Catalytic Approaches to the Tree Evaluation Problem

James Cook
University of Toronto
Canada
jcook@cs.berkeley.edu

Ian Mertz
University of Toronto
Canada
mertz@cs.toronto.edu

ABSTRACT

The study of branching programs for the Tree Evaluation Problem (TreeEval), introduced by S. Cook et al. (TOCT 2012), remains one of the most promising approaches to separating L from P. Given a label in $[k]$ at each leaf of a complete binary tree and an explicit function in $[k]^2 \to [k]$ for recursively computing the value of each internal node from its children, the problem is to compute the value at the root node. (While the original problem allows an arbitrary-degree tree, we focus on binary trees.) The problem is parameterized by the alphabet size k and the height h of the tree. A branching program implementing the straightforward recursive algorithm uses $\Theta((k+1)^h)$ states, organized into $2^h - 1$ layers of width up to k^h. Until now no better deterministic algorithm was known.

We present a series of three new algorithms solving TreeEval. They are inspired by the work of Buhrman et al. on *catalytic space* (STOC 2012), applied outside the catalytic-space setting. First we give a novel branching program with $2^{4h}\,\mathrm{poly}(k)$ layers of width 2^{3k}, which beats the straightforward algorithm when $h = \omega(k/\log k)$. Next we give a branching program with $k^{2h}\,\mathrm{poly}(k)$ layers of width k^3. This has total size comparable to the straightforward algorithm, but is implemented using the catalytic framework. Finally we interpolate between the two algorithms to give a branching program with $(O(\frac{k}{h}))^{2h}\,\mathrm{poly}(k)$ layers of width $(O(\frac{k}{h}))^{\epsilon h}$ for any constant $\epsilon > 0$, which beats the straightforward algorithm for all $h \geq k^{1/2+\mathrm{poly}\,\epsilon}$. These are the first deterministic branching programs to beat the straightforward algorithm, but more importantly this is the first non-trivial approach to proving deterministic upper bounds for TreeEval.

We also contribute new machinery to the catalytic computing program, which may be of independent interest to some readers.

CCS CONCEPTS

• **Theory of computation → Computational complexity and cryptography**; *Design and analysis of algorithms.*

KEYWORDS

complexity theory, branching programs, catalytic computing, tree evaluation problem

ACM Reference Format:
James Cook and Ian Mertz. 2020. Catalytic Approaches to the Tree Evaluation Problem. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3357713.3384316

1 INTRODUCTION

The deterministic time and space classes, such as L, P, PSPACE, EXP, and EXPSPACE are fundamental to complexity theory. While the containments $\mathsf{SPACE}(k) \subseteq \mathsf{TIME}(2^k)$ and $\mathsf{TIME}(k) \subseteq \mathsf{SPACE}(k)$ are exercises that would show up in a first complexity course, figuring out whether these containments are strict or not has proved to be one of the greatest challenges in the field. As an example, one way to separate P from PSPACE would be to separate P from NP, but determining whether P = NP has remained unsolved for fifty years. The Tree Evaluation Problem [7] has emerged over the past ten years as a candidate for separating L from P.

1.1 The Tree Evaluation Problem and L vs. P

Definition 1 (Tree Evaluation Problem [7])**.** The *tree evaluation problem* $\mathsf{TreeEval}_{h,k}$ is parameterized by a height h and an alphabet size k. The input is a full binary tree of height h, where every leaf is labeled with an element of $[k]$ and every internal node is labeled with a function from $[k] \times [k]$ to $[k]$. The output is the value of the root of the tree, where the tree is evaluated bottom-up in the natural way. We will often omit the subscripts and write TreeEval.

(In the original statement of the problem [7], the degree of the internal nodes in the tree is an additional parameter d. Here, we focus exclusively on binary trees ($d = 2$).)

The input to $\mathsf{TreeEval}_{h,k}$ has size $(2^{h-1}-1)k^2 \log k + 2^{h-1} \log k = O(2^h\,\mathrm{poly}(k))$. The problem is in P: it can be solved in polynomial time by evaluating every node, starting from the leaves, in an order that ensures a node's two children get evaluated before its parent.

However, it is not a log-space algorithm. The space used depends on the order in which the nodes are evaluated, since child values can be forgotten once a parent is evaluated. An argument based on a "pebbling game" [7, 14] shows that even the most space-efficient version of the algorithm must at some point simultaneously store h values, requiring space $\Omega(h \log k) \subseteq \omega(h + \log k)$ for non-constant h, k.

We call this algorithm the *pebbling algorithm.* (Specifically, the most efficient version of the algorithm, using $\Theta(h \log k)$ space.)

1.2 Branching Programs and Lower Bounds

In order to prove space lower bounds for TreeEval, a natural model which previous work has focused on is the *branching program* model, where space is represented by the (logarithm of the) size of the program. The pebbling algorithm described in the previous section can be translated into a branching program whose states are arranged into $2^h - 1$ layers, corresponding to the order in which the nodes are evaluated. The number of states in a layer varies depending on how many values the algorithm must remember at the corresponding point in its execution, but the pebbling-based lower bound shows that at least one layer will always have at least k^h nodes. We say this algorithm has *length* $2^h - 1$ and *width* k^h. The *size* of a branching program is the number of states; a careful analysis shows this one has size $\Theta((k+1)^h)$. (This is equivalent to $\Theta(k^h)$ so long as $h = O(k)$).)

While no unconditional lower bounds are known, a tight $\Omega(k^h)$ lower bound is known for a number of natural restrictions. In the *read-once* restriction the branching program only looks at each bit of the input at most once, while in the *thrifty* restriction the branching program must read only bits corresponding to the actual evaluation of the tree may be read (so for example if the children of a node v evaluate to x and y, the branching program must not read any values of the function at v other than the value at (x, y)). The pebbling algorithm fulfills both of these conditions, but either one of them is enough to guarantee a lower bound of $\Omega(k^h)$ [9][7], and neither of these restrictions assume any other structure on the branching program such as being layered.

1.3 Catalytic Computing

The *catalytic computing* framework of [3], which came out of a fascinating line of work [1, 2] on branching programs and circuits, proposes a novel way to use space in a more efficient way when computing circuits with simple invertible operations. The idea is deceptively simple: assume that we have a small amount of clean work space but an exponentially larger amount of "catalytic space", which is free to use but is full of junk bits that have to be returned to their original configuration at the end of the computation. Since we have no assumptions on the bits in the catalytic space it would seem like it can't help us compute anything, but Buhrman et al. [3] show that if we are working with mathematical instructions that are invertible, this invertibility can help us in two ways: first, by letting us use the space in a way that can be easily reset at the end of the computation, and second, by cleverly cancelling out the "noise" that the bits in the catalytic space introduce into the computation by inverting the computation and then subtracting off the contribution of the noise.

While there has been a flurry of work [4, 5, 8, 11, 15] following the definition of catalytic computing in [3] (see e.g. [12] for a survey of early results), the preliminary results of [1, 2] solved a slightly different type of problem. The catalytic computing model involves having a small clean work tape and exponentially more "catalytic space", but [1] and [2] study what can be done by constantly reusing a small (even constant size) work tape. Since we are looking to rule out logspace algorithms for TreeEval, it is this latter approach which seems more immediately applicable.

