Encodings and the Tree Evaluation Problem

James Cook Ian Mertz
University of Toronto

April 14, 2021

Abstract

We show that the Tree Evaluation Problem with alphabet size k and height h can be solved by branching programs of size $k^{O(h/\log h)} + 2^{O(h)}$. This answers a longstanding challenge of Cook et al. (2009) and gives the first general upper bound since the problem's inception.

Acknowledgements The authors would like to thank Stephen Cook and Toniann Pitassi for many helpful discussions leading up to this paper. The second author was partially funded by NSERC.

1 Introduction

The Tree Evaluation Problem (TreeEval) was introduced as a candidate for a problem that could separate L from P [BCM+09a]. For alphabet size parameter k and height parameter h, the input to $\mathsf{TreeEval}_{k,h}$ is a full binary tree of height h where every leaf is labeled with an element from $[k]$ and each internal node is labeled with a function from $[k] \times [k]$ to $[k]$; the output is defined by evaluating the tree in a bottom-up fashion where each internal node outputs the value of its function with inputs coming from its children.

While this can clearly be done in polynomial time by simply evaluating each node in turn, a pebbling argument shows that this algorithm is required to hold h values from $[k]$ in memory. To this effect the original authors conjectured that for alphabet size k and height h, any deterministic branching program solving $\mathsf{TreeEval}_{k,h}$ requires $\Omega(k^h)$ states [BCM+09b], which would translate to a superlogarithmic space lower bound as long as $k, h = \omega(1)$.

Since then many such lower bounds have been proven for restricted models of branching programs, including read-once and thrifty. On the upper bounds side, the original authors proposed a challenge: give a deterministic branching program that solves $\mathsf{TreeEval}_{k,h}$ with $O(k^{h-\epsilon})$ states for all superconstant values of k and h [CBM+09]. This challenge has stood for over a decade.

In a previous paper [CM20], we showed the challenge can be met when $h \geq k^{1/2+\mathrm{poly}(\epsilon)}$. In this note we show that a small modification to that algorithm fully answers the challenge and disproves the $\Omega(k^h)$ conjecture.

Theorem 1. *Let $k, h \in \mathbb{N}$. Then $\mathsf{TreeEval}_{k,h}$ can be solved by a uniform family of deterministic branching programs which have size $k^{O(h/\log h)}$ if $k \geq h$ and $2^{O(h)}$ if $k \leq h$.*

2 Preliminaries

Definition 1. Let $k, h \in \mathbb{N}$. The *tree evaluation problem* of height h and alphabet size k, denoted $\mathsf{TreeEval}_{k,h}$, is defined as follows. The input is a labeling of the full binary tree of height h, where every leaf is labeled with an element in $[k]$ and each internal node is labeled with a function in $[k] \times [k] \to [k]$, encoded as an explicit $k \times k$ table. Based on this input, we derive a value at each node in the tree in a bottom-up fashion: leaves take their values directly from the input, and each internal node's value is its function applied to its children's values. The output to the TreeEval instance is the value at the root.

By convention, the input to $\mathsf{TreeEval}_{k,h}$ is treated as a sequence of symbols in $[k]$. The input then has length $(2^{h-1}-1)\cdot k^2 + 2^{h-1} = 2^h \operatorname{poly}(k)$.[1]

We now introduce the computation models we will be using to compute $\mathsf{TreeEval}_{k,h}$. We use n to generically refer to the input size of our function f, where for $f = \mathsf{TreeEval}_{k,h}$ we have $n = 2^h \operatorname{poly}(k)$. Our first model is a standard syntactic notion of space-bounded computation.

Definition 2 (Branching program [CMW+12])**.** Let $f : [k]^n \to [k]$ be a function. A *branching program* is a directed acyclic graph G with the following properties:

- There is a single source node v and k sink nodes.
- Every non-sink node is labeled with an input variable x_i for $i \in [n]$ and has k outgoing edges, one for each value in $[k]$
- For every $j \in [k]$ there is one sink node labeled with j.

We say that G computes f if for every $x \in [k]^n$, the path defined by starting at the source and following the edge labeled by the value of the x_i labeling the current node ends at the sink labeled by $f(x)$. The *size* of the branching program is the number of nodes in G.

Our second model is less standard and comes from a line of work starting with [BoC92], more recently fleshed out in [BCK+14].

Definition 3 (Register program)**.** A *register program* P over a ring $\mathcal{R}$ is defined by a set of registers $R_1 \dots R_s$, each storing a value in $\mathcal{R}$, plus an ordered list of t instructions where for every $j \in [t]$ the jth instruction is $R_\ell \leftarrow R_\ell + p_j(f_j(x_i), R_1, \dots, R_{\ell-1}, R_{\ell+1}, \dots, R_s)$ for some $i \in [n]$, $\ell \in [s]$, function $f_j : [k] \to \mathcal{R}$, and polynomial p_j. The *size* of P is the number of registers s and the *time* of P is the number of instructions t.

For technical reasons we leave aside the notion of what it means to compute a function $f : [k]^n \to [k]$ with a register program until Section 3. For the rest of this paper, we'll set $\mathcal{R} = \mathbb{F}_2 = \{0, 1\}$, and all our constructions will be uniform. Uniformity allows us to make the following connections between register programs, branching programs, and space-bounded Turing machines.

[1] Measured in bits, the input length is still $2^h \operatorname{poly}(k)$.

