal{M}_f^{\lceil d/k \rceil}| \leq \binom{n}{\leq \lceil d/k \rceil}$.

Following our proof of Theorem 11, let $z_{j,S} = \tau_{j,S} + \prod_{i \in S} y_i$, where $\tau_{j,S}$ is the initial value of register $R_{j,S}$, and for every monomial in q_f we split the term $\prod_{i \in S'} y_i$ in the monomial arbitrarily into k different products $\prod_{i \in S'_j} y_i$ – each of which we can replace with $z_{j,S'_j} - \tau_{j,S'_j}$ – where $S'_j \in \mathcal{M}_f^{\lceil d/k \rceil}$ and $\cup_{j \in [k]} S'_j = S'$. This is possible because each non-zero term in q_f has degree at most d, meaning that $|S'| \leq d$ and furthermore every subset of S' of size at most $\lceil d/k \rceil$ appears in $\mathcal{M}_f^{\lceil d/k \rceil}$ by construction.[8]

Fixing some particular partition $(S'_j)_{j \in [k]}$ for each S', this gives us a new polynomial

$$r_f(\overrightarrow{\tau^{\text{in}}}, \vec{\tau}, \vec{z}) = \sum_{S \subseteq [n]} \sum_{S' \subseteq [n]} \sum_{T \subseteq [k]} d_{S,S',T} \left(\prod_{i \in S} \tau_i^{\text{in}} \right) \left(\prod_{j \in T} z_{j,S'_j} \right) \left(\prod_{j \in [k] \setminus T} \tau_{j,S'_j} \right)$$

which is equivalent to $f(x_1 \dots f_n)$ as long as $z_{j,S'_j} = \tau_{j,S'_j} + \prod_{i \in S'_j} (x_i + \tau_i^{\text{in}} + 1)$. We define ToggleMonomialsForGroup as before, using ToggleInput instead of ToggleInputForGroup since the variables are no longer split into groups, and using a Gray code we get our final algorithm:

Algorithm 5 Transparently compute f in time $2^k \cdot 2n$ with $n + k \cdot \binom{n}{\leq \lceil d/k \rceil}$ registers.

```
1: for $\ell = 0, \dots, 2^k - 1$ do
2:     $R^{\text{out}} \leftarrow R^{\text{out}} + \sum_{S \subseteq [n]} \sum_{S' \subseteq [n]} d_{S,S',T_\ell} \left(\prod_{i \in S} R_i^{\text{in}}\right) \left(\prod_{j=1}^{k} R_{j,S'_j}\right)$
3:     ToggleMonomialsForGroup($e_\ell$)
4: end for
```

The analysis is identical to that of Algorithm 4, except the runtime is $2^k \cdot 2n$ rather than $2^k \cdot 2\lceil n/k \rceil$ because we do not split the variables into groups. ◀

4 Saving Time

In the previous section we took the length $4n$ branching programs of [11, 12] as a starting point to analyze whether m could be significantly reduced while still maintaining a linear amortized size. In this section we investigate the opposite question: namely, is $4n$ optimal? If we do not restrict the amortized size of our program, then every function has a branching program of length n regardless of m. We will not only consider branching programs of linear amortized size, but the stricter model of *permutation branching programs.*

We focus on permutation branching programs in this section for two reasons. First, all our algorithms in the previous section can be converted to permutation branching programs. To see this, we resist our connection between register programs and m-catalytic branching programs, as proven in Observation 7. Our observation in fact says something stronger, which is that f_n can be computed by a family of permutation branching programs of width $2^{s(n)+1}$. This follows because we can choose to not fix R^{out} to be 0, and instead have one source node corresponding to each initial configuration $\tau_1 \dots \tau_s, \tau^{\text{out}}$; by Definition 6 this

[8] Our use of j here is slightly different than in our previous proof; namely, j is not linked to a specific block of variables, and rather we arbitrarily partitioned S' into k sets and assigned them each a distinct j. This will result in us having to spend time n to load the monomials in, rather than time $\lceil n/k \rceil$ as in the previous proof, but this is necessary as we have no guarantee that there is a partition of the variables such that every monomial of degree at most d is split into k monomials of degree at most $\lceil d/k \rceil$. Note that this is where our algorithm performs worse than Algorithm 4 when $d = \Omega(n)$.