1.4 Our Results

In this work we show how the catalytic computing framework can be applied to the tree evaluation problem to give novel algorithms, even for the simple model of layered branching programs. We present the following three programs (recall that the layered branching program for the pebbling algorithm has length $2^h \operatorname{poly}(k)$ and width k^h)

Theorem 1 (One-hot algorithm). *There exists a layered branching program solving* $\mathsf{TreeEval}_{h,k}$ *with length at most* $4^h \operatorname{poly}(k)$ *and width* 2^{3k}.

Theorem 2 (Binary algorithm). *There exists a layered branching program solving* $\mathsf{TreeEval}_{h,k}$ *with length at most* $(2k)^{2h} \operatorname{poly}(k)$ *and width* k^3.

Theorem 3 (Hybrid algorithm). *For every* $\epsilon > 0$ *there exists a* $C = O(1/\epsilon)$ *and a branching program solving* $\mathsf{TreeEval}_{h,k}$ *with length at most* $(C\frac{k}{h} + 1)^{2h} \operatorname{poly}(k)$ *and width* $(C\frac{k}{h} + 1)^{\epsilon h}$.

While the constants in the exponent mean these algorithms don't beat the pebbling algorithm in all cases, for h large (but still $o(k)$) we do indeed achieve an $o(k^h)$ size branching program.

1.5 Important Ideas

This is the first non-trivial approach to proving upper bounds for TreeEval, and in our opinion it highlights a number of interesting ideas in catalytic computing and branching programs, which we will highlight before going into the body of the paper.

Pebbling games. The previous best-known algorithm for TreeEval was based on a strategy for the *pebbling game* on a complete binary tree.

The optimal strategy for this game is well-understood, leading to a $\Omega(k^h)$ lower bound on the number of states used by any pebbling-based branching program. Every subsequent deterministic lower bound for TreeEval, for generalized classes of branching program beyond pebbling, has arrived at the same quantity $\Omega(k^h)$, effectively by showing how to relate every algorithm back to the pebbling game. From its initial definition in [7] it has been widely believed that pebbling gives the optimal lower bound [13].

Our algorithms defeat this common lower bound by using techniques far removed from pebbling.

Use of algebraic techniques. The main result of [3] is to use catalytic computing to efficiently compute the majority of poly n bits, which they do in an algebraic way by summing the input and then using Fermat's Little Theorem. This relies on an inherent way of turning majority into an algebraic object [16]. In TreeEval, each node is labeled with an arbitrary $[k] \times [k] \to [k]$ function. The encoding we present for each algorithm determines how this function can be interpreted as an algebraic object — for example, the one-hot encoding in §3 allows us to interpret it as a sum over products corresponding to a DNF with at most one AND evaluating to 1. We then apply techniques from [3]—as well as many novel improvements on them—to get our results.

Read-once/thrifty restrictions. Our algorithms avoid the lower bounds in the read-once and thrifty models (§1.2). How do they do this? Simply put, the "catalytic space" approach involves recomputing nodes many many times and checking the evaluation of every possible pair of inputs at every node, which leads them to break the read-once and thrifty restrictions in spectacular fashion. This is the first approach to TreeEval that breaks both restrictions, and furthermore in our opinion it does so in a very natural way.

Space-bounded models. We are working in the model of small total workspace, which in some ways puts our work morally closer to the results of [1, 2] than to [3]. However at the core of our results are extensions of techniques from [3], which gives us new ways to use these techniques designed for the large (catalytic) space regime in the setting where not even the catalytic tape is available. It would be interesting to see how many of our known catalytic techniques can be carried over in this way.

Future improvements. The extensions we prove are not known to be optimal in terms of how much recomputation is needed, which is the only bottleneck in the strength of our results. Thus improving them provides a direct approach to improving our results for TreeEval. In fact, an "optimal generalization" of the lemma in question would give a logspace algorithm for TreeEval, firmly shutting the door on the approach of [7].

2 PRELIMINARIES

While we think of the input to $\mathsf{TreeEval}_{h,k}$ as being of size $2^{h-1}\log k + (2^{h-1}-1)k^2\log k$, for our computation model it will be easier to think of our programs as always reading a whole element of $[k]$ at once.

Definition 2 (One piece of the input)**.** In the below definitions of register programs and branching programs, each instruction or state will be allowed to read one "piece" of the input to $\mathsf{TreeEval}_{h,k}$. A piece of the input is either the value associated with a leaf, or an internal node's function evaluated at one of its k^2 possible inputs.

The input consists of $2^{h-1} + (2^{h-1}-1)k^2$ pieces, each of which is a value in $[k]$.

2.1 Branching Programs

There are different definitions of branching programs. Ours is equivalent to that of S. Cook et al. [7], restricted to the deterministic case. Each state of a branching program reads one piece of the input to $\mathsf{TreeEval}_{h,k}$ (Definition 2): either the single value associated with a leaf, or an internal node's function evaluated at one of the k^2 possible inputs.

Definition 3 (Branching program [7])**.** A deterministic branching program[1] for $\mathsf{TreeEval}_{h,k}$ consists of:

- A set of *states* V, one of which is identified as the starting state.
- A set of output states identified in V, each labelled with a value for the program to output.
- For every non-output state, a piece of the input to query (Definition 2) , and a *transition function* mapping the result of the query (in $[k]$) to the next state.

If the sequence of states induced by an input to $\mathsf{TreeEval}_{h,k}$ ends at an output state (rather than looping infinitely), the program terminates with that output. The *size* of the branching program is $|V|$.

Additionally our programs will be from a restricted model called a *layered* branching program, wherein each state $v \in V$ is associated with a *layer* t, such that if v's transition function maps it to $v' \in V$ on some query, then v' is in layer $t+1$.

2.2 Invertible Programs and Transparent Computation

We describe our algorithms as *register machine programs*, which are described by a set of registers each storing values in some ring R, plus a list of mathematical instructions on updating those registers. Our algorithms for TreeEval use the two-element field $R = \mathbb{F}_2$, but many of our results apply more generally. Each instruction has the form $R_x \leftarrow R_x + \prod_i u_i$ (or later, $R_x \leftarrow R_x + \sum_j \prod_i u_{i,j}$), where R_x is a register and each u_i is either a constant or a register other than R_x. These are similar to the programs used in the catalytic computing work of Buhrman et al. [3]. In particular, every instruction is reversible—the instructions $R_x \leftarrow R_x + (-1)\cdot\prod u_i$ and $R_x \leftarrow R_x + \sum_j (-1)\cdot\prod_i u_{i,j}$ respectively suffice—so we call them *invertible programs*.

We will analyze the behaviour of subroutines by comparing the register values before and after a subroutine runs. Following Buhrman et al. [3], we denote the initial value of register R_i with τ_i and say that a subroutine *transparently computes* value v into register R_i if $R_i = \tau_i + v$ when it finishes. We use similar notation for vectors of registers: $\overrightarrow{\tau_i}$ is the initial value of $\overrightarrow{R_i}$, and transparently computing a vector $\overrightarrow{v}$ means ensuring $\overrightarrow{R_i} = \overrightarrow{\tau_i} + \overrightarrow{v}$.