source reaches the sink node labeled $\tau_1 \dots \tau_s, \tau^{\text{out}} + f(x)$, which is the identity permutation when $f(x) = 0$ and a fixed set of $2^{s(n)}$ transpositions otherwise. Thus as a corollary of [11, 12] we get the following:

▶ **Theorem 16.** *Every function f can be computed by a read-4 permutation branching program of width 2^{2^n-1} (or $2^{\binom{n}{\leq d}-1}$, where d is the $\mathbb{F}_2$-degree of p_f).*

Given the strength of these results, only a factor of 4 away from the best conceivable amortized size, there is no reason *a priori* to believe that permutation branching programs are too weak to consider even better upper bounds. Indeed we will show that improvements are possible.

This leads to our other reason for focusing on permutation branching programs: lower bounds against general branching programs are notoriously difficult. Besides the basic counting argument, the best known branching program lower bounds for an explicit function are not even quadratic, using techniques known to go no further [10]. Considering the amortized branching program size needed to compute any function f is always at most the basic branching program size, and considering the upper bound of $4n$ given by [11], proving lower bounds for concrete functions seems exceedingly difficult. Furthermore, even if we were to seek refuge in focusing on non-constructive lower bounds, the basic counting argument fails to prove *any* non-trivial lower bounds in the case of $m \geq 2^n/n$.

4.1 Notation and tools

Before going into our results, we formally define permutation branching programs along with the notation we will use in the rest of this section. These programs will have a more specialized form than when we introduced them in Section 2, which we subsequently justify. We assume basic familiarity with permutations, and we write $\sigma_1\sigma_2$ as a shorthand for $\sigma_2 \circ \sigma_1$.

▶ **Definition 17.** *Let $n, m, s \in \mathbb{N}$ and let $f : \{0,1\}^n \to \{0,1\}$ be a function such that $f(0\dots0) = 0$. A* permutation branching program *is a sequence $P = \pi_1 \dots \pi_\ell$, where each π_j is a pair $\langle i_j, \sigma_j \rangle$ where $i_j \in [0..n]$ and σ_j is a permutation of $[m]$. We refer to each π_j as an* instruction *of P. The* width *of P is m and the* length *of P is ℓ.*

For any $\alpha \in \{0,1\}^n$ we define $P(\alpha)$ as follows: fix $\sigma = id$, and for every $j = 1 \dots s$, we set σ to $\sigma\sigma_j$ if $\pi_j = \langle 0, \sigma_j \rangle$ or $\pi_j = \langle i_j, \sigma_j \rangle$ where $\alpha_{i_j} = 1$, and leave σ unchanged otherwise (that is, if $\pi_j = \langle i_j, \sigma_j \rangle$ where $\alpha_{i_j} = 0$). Our output is the final value of σ. We say that P computes f if there exists a permutation $\sigma^* \neq id$ such that $P(\alpha) = id$ if $f(\alpha) = 0$ and $P(\alpha) = \sigma^*$ if $f(\alpha) = 1$.

We make three observations about our choices in Definition 17. First, the restriction that $f(0\dots0) = 0$ will be a convenience; we can always compute $\neg f$ instead if this condition does not hold, or change Definition 17 such that $P(\alpha) = id$ if $f(\alpha) = 1$ and vice versa.[9] Second, in a layer $\langle i, \sigma_j \rangle$ reading variable x_i, we only fix a permutation in the case that x_i is set to 1. This is without loss of generality, as adding a layer of the form $\langle 0, \sigma'_j \rangle$ before an instruction can be thought of as choosing a permutation in the case that x_i is set to 0 (while the permutation for $x_i = 1$ can be adjusted accordingly).

Before going on to our third observation, we state and prove four simple lemmas which will allow us to conveniently restructure our programs P.

[9] We also note that if P computes $\neg f$, we can compute f by appending the instruction $\langle 0, (\sigma^*)^{-1} \rangle$ to P. We avoid taking this route because a later observation will allow us to remove these fixed layers, but only when $f(\alpha) = 0$, which would cause our logic to become circular.

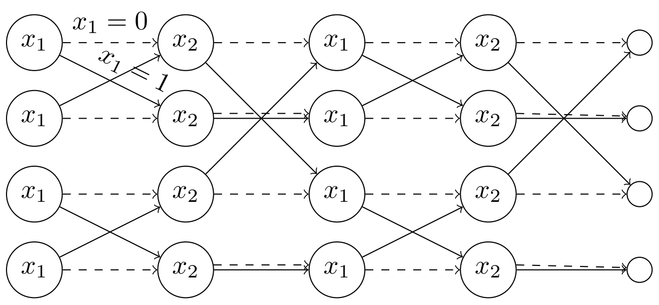


Figure 3 A permutation branching program computing $\mathrm{AND}(x_1, x_2)$. It is identical to the one from Figure 2, except that two more source nodes have been added so that all layers have the same width. As before, each non-sink node is labelled with the input to be read, dashed lines represent transitions taken when that input is zero, and solid lines are taken when the input is one. In the terminology of Definition 17, this program can be written as $\langle 1, (12)(34)\rangle, \langle 2, (13)\rangle, \langle 1, (12)(34)\rangle, \langle 2, (13)\rangle$ (using cycle notation for the permutations).

▶ **Lemma 18.** *Let P be a permutation branching program computing f and let j be such that $i_j = i_{j+1}$. Then the program P' resulting from replacing π_j, π_{j+1} with $\pi'_j = \langle i_j, \sigma_j \sigma_{j+1}\rangle$ is also a valid program for computing f.*

Proof. In both P and P', the permutations σ_j and σ_{j+1} are both applied when $i_j = 1$ and neither are applied when $i_j = 0$. ◀

▶ **Lemma 19.** *Let P be a permutation branching program computing f and let j be such that $\sigma_j \sigma_{j+1} = \sigma_{j+1}\sigma_j$. Then the program P' resulting from switching the order of π_j and π_{j+1} is also a valid program for computing f.*

Proof. Consider any assignment α to x. In the case that either α_{i_j} or $\alpha_{i_{j+1}}$ is set to 0, these programs compute identical permutations as either σ_j or σ_{j+1} will not be applied. If both are set to 1, then

$$P'(\alpha) = \Sigma_1 \sigma_{j+1}\sigma_j \Sigma_2 = \Sigma_1 \sigma_j \sigma_{j+1} \Sigma_2 = P(\alpha)$$

where Σ_1, Σ_2 are the permutations corresponding to the rest of the instructions on input α. ◀

▶ **Lemma 20.** *Let $P = \pi_1 \dots \pi_s$ be a permutation branching program computing f, let $\pi_j = \langle i_j, \sigma_j\rangle$ for all j, and let $j^* \in [\ell]$ be such that $i_{j^*} = 0$. Then there exists a permutation branching program $P' = \pi'_1 \dots \pi'_{j^*-1}\pi'_{j^*+1} \dots \pi'_\ell \pi_{j^*}$ computing f, where $\pi'_j = \langle i_j, \sigma'_j\rangle$ for some permutation σ'_j.*

Proof. For $j < j^*$ define $\sigma'_j = \sigma_j$, and for $j > j^*$ define $\sigma'_j = \sigma_{j^*}\sigma_j\sigma_{j^*}^{-1}$. Clearly because $\sigma_{j^*}^{-1}$ and σ_{j^*} cancel out for every adjacent pair of permutations σ'_j, $P'(\alpha)$ contains exactly the same permutations as $P(\alpha)$ in the same order regardless of the assignment α.[10] ◀

Our next observation is that the layers of the form $\langle 0, \sigma_j\rangle$ are only a convenience and are not necessary to our definition. Let P be our program for f and let $\sigma_1 \dots \sigma_k$ be the permutations corresponding to instructions π_j where $i_j = 0$. By our restriction that $f(0 \dots 0) = 0$ we get

[10] This argument actually allows us to move π_{j^*} to any spot in the program we want, but we are content with just moving them to the end, for reasons which will become immediately clear.