We depart from [3] by allowing some register indices to depend on the input. For example, if v_x denotes the value of leaf node x in the input to TreeEval, then the instruction $R_{1,v_x} \leftarrow R_{1,v_x} + 1$ increments a coordinate of $\overrightarrow{R_1}$ depending on v_x. To connect register programs to branching programs, we make one restriction of our register programs, which is that each instruction uses at most one piece of the input. We can then transform register programs into branching programs:

Lemma 4. *Suppose P is a register program consisting of $|P|$ instructions using m registers over $\mathbb{F}_2$ that transparently computes $\overrightarrow{v}$ into a vector of registers $\overrightarrow{R_1}$. Then there is a layered branching program B_P of size $1 + |P|2^m + k$ which outputs v. The states of B_P other than the starting and output states are organized into $|P|$ layers of size 2^m.*

Proof. The first layer of B_P consists of a single starting state, and the last consists of k output states. Every other layer corresponds to an instruction in P, and has a state for each of the 2^m possible register configurations. A state reads whichever piece of the input the corresponding instruction uses, and its transition function leads to the state in the following layer corresponding to the new register values.

In order to make B_P output the correct value, initialize all registers to zero by choosing $0 \in {\mathbb{F}_2}^m$ in the first layer as the starting state, and designating each state in the final layer as an output state labelled with the value of $\overrightarrow{R_1}$. □

[1] In this paper we only consider *uniform* branching programs, which are branching programs which can be efficiently constructed. The exact notion of uniformity we use is unimportant, except for noting that any size s branching program we construct for TreeEval can certainly be constructed uniformly in $\log s$ space.

3 ALGORITHM 1: ONE-HOT

We first present our *one-hot* algorithm, named after the encoding scheme it uses.

Definition 4 (One-hot encoding)**.** Let $\vec{v_i} = \{v_{i,x}\}_{x\in[k]}$ be a vector of length k. We say that $\vec{v_i}$ stores the value $x \in [k]$ if $v_{i,x} = 1$ and $v_{i,x'} = 0$ for all $x' \neq x$.

The foundation for Algorithm 1 is a formula for the one-hot encoding of a node in terms of its childrens' encodings. In TreeEval, if p is the parent of nodes ℓ and r, then for all $x \in [k]$, the coordinate $v_{p,x} = [v_p = x]$ of p's one-hot encoding is

$$v_{p,x} = \sum_{(y,z)\in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

To build toward Algorithm 1, in subsection 3.1 we show how to build a reversible program that transparently computes a single product $v_\ell v_r$ given programs that compute each factor, then in subsection 3.2 we extend it to efficiently compute the entire vector $\vec{v_p}$.

3.1 Binary Catalytic Products

The key tool in Theorem 1 and all our algorithms is a modified form of Lemma 4 from [3]. We state and prove it below, using a different program than was originally presented in [3] which will be easier to generalize.

Lemma 5 (Lemma 4, [3])**.** *Let R_ℓ, R_r and R_p be distinct registers. Let P_ℓ be an invertible program which transparently computes v_ℓ into register R_ℓ and leaves the other registers unchanged: in other words,*

$$R_\ell = \tau_\ell + v_\ell$$

$$R_i = \tau_i \quad \forall i \neq \ell.$$

Similarly, let P_r be a program which updates

$$R_r = \tau_r + v_r$$

$$R_i = \tau_i \quad \forall i \neq r$$

Then there exists an invertible program P_p which transparently computes $v_\ell v_r$ into R_p, leaving all other registers unchanged, i.e.

$$R_p = \tau_p + v_\ell v_r$$

$$R_i = \tau_i \quad \forall i \neq p$$

P_p uses only the three registers R_v, R_ℓ, R_p (not counting any space used by the programs P_ℓ and P_r) and makes two calls to P_ℓ and P_r each, plus four basic instructions of the form $R_p \leftarrow R_p \pm R_\ell R_r$.

Proof. Program P_p performs as follows:

```
1: $P_\ell$
2: $R_p \leftarrow R_p - R_\ell R_r$        ▷ $R_p = \tau_p - \tau_\ell\tau_r - v_\ell\tau_r$
3: $P_r$
4: $R_p \leftarrow R_p + R_\ell R_r$        ▷ $R_p = \tau_p + \tau_\ell v_r + v_\ell v_r$
5: $P_\ell^{-1}$
6: $R_p \leftarrow R_p - R_\ell R_r$        ▷ $R_p = \tau_p - \tau_\ell\tau_r + v_\ell v_r$
7: $P_r^{-1}$
8: $R_p \leftarrow R_p + R_\ell R_r$        ▷ $R_p = \tau_p + v_\ell v_r$
```

While correctness is given by the inline comments, we motivate this program intuitively. At some point (specifically in step 4) we add $(\tau_\ell + f_\ell)(\tau_r + f_r)$ to R_p. This yields the terms $\tau_\ell\tau_r + f_\ell\tau_r + \tau_\ell f_r + f_\ell f_r$, where the $f_\ell f_r$ term is ultimately what we want to be added to R_p. To cancel out all other terms, we use P_ℓ, P_ℓ^{-1}, P_r, and P_r^{-1} to isolate each term in succession, noting that the only spurious term that will come up is $\tau_\ell\tau_r$. All other registers were reset because they each have one forward program and one inverse program.

The number of recursive calls and basic instructions is clear, and the program P_p can be inverted by running it in reverse order, changing all + operations to − operations and vice-versa, and switching P calls with P^{-1} calls for all recursive calls. □

3.2 Parallel Binary Catalytic Products

Our algorithm for Theorem 1 will use one-hot encodings, so we will need to adapt Lemma 5 to work with vectors of registers. Thus in place of R_p, R_ℓ, and R_r, we will instead have vectors $\vec{R_p}$, $\vec{R_\ell}$, and $\vec{R_r}$, and for convenience we treat each $R_{i,x}$ as being an element in $\mathbb{F}_2$ (so + and − are both equivalent to a bitflip).

When we execute the program P_ℓ (P_r), this will flip exactly one register in $\vec{R_\ell}$ (exactly one register in $\vec{R_r}$), corresponding to the value of node ℓ (the value of node r, respectively). Our goal will be to do the same for v, i.e. add the function vector $\vec{f_v}$ to the register vector $\vec{R_v}$, where $f_{v,i}$ will be 1 if the value of the function computed at node v is i and 0 otherwise.

The key subroutine will be a version of Lemma 5 where ℓ and r are the left and right children of p. The value of $v_{x,i}$ will be 1 if the value of the function computed at node x is i, and 0 otherwise. Note that while we specialize the following lemma to the parameters of our TreeEval instance, this can easily be adapted as a generalization of Lemma 5.