$\sigma_1 \dots \sigma_k = P(0 \dots 0) = id$, and by Lemma 20 we can move the instructions π_j with $i_j = 0$ to the end of the program, in order, at which point we can simply remove them all using Lemma 18 as they compose to the identity for any input α.

We can also generalize Lemma 20 for *restrictions* of the function f, meaning when we fix the values of some variables and consider the function on the remaining variables. This is simply the observation that fixing variable x_i turns all instructions of the form $\langle i, \sigma_j \rangle$ into fixed layers $\langle 0, \sigma_j \rangle$.

▶ **Corollary 21.** *Let $\rho \in \{0,1,*\}^n$ and let f_ρ be the function f with x_i fixed to $\rho(i)$ wherever $\rho(i) \neq *$. Let $P = \pi_1 \dots \pi_\ell$ be a permutation branching program computing f, let $\pi_j = \langle i_j, \sigma_j \rangle$ for all j, and let $j^* \in [s]$ be such that $i_{j^*} = 0$ or $\rho(i_{j^*}) \neq *$. Then there exists a permutation branching program $P' = \pi'_1 \dots \pi'_{j^*-1}\pi'_{j^*+1} \dots \pi'_\ell \pi_{j^*}$ computing f_ρ, where $\pi'_j = \langle i'_j, \sigma'_j \rangle$ for some permutation σ'_j and $i'_j = i_j$ iff $\rho(i_j) = *$ and 0 otherwise.*

Proof. Let program P'' be the result of replacing i_j with 0 in each instruction $\pi_j \in P$ such that $\rho(i_j) \neq *$. Clearly this program computes f_ρ, and so applying Lemma 20 to P'' completes the proof. ◀

Assuming that $f_\rho(0 \dots 0) = 0$, by our previous observation this allows us to remove all layers that read variables fixed by ρ. We also note that the other three lemmas hold for f_ρ with no changes.

Finally, when f is the AND function, we can *rotate* our program by moving the first instruction to the end as many times as we like:

▶ **Lemma 22.** *Let $P = \Pi_1, \Pi_2$ be a permutation branching program computing AND$(x_1, \dots, x_n)$, where Π_1 and Π_2 are sequences of instructions. Then $P' = \Pi_2, \Pi_1$ also computes AND$(x_1, \dots, x_n)$.*

Proof. Let $\sigma^* = P'(1, \dots, 1)$ (that is, P' applied to the all-ones input). Referring to Definition 17, to prove that P' computes AND, we must show (a) that $\sigma^* \neq id$, (b) that $P'(\alpha) = id$ whenever AND$(\alpha) = 0$, and (c) that $P'(\alpha) = \sigma^*$ whenever AND$(\alpha) = 1$.

(c) follows from the fact that the only string α satisfying AND$(\alpha) = 1$ is the all-ones string.

Now, let $\alpha \in \{0,1\}^n$. Note that $P(\alpha) = \Pi_1(\alpha)\Pi_2(\alpha)$. If AND$(\alpha) = 0$, then $P(\alpha) = id$, so $\Pi_1(\alpha) = (\Pi_2(\alpha))^{-1}$, and so $P'(\alpha) = id$ as well: this proves (b).

Finally, note that $P(1, \dots, 1) \neq id$, so $\Pi_1(1, \dots, 1) \neq (\Pi_2(1, \dots, 1))^{-1}$, and so $\sigma^* = P'(1, \dots, 1) \neq id$, proving (a). ◀

4.2 Upper bounds

For our main upper bound, we modify our algorithm recreating the result of [11] (and analogously [12]) to have length $4n - 4$. In particular, our program will read all but two variables four times, while the last two variables will be read twice.

Proof of Theorem 3. First, we make an easy change to Theorem 9 which allows us to achieve $4n - 3$. Observe that in ToggleInput the order in which we add the inputs is irrelevant, and so consider ToggleMonomials where we reverse the order of toggling on Line 6. Then notice that the last query on Line 2 and the first query on Line 6 are both made to x_n, and so we can merge these two layers along with our entire for loop (which reads no variables) into a single layer querying x_n. Moving to Algorithm 1, this means we only query x_n twice, and furthermore the last query on Line 1 and the first query on Line 3 are both made to x_1, and so again by merging these two queries along with Line 2 we only query x_1 three times.

Now we will change our program so that x_1 is only read twice. Consider two new functions obtained by fixing the value of x_1, namely $f^0 = f(0, x_2 \dots x_n)$ and $f^1 = f(1, x_2 \dots x_n)$. Recall that we used the following polynomial to compute f, where $y_i = \tau_i^{\text{in}} + x_i$:

$$q_f = \sum_{S,S' \subseteq [n]} c_{S,S'} \left(\prod_{i \in S} \tau_i^{\text{in}} \right) \left(\prod_{i \in S'} y_i \right)$$

If we choose $b \in \{0, 1\}$ and fix $\tau_1^{\text{in}} = 0$ and $y_1 = x_1 = b$, we get the following, which can be used to compute (since it is equal to) f^b:

$$q_{f^b} = \sum_{S,S' \subseteq [2..n]} (c_{S,S'} + b \cdot c_{S,S' \cup \{1\}}) \left(\prod_{i \in S} \tau_i^{\text{in}} \right) \left(\prod_{i \in S'} y_i \right)$$

We will use Algorithm 1 to compute q_{f^b}, where $b = x_1$, by removing all reference to x_1 from ToggleInput and ToggleMonomials, and querying x_1 whenever we execute Lines 2 or 4 to determine whether to compute q_{f^0} or q_{f^1} in place of q_f. More specifically, ToggleInput will now only loop over $i = 2 \dots n$, while ToggleMonomials will now only loop over $S \subseteq [2..n]$. Finally in Algorithm 1 we change Lines 2 and 4 to

$$R^{\text{out}} \leftarrow R^{\text{out}} \pm \sum_{S,S' \subseteq [2..n]} (c_{S,S'} + x_1 \cdot c_{S,S' \cup \{1\}}) \left(\prod_{i \in S} R_i^{\text{in}} \right) R_{S'}$$

where Line 2 uses $+$ and Line 4 uses $-$. Note that to execute these lines correctly, we will query x_1 and perform the corresponding instruction; thus we no longer ignore these two lines in calculating our program length.

By our earlier definition of q_{f^b}, this exactly computes q_{f^b} for $x_1 = b$ as claimed. As above we will reverse the order of the queries in ToggleInput the second time it is called in ToggleMonomials, which allows us to read x_n only once per execution for a total of two reads. x_1 will be queried in Lines 2 and 4, and all other variables will be queried four times.

Since we no longer use register R_1, or R_S when $1 \in S$, the number of registers is reduced from the original $n + 2^n$ in Theorem 9 to $n - 1 + 2^{n-1}$. The output register R^{out} is now catalytic, so we add one register for a total $m = 2^{n+2^{n-1}}$. ◀

▶ Note 23. This strategy also allows us to save an exponential number of registers, as we only need a register R_S for each $S \subseteq [2..n]$. While it may be tempting to extend this trick to more variables, say by fixing the values of both x_1 and x_2, the fact that Lines 2 and 4 depend on the value(s) of the fixed variable(s) means that we will have to store at least one of these values in a non-catalytic register, which will add to our width and take us out of the realm of permutation programs. If we go back to m-catalytic branching programs, this gives us another way to save over [11, 12], but with worse parameters; for any $k \in [n]$ by fixing k values we can get a program of length $2(k + 1) + 4(n - k - 1)$ and amortized size $2^k \cdot O(n)$ as before, but for $m = 2^{2^{n-k}-1}$ instead of $2^{2^{n/k}-1}$.

There are two known cases in which we can achieve better than read-4 for AND: $n = 2, 3$. The $n = 2$ case is unsurprising, as our argument allows for two variables to be read twice; it has appeared in many previous works (see c.f. [2]). The case of $n = 3$ is more surprising, and suggests that read-3 may be achievable in general. Note that because of the small values of n involved, neither result gives a program smaller than length $4n - 4$.