Lemma 6 (Lemma 5, parallel sum version)**.** *Let $\vec{R_\ell}$, $\vec{R_r}$ and $\vec{R_p}$ be distinct k-dimensional vectors of registers. Let $v_{\ell,x} = 1$ iff x is the value of node ℓ, and let P_ℓ be a program which transparently computes $\vec{v_\ell}$ into register $\vec{R_\ell}$ and leaves the other registers unchanged: in other words,*

$$\vec{R_\ell} = \vec{\tau_\ell} + \vec{v_\ell}$$
$$\vec{R_i} = \vec{\tau_i} \quad \forall i \neq \ell$$

Similarly, let P_r be a program which updates

$$\vec{R_r} = \vec{\tau_r} + \vec{v_r}$$
$$\vec{R_i} = \vec{\tau_i} \quad \forall i \neq r$$

where $v_{r,x} = 1$ iff r evaluates to x. Then there exists a program P_p which updates

$$\vec{R_p} = \vec{\tau_p} + \vec{v_p}$$
$$\vec{R_i} = \vec{\tau_i} \quad \forall i \neq p.$$

P_p uses only the $3k$ registers $\vec{R_p}, \vec{R_\ell}, \vec{R_r}$ (not counting any space used by the programs P_ℓ and P_r). P_p uses two calls to P_ℓ and P_r each, plus $4k^2$ basic instructions of the form $R_{p,f_p(y,z)} \leftarrow R_{p,f_p(y,z)} \pm R_{\ell,y}R_{r,z}$, where f_p is the function associated with node p of the TreeEval *instance.*

Proof. Program P_p performs as follows (note that we retain the +/− and P/P^{-1} distinctions only to stress the similarity of this program with the one in Lemma 5):

```
1: $P_\ell$
2: for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
3:     $R_{p,x} \leftarrow R_{p,x} - R_{\ell,y}R_{r,z}$
           ▷ $R_{p,x} = \tau_{p,x} - \sum_{(y,z)\in f_p^{-1}(x)}(\tau_{\ell,y}\tau_{r,z} + v_{\ell,y}\tau_{r,z})$
4: end for
5: $P_r$
6: for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
7:     $R_{p,x} \leftarrow R_{p,x} + R_{\ell,y}R_{r,z}$
           ▷ $R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)}(\tau_{\ell,y}v_{r,z} + v_{\ell,y}v_{r,z})$
8: end for
9: $P_\ell^{-1}$
10: for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
11:     $R_{p,x} \leftarrow R_{p,x} - R_{\ell,y}R_{r,z}$
           ▷ $R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)}(-\tau_{\ell,y}\tau_{r,z} + v_{\ell,y}v_{r,z})$
12: end for
13: $P_r^{-1}$
14: for $x$; $(y,z)$ such that $f_p(y,z) = x$ do
15:     $R_{p,x} \leftarrow R_{p,x} + R_{\ell,y}R_{r,z}$
           ▷ $R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)} v_{\ell,y}v_{r,z}$
16: end for
```

The analysis is the same as in Lemma 5, as the instructions for each pair (y,z) can be treated separately since the only instructions are $R_{p,f_p(y,z)} \leftarrow R_{p,f_p(y,z)} \pm R_{\ell,y}R_{r,z}$. Each basic instruction from Lemma 5 is now k^2 basic instructions, exactly one for each (y,z) pair, and so all counts are as claimed. □

Proof of Theorem 1. We show by induction on h that there is an invertible program of length at most $(4^h - 2)k^2$ using $3k$ binary registers which transparently computes the one-hot encoding of the value of the root node of a $\mathsf{TreeEval}_{h,k}$ instance into one set of k registers, leaving the remaining $2k$ registers at their original values. Given such a program P, Lemma 4 shows we can turn it into a layered branching program with $4^h\,\mathrm{poly}(k)$ layers each containing 2^{3k} states. The branching program produced by Lemma 4 outputs a one-hot encoding; by relabelling the output states with their decoded values, we can turn into a program that solves $\mathsf{TreeEval}_{h,k}$.

For the base case $h = 1$, the program only needs to read the value of the single node and flip the single register corresponding to its value, so the program has length $1 \le (4^1 - 2)k^2$. For the inductive step we are given an instance $\mathsf{TreeEval}_{h+1,k}$, and we inductively assume that there exist programs P_ℓ and P_r corresponding to the children ℓ and r of the root p, each of which computes a subinstance of height h.

By Lemma 6, from this we can build a program P_p computing the value at node p which uses $3k$ registers and consists of two calls each to P_ℓ and P_r plus $4k^2$ basic instructions. Thus the total length of the program is at most $(2+2)(4^h - 2)k^2 + 4k^2 \le (4^{h+1} - 2)k^2$ as promised. For the space usage, since the programs P_ℓ and P_r work regardless of the initial state of the $3k$ registers they use and reset everything except the target registers, we will allow both of them as well as P_p to use the *same* set of $3k$ registers, relabeling them as necessary within each program call. □

4 ALGORITHM 2: BINARY

Next is the *binary* algorithm, once again named after its encoding scheme. This algorithm never uses less space than the straightforward "pebbling" algorithm described in subsection 1.1, but it is an important step toward building the "hybrid" algorithm described in section 5. It is worth noting that while it performs slightly worse than pebbling, it does so with very low width.

This algorithm uses a more compact encoding.

Definition 5 (Binary encoding). Let $\vec{v_i} = \{v_{i,b}\}_{b\in[\log k]}$ be a vector of length $\log k$. We say that $\vec{v_i}$ stores the value $x \in [k]$ if $v_{i,b} = x_b$ for all $b \in [\log k]$, where x_b is the bth bit of x when written in binary.

Working with this encoding will require moving from the binary products used by Algorithm 1 to products of fan-in $2\log k$. Following the same structure as section 3, we first show how to compute a product of more than two scalar values (subsection 4.1), then extend it to efficiently compute the entire vector $\vec{v_p}$ (subsection 4.2).

4.1 d-ary Catalytic Products

While Lemma 6 can be thought of as a generalization of Lemma 5 to accomodate sums of binary products, our next lemma will be a generalization to products of more than two variables.

Lemma 7 (Lemma 5, d-ary version). *Let $R_0,\dots,R_d$ be distinct registers, and let $P_1\dots P_d$ be invertible programs where P_i updates*

$$R_i = \tau_i + v_i$$
$$R_j = \tau_j \quad \forall j \ne i$$

Then there exists an invertible program P which updates

$$R_0 = \tau_0 + \prod_i v_i$$
$$R_j = \tau_j \quad \forall j \ne 0$$

P uses only the $d+1$ registers $R_0,\dots,R_d$ (not counting any space used by the programs P_i) and makes at most 2^d calls to each P_i, plus 2^d basic instructions of the form $R_0 \leftarrow R_0 + c\prod_i R_i$ for values c independent of the register/function values.

Proof. We define some shorthand for the sake of presenting P_v. When we say P_S we mean apply all P_i and P_i^{-1} necessary so that $R_i = \tau_i$ for all $i \in S$ and $R_i = \tau_i + v_i$ for all $i \notin S$. In other words for all $i \in S$ such that $R_i = \tau_i + v_i$ we run P_i^{-1} and for all $i \notin S$ such that $R_i = \tau_i$ we run P_i, and leave all other registers untouched. Now program P performs as follows (with c_S left undefined):

```
1: for $S \subseteq [d]$ do
2:     $P_S$
3:     $R_0 \leftarrow R_0 + c_S \prod_{i=1}^{d} R_i$
4: end for
```

At the end of this program,

$$R_0 = \tau_0 + \sum_{S\subseteq[d]} c_S \left(\prod_{i\in S}\tau_i\right)\left(\prod_{i\notin S}\tau_i + v_i\right)$$

We will now choose the coefficients c_S to make that equal $\tau_0 + \prod_{i=1}^{d} v_i$.

Expanding the polynomial, we can rewrite it as $R_0 = \tau_0 + \sum_{S\subseteq[d]} d_S \left(\prod_{i\in S}\tau_i\right)\left(\prod_{i\notin S} v_i\right)$ where $d_S = \sum_{S'\subseteq S} c_{S'}$. Since our goal is to get $R_0 = \tau_0 + 1\cdot\prod_i v_i$, we can restate our goal as: choose coefficients c_S such that $d_\emptyset = 1$ and $d_S = 0$ for all $S \neq \emptyset$.