▶ **Lemma 24.** *There is a read-2 permutation branching program of width* 3 *computing* $AND(x_1, x_2)$.

Proof. Choose any two permutations σ_1 and σ_2 such that $\sigma_1\sigma_2\sigma_1^{-1}\sigma_2^{-1} \neq id$; for example we can choose $\sigma_1 = (12)$ and $\sigma_2 = (23)$. Then consider the following program:

$$\langle 1, \sigma_1\rangle, \langle 2, \sigma_2\rangle, \langle 1, \sigma_1^{-1}\rangle, \langle 2, \sigma_2^{-1}\rangle$$

By definition of σ_1 and σ_2, $P(1,1) \neq id$, and if either variable is set to 0 then the only permutations left are σ_j and σ_j^{-1} for some $j \in \{1,2\}$, and the composition of these permutations is id. ◀

▶ **Lemma 25.** *There is a read-3 permutation branching program of width* 3 *computing* $AND(x_1, x_2, x_3)$.

Proof. We state the program and leave the reader to check correctness.[11] Our permutations σ_j are given in cycle notation.

$$\begin{aligned}&\langle 1, (23)\rangle, \langle 2, (12)\rangle, \langle 3, (123)\rangle,\\ &\langle 1, (12)\rangle, \langle 2, (13)\rangle, \langle 3, (23)\rangle,\\ &\langle 1, (132)\rangle, \langle 2, (132)\rangle, \langle 3, (13)\rangle\end{aligned}$$ ◀

▶ Note 26. We could also consider a stronger model of permutation branching programs where we only require that $P(\alpha) \neq id$ whenever $f(\alpha) = 1$, instead of requiring $P(\alpha)$ always equal the same permutation when $f(\alpha) = 1$. This is the model used by e.g. [2]. In this case, it is not hard to show that for any n, if we can compute AND$(x_1 \dots x_n)$ in length ℓ, we can also compute any function $f(x_1 \dots x_n)$ in length ℓ by "tensoring" the permutations in P with themselves for each $\alpha \in f^{-1}(1)$. Our lower bounds in the following section will still hold against this model.

4.3 Lower bounds

In this section we show that if one tries to get a program of length less than $3n$, one cannot beat Theorem 3.

Proof of Theorem 4. Let $P = \pi_1\pi_2 \dots \pi_s$ be any program computing AND$(x_1, \dots, x_n)$. We will write σ_i^j to refer to the permutation in the jth instruction in P that reads variable x_i; in other words, the instructions corresponding to x_i will be $\langle i, \sigma_i^1\rangle \dots \langle i, \sigma_i^k\rangle$ for some k.

▷ Claim 27. Any program P computing AND of more than one variable must read every variable at least twice

Proof. Assume that some variable x_i is read only once in P. Then setting $x_{i'} = 0$ for all $i' \neq i$, we get $\sigma_i^1 = P(0, \dots, 0, 1, 0, \dots, 0) = id$. Therefore P acts identically whether x_i is 0 or 1, which is a contradiction because AND depends on x_1. ◁

[11] It should be noted that we found this program through an automated search, and it would be interesting to see what nice properties of the program – of which there are many candidates – could be useful in searching for read-3 programs for higher n.

Now consider when some variable x_i is read exactly twice. Then P has the form:

$$\Sigma_1, \langle i, \sigma_i^1 \rangle, \Pi_1, \langle i, \sigma_i^2 \rangle, \Sigma_2$$

Rotate this program to produce

$$P' = \langle i, \sigma_i^1 \rangle, \Pi_1, \langle i, \sigma_i^2 \rangle, \Pi_2$$

where $\Pi_2 = \Sigma_2, \Sigma_1$. By Lemma 22, P' also computes AND.

The following is our main claim.

▷ Claim 28. Every variable besides x_i is read at least once in Π_1, and there is at most one such variable $x_{i'}$ which is not read at least twice in Π_1.

The same holds for Π_2.