Since $d_\emptyset = c_\emptyset$, we start by setting $c_\emptyset = 1$. Now this determines the singleton sets $d_{\{i\}} = c_{\{i\}} + c_\emptyset = 0$, namely $c_{\{i\}} = -c_\emptyset$ for all i. We similarly set the remaining c_S values in increasing order of $S \neq \emptyset$ under the partial order $\subseteq$, using the formula:

$$c_S = -\sum_{S'\subsetneq S} c_{S'}$$

This ensures that $d_S = 0$ for $S \neq \emptyset$.

The number of recursive calls to any P_i is at most once per loop iteration for a total of at most 2^d, while there is one basic instruction per S for a total of 2^d. As before the program P can be inverted by running it in reverse order, changing all + operations to − operations and vice-versa, and switching P_i calls with P_i^{-1} calls for all recursive calls. □

It should be noted that the number of calls to each P_i can be improved from at most 2^d to exactly $2^d/d$ by having the loop over all S use a Gray code [10] for order, rather than choosing the order arbitrarily. However, what will be important for our TEP algorithm is that the loop runs exactly 2^d times.

4.2 Parallel d-ary Catalytic Products

We now prove an alternative version of Lemma 6 by replacing the one-hot encoding with the binary encoding (Definition 5). If ℓ and r are children of node p and $\vec{v_\ell}$ and $\vec{v_r}$ are the binary encodings of the values at those nodes, we can compute $\vec{v_p}$ as follows:

$$v_{p,b} = \sum_{(x,y,z)\in[k]^3}[x_b = 1][f_p(y,z) = x]\prod_{b'\in[\log k]}[v_{\ell,b'} = y_{b'}][v_{r,b'} = z_{b'}]$$

where t_b is the bth bit of t written in binary.

Calculating $v_{p,b}$ in this form requires us to compute a product of fan-in $2\log k$ (not counting the constant terms $[x_b = 1]$ and $[f_p(y,z) = x]$) and thus requires considerably heavier machinery than Lemma 6, namely Lemma 7. We also need to be able to compute each bit $v_{\ell,b}$ or $v_{r,b}$ separately in order to cover all subsets of the product $\prod_{b\in[\log k]}[v_{\ell,b} = y_b][v_{r,b} = z_b]$, so we have to formulate our recursive statement a bit differently.

Again note that this is a generalization of all our previous lemmas, and the "most general" version of Lemma 5, up to improvements in the parameters themselves. We discuss these issues in Appendix A.

Lemma 8 (Lemma 5, parallel sum + d-ary version). *Let $\vec{R_\ell}$, $\vec{R_r}$ and $\vec{R_p}$ be distinct $(\log k)$-dimensional vectors of registers. For all $T \subseteq [\log k]$ let $P_\ell(T)$ be a program which updates*

$$R_{\ell,b} = \tau_{\ell,b} + v_{\ell,b} \quad \forall b \in T$$

$$R_{i,b} = \tau_{i,b} \quad \text{when } i \neq \ell \text{ or } b \notin T$$

where $v_{\ell,b} = 1$ iff the binary encoding of the value at node ℓ has a 1 in the bth position. Let $P_r(T)$ be defined similarly. Then for every $T \subseteq [\log k]$ there exists a program $P_p(T)$ which updates

$$R_{p,b} = \tau_{p,b} + v_{p,b} \quad \forall b \in T$$

$$R_{i,b} = \tau_{i,b} \quad \text{when } i \neq p \text{ or } b \notin T$$

P_p uses only the $3\log k$ registers $\vec{R_p}, \vec{R_\ell}, \vec{R_r}$ (not counting any space used by the programs P_ℓ and P_r) and makes k^2 calls to each $P_\ell(T)$ and k^2 calls total to each $P_r(T)$ for a total of $2k^2$ recursive calls, plus $O(k^3 \log k)$ basic instructions.

Proof. We recall our key equation, which we restate as

$$v_{p,b} = \sum_{(x,y,z)\in[k]^3}[x_b = 1][f_p(y,z) = x]\prod_{b'\in[\log k]}(v_{\ell,b'} + \overline{y_{b'}})(v_{r,b'} + \overline{z_{b'}})$$

This is because the indicators $[v_{\ell,b'} = y_{b'}]$ and $[v_{r,b'} = z_{b'}]$ can be replaced by taking the negation of their XOR, which is the same as taking the negation of either one and adding them together mod 2.

Like in the proof of Lemma 7, we say that P_{S_ℓ,S_r} means apply whichever $P(S_\ell')$ and $P(S_r')$ is necessary so that $R_{\ell,b} = \tau_\ell$ for each $b \in S_\ell$, $R_{\ell,b} = \tau_\ell + v_{\ell,b}$ for $b \notin S_\ell$, and similarly for S_r. Now program $P_p(T)$ performs as follows (with c_{S_ℓ,S_r} left undefined):

```
1: for $S_\ell, S_r \subseteq [\log k]$ do
2:     $P_{S_\ell,S_r}$
3:     for $b \in T$; $(x,y,z)$ such that $x_b = 1 \wedge f_p(y,z) = x$ do
4:         $R_{p,b} \leftarrow R_{p,b} + c_{S_\ell,S_r} \prod_{b'\in[\log k]} (R_{\ell,b'} + \overline{y_{b'}}\cdot[b' \notin S_\ell])(R_{r,b'} + \overline{z_{b'}}\cdot[b' \notin S_r])$
5:     end for
6: end for
```

The analysis is the same as in Lemma 6 and Lemma 7, where we think of S_ℓ and S_r as being one large set together, over two disjoint parts of a universe of size $2\log k$. Again we are forced to choose $c_{\emptyset,\emptyset} = 1$ and then for all other S_ℓ, S_r which are not *both* empty

$$c_{S_\ell,S_r} = \sum_{\substack{S_\ell'\subseteq S_\ell\\ S_r'\subseteq S_r\\ (S_\ell',S_r')\neq(S_\ell,S_r)}} -c_{S_\ell',S_r'}$$

Since there are $2^{\log k} = k$ possible sets S_ℓ and S_r, there are k^2 possible pairs of sets, so the number of recursive calls is as claimed. Each line of basic instructions in the loop consists of k^3 basic instructions per $b \in T \subseteq [\log k]$ the number of basic instructions is also as claimed. □

Proof of Theorem 2. We show by induction on h that there is an invertible program of length $(2k)^{2h}\operatorname{poly}(k)$ using $3\log k$ binary registers which transparently computes the binary encoding of the value of the root node of a $\mathsf{TreeEval}_{h,k}$ instance into one set of $\log k$ registers, leaving the remaining $2\log k$ registers at their original values. As in the proof of Theorem 1, we can use Lemma 4 to transform this into a layered branching program with $(2k)^{2h}\operatorname{poly}(k)$ layers each containing k^3 states.

For the base case $h = 1$, the program need only read the value of the single node and flip up to $\log k$ registers corresponding to the binary encoding of its value, so the program has length $\log k \leq 4^0\operatorname{poly}(k)$. For the inductive step we are given a $\mathsf{TreeEval}_{h+1,k}$ instance of height $h+1$, and we inductively assume that there exist programs $P_\ell(S)$ and $P_r(S)$ corresponding to the children ℓ and r of the root p, each of which computes any subset of bits for a subinstance of height h.