Proof. First, assume for contradiction that there exists $i' \neq i$ such that $x_{i'}$ does not appear in Π_1. Then if we fix $x_{i''} = 1$ for all $i'' \neq i, i'$, we can apply Corollary 21 to move all instructions querying variables other than x_i and $x_{i'}$ to the end of the program, to get an equivalent program of the following form which computes AND$(x_i, x_{i'})$:

$$\langle i, {\sigma'}_i^1 \rangle, \langle i, {\sigma'}_i^2 \rangle, \Sigma$$

where Σ is a sequence of instructions which do not query x_i. Applying Lemma 18, we can replace $\langle i, {\sigma'}_i^1 \rangle, \langle i, {\sigma'}_i^2 \rangle$ with a single instruction $\langle i, \sigma'' \rangle$ to we get an equivalent program

$$\langle i, \sigma''_i \rangle, \Sigma$$

which also computes AND$(x_i, x_{i'})$, which contradicts Claim 27 as i is only read once.

Next, assume for contradiction that there exist $i' \neq i'' \neq i$ such that i' and i'' appear only once each in Π_1. If we fix $x_{i'''} = 1$ for all $i''' \neq i, i', i''$, applying Lemmas 21 and 18, without loss of generality the following program computes AND$(x_i, x_{i'})$:

$$\langle i, {\sigma'}_i^1 \rangle, \langle i', \sigma_{i'} \rangle, \langle i'', \sigma_{i''} \rangle, \langle i, {\sigma'}_i^2 \rangle, \Sigma$$

where $\sigma_{i'}$ and $\sigma_{i''}$ are some permutations and Σ is a set of instructions reading only the variables $x_{i'}$ and $x_{i''}$.

Define $\Sigma_{i'}$ to be the result of fixing $x_{i''} = 0$ in Σ, and define $\Sigma_{i''}$ to be the result of fixing $x_{i'} = 0$ in Σ. Note that if only one remaining variable is set to 1 then the program must output 0, so ${\sigma'}_i^2 = ({\sigma'}_i^1)^{-1}$, $\Sigma_{i'} = (\sigma_{i'})^{-1}$, and $\Sigma_{i''} = (\sigma_{i''})^{-1}$. Thus if we set $x_{i''} = 0$ our resulting program is

$$\langle i, {\sigma'}_i^1 \rangle, \langle i', \sigma_{i'} \rangle, \langle i, ({\sigma'}_i^1)^{-1} \rangle, \langle i', (\sigma_{i'}^1)^{-1} \rangle$$

and so setting $x_i = x_{i'} = 1$ we get that ${\sigma'}_i^1 \sigma_{i'} ({\sigma'}_i^1)^{-1} (\sigma_{i'})^{-1} = id$. Therefore by Lemma 19 we can swap the order of these two instructions and get an equivalent program for AND$(x_i, x_{i'}, x_{i''})$ of the form

$$\langle i', \sigma_{i'} \rangle, \langle i, {\sigma'}_i^1 \rangle, \langle i'', \sigma_{i''} \rangle, \langle i, ({\sigma'}_i^1)^{-1} \rangle, \Sigma$$

and similarly by fixing $x_{i'} = 0$ we have ${\sigma'}_i^1 \sigma_{i''} ({\sigma'}_i^1)^{-1} (\sigma_{i''})^{-1} = id$, which by Lemma 19 leaves us with the program

$$\langle i', \sigma_{i'} \rangle, \langle i'', \sigma_{i''} \rangle, \langle i, {\sigma'}_i^1 \rangle, \langle i, ({\sigma'}_i^1)^{-1} \rangle, \Sigma$$

and applying Lemma 18 on our two layers reading i gives us a program which never reads x_i, which is a contradiction.