By Lemma 7 from this we can build a program $P_p := P_p([\log k])$ solving the function at p using $3\log k$ registers and which makes k^2

recursive calls to P_ℓ functions plus k^2 recursive calls to P_r functions, plus $O(k^3 \log k)$ basic instructions. The total length of the program is at most $2k^2 \cdot (2k)^{2h} \operatorname{poly}(k) + O(k^3 \log k) \leq (2k)^{2(h+1)} \operatorname{poly}(k)$ as promised. For the space usage we again reuse all registers for each recursive call. □

5 ALGORITHM 3: HYBRID

Our final algorithm is the *hybrid* algorithm. As the name suggests it is a synthesis of the two previous approaches: we break our registers into blocks such that each element in $[k]$ falls into only one block (one-hot), and inside the block is identified by a binary encoding (binary).

To prove Theorem 3 we no longer need to generalize Lemma 5 further, as Lemma 8 provides the most general form we need. However as mentioned before we will generalize the encoding to a *hybrid encoding* that interpolates between the one-hot and binary encodings.

Definition 6 (Hybrid encoding). Let $a \in [\log k]$ be fixed, and let $\vec{v_i} = \{v_{i,(A,B)}\}_{(A,B)\in[a]\times[k/(2^a-1)]}$ be a vector of length $a \cdot \frac{k}{2^a-1}$. Intuitively we break $[k]$ into blocks of length $2^a - 1$ so that within a block each element gets a unique non-zero binary encoding of length a. More formally for $x \in [k]$ let $E(x) = (x \mod (2^a - 1)) + 1$ and let $F(x) = \lceil \frac{x}{2^a-1} \rceil$. We say that $\vec{v_i}$ stores the value $x \in [k]$ if $v_{i,(A,B)} = [E(x)_A = 1 \wedge F(x) = B]$ for all $(A, B) \in [a] \times [k/(2^a - 1)]$. Note that for $a = 1$ and $a = \log k$ we get k blocks of size 1 and 1 block of size $\log k$ respectively, which recovers the encodings in Definition 4 and Definition 5.

For all $(A, B) \in [a] \times [k/(2^a - 1)]$ the value $v_{p,(A,B)}$ is given by

$$v_{p,(A,B)} = \sum_{(x,y,z)\in[k]^3} [E(x)_A = 1 \wedge F(x) = B][f_p(y,z) = x] \cdot \prod_{b\in[a]} [v_{\ell,(b,F(y))} = E(y)_b][v_{r,(b,F(z))} = E(z)_b]$$

Note that if the output of ℓ is not y, then all bits $v_{\ell,(b,F(y))}$ are zero, and since $E(y)$ is nonzero the term will be zeroed out (and similarly for the output of r and z).

This is a product of fan-in $2a$, and so Lemma 7 will step in to do the work. However one other important component is that in any term since (y, z) is fixed all $[v_{\ell,(b,F(y))} = E(y)_b]$ factors only read from block $F(y)$ and all $[v_{r,(b,F(z))} = E(z)_b]$ only read from block $F(z)$. Thus instead of running over all subsets of $[a]$ for each block in $[\frac{k}{2^a-1}]$ separately, we can simply run over subsets of $[a]$ and apply them to every block in $[\frac{k}{2^a-1}]$ simultaneously, for a total of 2^{2a} recursive calls for P_ℓ programs and 2^{2a} for P_r programs.

While Theorem 3 gives one setting of parameters chosen to make the total size of the branching program small, in reality this approach gives a whole family of branching programs for TreeEval, in particular subsuming the constructions in Theorem 1 and Theorem 2.

Lemma 9 (Hybrid lemma). *Let $\vec{R_\ell}, \vec{R_r}$ and $\vec{R_p}$ be distinct $(a \times \frac{k}{2^a-1})$-dimensional vectors of registers. For all $T \subseteq [a]$ let $P_\ell(T)$ be a program which updates*

$$R_{\ell,(A,B)} = \tau_{\ell,(A,B)} + v_{\ell,(A,B)} \quad \forall (A,B) \in T \times [\frac{k}{2^a - 1}]$$

$$R_{i,(A,B)} = \tau_{i,(A,B)} \quad \text{when } i \neq \ell \text{ or } A \notin T$$

where $v_{\ell,(A,B)} = 1$ iff the hybrid encoding of the value at node ℓ has a 1 in the (A, B)th position. Define $P_r(T)$ similarly. Then for every $T \subseteq [a]$ there exists a program $P_p(T)$ which updates

$$R_{p,(A,B)} = \tau_{p,(A,B)} + v_{p,(A,B)} \quad \forall (A,B) \in T \times [\frac{k}{2^a - 1}]$$

$$R_{i,(A,B)} = \tau_{i,(A,B)} \quad \text{when } i \neq p \text{ or } A \notin T$$

$P_p(T)$ uses only the $3 \cdot \frac{ak}{2^a-1}$ registers $\vec{R_p}, \vec{R_\ell}, \vec{R_r}$ (not counting any space used by the programs P_ℓ and P_r) and makes $O(2^{2a})$ calls to P_ℓ programs and $O(2^{2a})$ calls to P_r programs, plus $O(k^3 \log k)$ basic instructions.

Proof. We can rewrite our main equation for the hybrid algorithm as

$$v_{p,(A,B)} = \sum_{(x,y,z)\in[k]^3} [E(x)_A = 1 \wedge F(x) = B][f_p(y,z) = x] \cdot \prod_{b\in[a]} (v_{\ell,(b,F(y))} + \overline{E(y)_b})(v_{r,(b,F(z))} + \overline{E(z)_b})$$

This is the same as the equation stated before this lemma, except that as in Lemma 8, we have expressed the indicators $[v_{\ell,(b,F(y))} = E(y)_b]$ and $[v_{r,(b,F(z))} = E(z)_b]$ using negations of XORs. Now program $P_p(T)$ performs as follows (with c_{S_ℓ,S_r} left undefined):

```
1: for $S_\ell, S_r \subseteq [a]$ do
2:     $P_{S_\ell,S_r}$
3:     for $A \in T; B; (x,y,z)$ such that $E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x$ do
4:         $R_{p,(A,B)} \leftarrow R_{p,(A,B)} + c_{S_\ell,S_r} \prod_{b\in[a]} (R_{\ell,(b,F(y))} + \overline{E(y)_b}[b \notin S_\ell])(R_{r,(b,F(z))} + \overline{E(z)_b}[b \notin S_r])$
5:     end for
6: end for
```

Notice that we never multiply bits $R_{\ell(b,B)}$ and $R_{\ell(b',B')}$ for $B \neq B'$, and so our program is able to run the protocol from Lemma 8 "in parallel" for every block $B \in [\frac{k}{2^a-1}]$ in the inner loop, so we only need 2^{2a} recursive calls each to P_ℓ and P_r (one per iteration of the outer loop).

To determine the values of c_{S_ℓ,S_r}, let us compute the final value of $R_{p,(A,B)}$, for an arbitrary $A \in T$ and $B \in [k/(2^a - 1)]$. To do this, we expand the polynomial added on line 4 of the program, and take the sum over all iterations of the loop: that is, all $S_\ell, S_r \subseteq [a]$ and all x, y, z satisfying $E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x$. This produces:

$$\begin{aligned}
&R_{p,(A,B)} \\
&= \tau_{p,(A,B)} + \sum_{\substack{S_\ell,S_r\subseteq[a]\\(x,y,z)\in[k]^3}} [E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x] \cdot \\
&\quad c_{S_\ell,S_r} \Big(\prod_{i\in S_\ell} \tau_{\ell,(b,F(y))}\Big)\Big(\prod_{i\notin S_\ell} (\tau_{\ell,(b,F(y))} + v_{\ell,(b,F(y))} + \overline{E(y)_b})\Big) \cdot \\
&\quad \Big(\prod_{i\in S_r} \tau_{r,(b,F(z))}\Big)\Big(\prod_{i\notin S_r} (\tau_{r,(b,F(z))} + v_{r,(b,F(z))} + \overline{E(z)_b})\Big) \\
&= \tau_{p,(A,B)} + \sum_{(x,y,z)\in[k]^3} [E(x)_A = 1 \wedge F(x) = B \wedge f_p(y,z) = x] \cdot
\end{aligned}$$

$$\sum_{S_\ell,S_r\subseteq[a]} d_{S_\ell,S_r}\Big(\prod_{i\in S_\ell}\tau_{\ell,(b,F(y))}\Big)\Big(\prod_{i\notin S_\ell}(v_{\ell,(b,F(y))}+\overline{E(y)_b})\Big)\cdot \Big(\prod_{i\in S_r}\tau_{r,(b,F(z))}\Big)\Big(\prod_{i\notin S_r}(v_{r,(b,F(z))}+\overline{E(z)_b})\Big)$$

where $d_{S_\ell,S_r}=\sum_{S'_\ell\subseteq S_\ell,S'_r\subseteq S_R} c_{S_\ell,S_r}$. Note that this is essentially the same structure as Lemma 8, with the only differences being the binary flag $[[E(x)_A=1\wedge F(x)=B\wedge f_p(y,z)=x]$ and the exact registers being multiplied together.

As usual, if we can ensure $d_{\emptyset,\emptyset}=1$ and all other coefficients are 0, then the part after $\tau_{p,(A,B)}$ exactly matches our formula for $v_{p,(A,B)}$, and so we have $R_{p,(A,B)}=\tau_{p,(A,B)}+v_{p,(A,B)}$ as required. We start by setting $c_{\emptyset,\emptyset}=1$, and then for all S_ℓ,S_r such that $S_\ell\neq\emptyset\vee S_r\neq\emptyset$,

$$c_{S_\ell,S_r}=\sum_{\substack{S'_\ell\subseteq S_\ell\\ S'_r\subseteq S_r\\ (S'_\ell,S'_r)\neq(S_\ell,S_r)}} -c_{S'_\ell,S'_r}$$

again treating S_ℓ and S_r as being one set over disjoint parts.

Since there are 2^a possible sets S_ℓ and S_r, there are 2^{2a} possible pairs of sets, so the number of recursive calls is as claimed. □

Proof of Theorem 3. By the same induction as in Theorem 1 and Theorem 2, we can use Lemma 9 to build a program $P_p := P_p([a]\times[\frac{k}{2^a-1}])$ which finds the value at node p using $3\cdot\frac{ak}{2^a-1}$ registers and 2^{2a} recursive calls plus $O(k^3\log k)$ basic steps. The total length of the program is at most $2^{2ah}\operatorname{poly}(k)$ and by reusing space the width is $2^{3ak/(2^a-1)}$. For any C choosing $a=\log(C\frac{k}{h}+1)$ we get that

$$2^{2ah}\operatorname{poly}(k)=2^{2h\log(Ck/h+1)}\operatorname{poly}(k)=(C\frac{k}{h}+1)^{2h}\operatorname{poly}(k)$$

$$2^{3ak/(2^a-1)}\leq 2^{(3/C)h\log(Ck/h+1)}=(C\frac{k}{h}+1)^{(3/C)h}$$

and so choosing $C=\frac{3}{\epsilon}$ completes the proof. □

6 CONCLUSION

A reasonable question to ask is if a better version of Lemma 7 might be too much to ask for. Certainly in terms of TreeEval it represents one possible path directly to proving TreeEval $\in$ L. Namely if Lemma 7 can be improved in the following ways:

- make only $O(1)$ calls to each P_i
- all rounds of calls to P_i's can be parallelized as in Lemma 8 such that only $O(1)$ rounds of calls are needed
- these rounds can be parallellized such that $2^{O(d)}$ instances sharing some set of $O(d)$ target registers can be run in the same round, with only $O(d)$ registers being used in total

then it should be possible to run Lemma 8 with length $2^{O(h)}\operatorname{poly}(k)$ and width $\operatorname{poly}(k)$, yielding a logspace algorithm. However it may be that such a lemma would have implications on L itself. As the most immediate avenue to improving our main theorems, studying the feasibility of such an algorithm is our most important open problem. We discuss one potential avenue in Appendix A.

S. Cook et al. [6] offer a prize for any algorithm which, for a fixed h, proves TreeEval$_{h,k}\in O(k^{h-\epsilon})$ for any constant $\epsilon>0$. Note that if $h\geq k^{1/2+\epsilon/4}$ then

$$\begin{aligned}(C\tfrac{k}{h}+1)^{(2+\epsilon)h} &\leq ((C+1)k^{1/2-\epsilon/4})^{(2+\epsilon)h}\\ &= (C'k)^{(2+\epsilon)(1/2-\epsilon/4)h}\\ &\leq k^{(1-\epsilon^2/4+\log C'/\log k)h}\ll k^{h-\epsilon}\end{aligned}$$

and so we far surpass what is required for the prize. However to get an $o(k^h)$ upper bound for *all* h, the catalytic technique seems to inevitably require $3d$ registers for a representation of length d, and so getting more efficient algorithms for succinct representations where $d\ll O(k)$ seems to be a necessary next step for our approach.

As discussed in section 1, our techniques come from and generalize the catalytic computing framework despite being in a small space regime. Understanding the power of catalytic techniques to run many parallel d-ary products in the same space could help us understand the power of using catalytic techniques for small space classes, which could help us understand the power of small space.

ACKNOWLEDGMENTS

The authors would like to thank Stephen Cook and Toniann Pitassi for many helpful discussions leading up to this paper. We would also like to thank the anonymous reviewers for their useful suggestions. Both authors are partially funded by NSERC.

A IMPROVED d-ARY CATALYTIC PRODUCTS

As touched upon before, Lemma 7 gives a d-ary form of Lemma 5, with the main slowdown being the $O(2^d)$ recursive calls to each P_i program. As a greedy next step, we could try to reduce the number of recursive calls, possibly even to match the constant number needed in Lemma 5.

It is worth noting that we actually have such a construction which makes only poly(d) calls to each P_i, a major improvement on Lemma 7. We state and prove this improvement, and then discuss the problems with using the construction presented for improving our main results.