Finally, to prove the claim for Π_2, observe that by Lemma 22,

$$P'' = \langle i, \sigma_i^2 \rangle, \Pi_2, \langle i, \sigma_i^1 \rangle, \Pi_1$$

also computes AND, and apply the above argument to P'', with Π_2 playing the role of Π_1. ◁

By a simple analysis of Claim 28, one of two cases must occur for the variables besides x_i: either 1) one variable $x_{i'}$ is read at least twice and all other variables are read at least four times or 2) two variables $x_{i'}, x_{i''}$ are read at least three times and all other variables are read at least four times. This is because a read-2 variable can only be read at most once in each of Π_1 and Π_2, while a read-3 variable will be read at most once in either Π_1 or Π_2. In either of these cases, our branching program must have length at least $4(n-2) + 2 \cdot 2 = 4(n-3) + 2 \cdot 1 + 3 \cdot 2 = 4n - 4$. ◀

▶ Note 29. Besides the fact that our lower bound in Theorem 4 quantitatively matches up with our upper bound in Theorem 3 in the case of reading any variable twice, qualitatively both cases in the analysis at the end of our lower bound proof match with a possible construction given by our upper bound. After fixing a read-2 variable to condition f on, we get two halves to our top level program, and in each of them we will merge two reads of a variable in order to save a further layer. The choice of which variable to merge the reads of is arbitrary, so consider our choices for the first and second half. If we choose the same variable for both halves, it will be read twice and all other variables will be read four times. If we choose different variables in each half, both will be read three times and the rest will be read four times.

5 Open Problems

The most obvious problem left open by our work is to figure out, for an arbitrary function f, the optimal value of m under which we can still achieve linear amortized size. In the upper bounds direction, any improved transparent register program for computing the polynomial q_f could potentially give an upper bound of $2^{2^{o(n)}}$ on m. In the other direction, nothing is known besides the basic counting argument ($m \geq 2^n/O(n)$), and even getting a lower bound of $m \geq 2^n$ for some function f could shed some light on where the correct answer should lie.

In terms of connecting uniform and non-uniform models of space, $\mathsf{L}/\operatorname{poly}$ is equivalent to the class of problems solvable by poly n-size branching programs. However, this gets trickier for *catalytic logspace* (CL), as the corresponding object for $\mathsf{CL}/\operatorname{poly}$ would be m-catalytic branching programs of amortized size $\operatorname{poly}(n)$ for $m = 2^{\operatorname{poly}(n)}$, which has exponential size and thus cannot be written down in polynomial advice. It would be very interesting to understand the connection between such m-catalytic branching programs and $\mathsf{CL}/\operatorname{poly}$, as this would immediately give lower bounds on m for random functions.

Also of interest would be to study the same question for other restricted classes of functions. For example, it is possible that our result for VP could be extended to VNP, although such a result would presumably need to use non-uniformity in a stronger way lest we accidentally prove that uniform VNP is contained in CL – and by extension ZPP [4].

Finally, closing the gap between $3n$ and $4n-4$ on the upper and lower bounds for the optimal length of permutation branching programs seems within reach. For example, a cursory machine search gave no read-3 permutation branching programs for AND on four variables, and if we could formally verify this then it would immediately lead to fully closing the gap at $4n-4$.

References

1 Eric Allender, Anna Gál, and Ian Mertz. Dual VP classes. *Comput. Complex.*, 26(3):583–625, 2017.

2 David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc^1. *J. Comput. Syst. Sci.*, 38(1):150–164, 1989.

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

4 Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 857–866. ACM, 2014.

5 James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 752–760. ACM, 2020.

6 James Cook and Ian Mertz. Encodings and the tree evaluation problem. *Electron. Colloquium Comput. Complex.*, page 54, 2021. URL: `https://eccc.weizmann.ac.il/report/2021/054`.

7 Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Trans. Comput. Theory*, 3(2):4:1–4:43, 2012.

8 Vincent Girard, Michal Koucky, and Pierre McKenzie. Nonuniform catalytic space and the direct sum for space. *Electronic Colloquium on Computational Complexity (ECCC)*, 138, 2015.

9 Frank Gray. Pulse code communication. `https://patents.google.com/patent/US2632058A/en`, 1953. US Patent 2632058A.

10 E.I. Nečiporuk. A boolean function. *Dokl. Akad. Nauk SSSR*, 169(4), 1966.

11 Aaron Potechin. A note on amortized branching program complexity, 2017.

12 Robert Robere and Jeroen Zuiddam. Amortized circuit complexity, formal complexity measures, and catalytic algorithms. In *FOCS*, pages 759–769. IEEE, 2021.