Lemma 10 (Lemma 7, polynomially efficient version). *Let $R_0,\ldots,R_d$ be distinct registers. Let $P_1\ldots P_d$ be a invertible programs where P_i updates*

$$R_i=\tau_i+v_i$$

$$R_j=\tau_j\quad\forall j\neq i.$$

Then there exists an invertible program P which updates

$$R_0=\tau_0+\prod_i v_i$$

$$R_j=\tau_j\quad\forall j\neq 0.$$

P uses the $d+1$ registers $R_0,\ldots,R_d$ plus d additional registers (not counting any space used by the programs P_i) and makes d^2 calls to each P_i plus poly(d) *basic instructions of the form $R_p\leftarrow R_p\pm\prod_i R_i$.*

Proof. We inductively compute $\prod_i v_i$ using Lemma 5 as a subroutine. We will compute the product like a binary tree, at each level j computing products of pairs from level $j-1$. Inductively each register at level j will be the product of 2^j f_i's. For all $j=0\ldots\log d$ let $\overrightarrow{R^j}=\{R^j_i\}$ be a vector of $\frac{d}{2^j}$ registers, where $R^0_i:=R_i$ for all

$i \in [d]$ and $R_1^{\log d} := R_0$. Since $\sum_j \frac{d}{2^j} = 2d$ this gives us $2d$ registers as claimed. Let $P_\ell^0 := P_1, P_3 \ldots$ and $P_r^0 := P_2, P_4 \ldots$, and for $j \in [\log d]$ we inductively define programs P_ℓ^j as follows:

```
1: $P_\ell^{j-1}$
2: for $i = 1, 3 \ldots$ do
3:     $R_i^j \leftarrow R_i^j - R_{2i-1}^{j-1} R_{2i}^{j-1}$
4: end for
5: $P_r^{j-1}$
6: for $i = 1, 3 \ldots$ do
7:     $R_i^j \leftarrow R_i^j + R_{2i-1}^{j-1} R_{2i}^{j-1}$
8: end for
9: $(P_\ell^{j-1})^{-1}$
10: for $i = 1, 3 \ldots$ do
11:     $R_i^j \leftarrow R_i^j - R_{2i-1}^{j-1} R_{2i}^{j-1}$
12: end for
13: $(P_r^{j-1})^{-1}$
14: for $i = 1, 3 \ldots$ do
15:     $R_i^j \leftarrow R_i^j + R_{2i-1}^{j-1} R_{2i}^{j-1}$
16: end for
```

We define the program P_r^j similarly but for even i in each loop instead.

We claim that P_ℓ^j (P_r^j) sets $R_i^j = \tau_i^j + \prod_{i'=2^j(i-1)+1}^{2^j i} v_{i'}$ for all odd (even) i and leaves all other registers untouched, and that it uses at most 4^j calls to each P_i plus $\frac{2}{3}(4^j - 1)d$ basic instructions. This is clear for $j = 0$ as P_ℓ^0 and P_r^0 simply add v_i to all the corresponding odd and even registers respectively using one call to each relevant P_i and no basic instructions. Inductively correctness follows by the correctness of the program for Lemma 5, as for each i our program performs the same steps. Since P_ℓ^{j-1} and P_r^{j-1} each make at most 4^{j-1} calls to each P_i and two calls are made to each of these programs, we get at most $4 \cdot 4^{j-1} = 4^j$ calls to each P_i program. The for loops add $2d$ basic instructions for a total of $4 \cdot \frac{2}{3}(4^{j-1} - 1)d + 2d = \frac{2}{3}(4^j - 1)d$.

Running $P_1^{\log d}$ we thus get $R_0 = R_1^{\log d} = \tau_0 + \prod_{i'=1}^{d} v_{i'}$ as required, with a total of $4^{\log d} = d^2$ calls to each P_i and $O(4^{\log d} d) = \mathrm{poly}(d)$ basic instructions. □

This would seem like major leap for all our results: plugging these numbers into our construction for Theorem 2 would give a branching program with length $(\log k)^{3h}\,\mathrm{poly}(\log k)$, which would go far and beyond the task of beating k^h for every super-constant k and h. Unfortunately, the recursive steps of Algorithms 2 and 3 (specifically, Lemmas 8 and 9) don't just compute one product $\prod_{i=1}^{d} v_i$ — they actually compute a sum involving k^2 different products. Recall that in Lemma 8 our key equation was

$$v_{p,b} = \sum_{(x,y,z) \in [k]\times[k]\times[k]} [x_b = 1][f_p(y,z) = x] \cdot \prod_{b' \in [\log k]} [v_{\ell,b'} = y_{b'}][v_{r,b'} = z_{b'}]$$

Each distinct (y, z) gives rise to a different product, because the y'_b or z'_b values will be distinct. This is no issue for Lemmas 8 and 9, because each different product in the sum is computed directly into the same R_v registers, whereas the tree-like construction in Lemma 10 uses $\log k$ extra registers, denoted R_i^j in the proof. Naïvely, then, computing all k^2 products simultaneously would require $k^2 \log k$ extra registers, at which point Algorithm 1 is a better option. It would be interesting to try to improve on this approach.

REFERENCES

[1] David A Barrington. 1989. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *J. Comput. System Sci.* 38, 1 (1989), 150–164.

[2] Michael Ben-or and Richard Cleve. 1992. Computing Algebraic Formulas Using a Constant Number of Registers. *SIAM J. Comput.* 21, 1 (Feb. 1992), 54–58.

[3] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. 2014. Computing with a full memory: catalytic space. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. ACM, 857–866.

[4] Harry Buhrman, Michal Koucký, Bruno Loff, and Florian Speelman. 2018. Catalytic Space: Non-determinism and Hierarchy. *Theory Comput. Syst.* 62, 1 (2018), 116–135.

[5] Diptarka Chakraborty, Debarati Das, Michal Koucký, and Nitin Saurabh. 2018. Space-Optimal Quasi-Gray Codes with Logarithmic Read Complexity. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland (LIPIcs)*, Vol. 112. 12:1–12:15.

[6] Stephen Cook, Mark Braverman, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. 2009. Branching Programs: Avoiding Barriers. (August 2009). https://www.cs.toronto.edu/~sacook/barriers.ps Talk at Barriers Workshop at Princeton.

[7] Stephen Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. 2012. Pebbles and Branching Programs for Tree Evaluation. *ACM Trans. Comput. Theory* 3, 2, Article 4 (Jan. 2012), 43 pages. https://doi.org/10.1145/2077336.2077337 arXiv version freely available at http://arxiv.org/abs/1005.2642.

[8] Samir Datta, Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. 2020. Randomized and Symmetric Catalytic Computation. *Electronic Colloquium on Computational Complexity (ECCC)* 27 (2020), 24. https://eccc.weizmann.ac.il/report/2020/024

[9] Jeff Edmonds, Venkatesh Medabalimi, and Toniann Pitassi. 2018. Hardness of Function Composition for Semantic Read once Branching Programs. In *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA (LIPIcs)*, Vol. 102. 15:1–15:22.

[10] Frank Gray. 1953. Pulse code communication. https://patents.google.com/patent/US2632058A/en. US Patent 2632058A.

[11] Chetan Gupta, Rahul Jain, Vimal Raj Sharma, and Raghunath Tewari. 2019. Unambiguous Catalytic Computation. *Electronic Colloquium on Computational Complexity (ECCC)* 26 (2019), 95.

[12] Michal Koucký. 2016. Catalytic computation. *Bulletin of the EATCS* 118 (2016).

[13] David Liu. 2013. Pebbling Arguments for Tree Evaluation. *CoRR* (2013).

[14] Michael S. Paterson and Carl E. Hewitt. 1970. Comparative Schematology. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, Jack B. Dennis (Ed.). ACM, New York, NY, USA, 119–127. https://doi.org/10.1145/1344551.1344563

[15] Aaron Potechin. 2016. A Note on Amortized Branching Program Complexity. arXiv:cs.CC/1611.06632

[16] John H. Reif and Stephen R. Tate. 1992. ON THRESHOLD CIRCUITS AND POLYNOMIAL COMPUTATION.