Observation 1. *Let $f_n : [k]^n \to \{0,1\}^{m(n)}$ be a family of functions. Then there exists a uniform family of branching programs of size $2^{s(n)}$ computing f_n iff f_n can be deterministically computed in space $O(s(n))$.*

Observation 2. *Let $f_n : [k]^n \to \mathcal{R}^{m(n)}$ be a family of functions and let P_n be a uniform family of register programs with size $s(n)$ and time $t(n)$ such that when P_n is run on input $x \in [k]^n$ the final value in the registers is $f_n(x)$. Then f_n can be computed by a uniform family of branching programs of size $2^{s(n)} \cdot t(n)$.*

There is a converse to Observation 2 for well-structured branching programs called *commutative branching programs*, but we will not use this fact.

3 Main results

In this section, we present a new space-efficient TreeEval algorithm as a simple but effective variation on a recently-discovered "catalytic" approach [CM20].

Any algorithm following this approach has two ingredients, described in the following sections.

- Section 3.1 describes a way to encode each value in the TreeEval instance as a bit vector, and under that encoding, how to compute each internal node's associated function as a polynomial with carefully-controlled degree.
- In Section 3.2, we show how to build an algorithm for TreeEval based on this encoding. This is a straightforward adaptation of previous work, requiring only minor changes to work with a different encoding [CM20]. This will finish the proof of Theorem 1.

3.1 Encodings

The catalytic TreeEval approach begins with an *encoding*: a choice of vector corresponding to every value in $[k]$. Our algorithm will represent values using this encoding whenever possible.

Three encodings are defined below. We include the first two only for the sake of comparison; in this work, only the *d-hot* encoding is relevant.

Definition 4 (Encodings)**.** Let $\text{digit}(b, x, i)$ denote the i-th digit of the base b representation of x. For any $x \in [k]$,

	one-hot	binary	base 4	d-hot with $d = 2, b = 4$
0	0000000000000001	0000	00	0001 0001
1	0000000000000010	0001	01	0001 0010
5	0000000000100000	0101	11	0010 0010
15	1000000000000000	1111	33	1000 1000

Table 1: Example encodings with $k = 16$, written in reverse to match the usual convention for writing numbers. The encodings are described in Definition 4. The second-last column shows each number in base 4, for comparison with the d-hot encoding.

- The *one-hot encoding* [CM20] of x is the vector $\vec{p} \in \{0,1\}^k$ where $p_x = 1$ and $p_{x'} = 0$ for all $x' \neq x$. (k bits)
- The *binary encoding* [CM20] of x is just x written in base 2; that is, a vector $\vec{p} \in \{0,1\}^{\lceil \log k \rceil}$ where $p_i = \text{digit}(2, x, i)$. ($\lceil \log k \rceil$ bits)
- The *d-hot* encoding of x is parameterized by positive integers b, d where $b^d \geq k$. We write x as d digits in base b, and encode each digit using a one-hot encoding in $\{0,1\}^b$. In other words, the encoding is a vector $\vec{p} \in \{0,1\}^{d \cdot b}$ where for each $i \in [d]$, $p_{i,\text{digit}(b,x,i)} = 1$, and all other coordinates are 0. ($d \cdot b \geq d \cdot \lceil k^{1/d} \rceil$ bits)

Examples of each of the encodings in Definition 4 are illustrated in Table 1. The one-hot and binary encodings are natural and widely-used encodings, and were the central focus of [CM20]. Our central contribution in this paper is a definition and analysis of the d-hot encoding, which interpolates between them.[2] In particular, we can view each encoding as first writing down our value in some base $b \in [2 \ldots k]$, and second writing each digit in the resulting string using a string of length b with exactly one 1 in the position corresponding to the value of the digit.[3]

First we should ask: why do we need to interpolate between the two encodings? The binary encoding is the most space-efficient, using the minimum possible $\lceil \log k \rceil$ bits. By contrast, the one-hot encoding is the most efficient by a different measure: polynomial degree.

[2] A different hybrid encoding appeared in [CM20], but it fell short of meeting the challenge of [CBM+09].

[3] The binary encoding can be written in this way by doubling the size of the encoding to write 0 as 01 and 1 as 10. For example 5 would be written as 01 10 01 10 instead of 0101 as before.

Recall that a $\mathsf{TreeEval}_{k,h}$ input includes a function $f_v : [k] \times [k] \to [k]$ at each internal node v. For each encoding, we show how to convert this function into a polynomial.

Definition 5 (Polynomial representation of a function)**.** Fix a number $k \in \mathbb{N}$, an encoding E for values in $[k]$ as s-bit strings (as in Definition 4), and any function $f : [k] \times [k] \to [k]$.

A *polynomial representation of* f with respect to E is a tuple of s polynomials $\overrightarrow{Q_f} = (Q_{f,1}, \ldots, Q_{f,s})$ over $\mathcal{R} = \mathbb{F}_2$ which together compute f in the following sense. For any $x_\ell, x_r \in [k]$, let $\vec{p_\ell}, \vec{p_r} \in \{0,1\}^s$ be their encodings. Then $(Q_{f,i}(\vec{p_\ell}, \vec{p_r}))_{i \in [s]}$ is the encoding of $f(x_\ell, x_r)$.

We associate a polynomial representation with each of the encodings from Definition 4.

- One-hot encoding [CM20]: for $w \in [k]$,

$$Q_{f,w}(\vec{p_\ell}, \vec{p_r}) = \sum_{(y,z)} [f(y,z) = w] p_{\ell,y} p_{r,z}$$

  (degree 2).

- Binary encoding [CM20]: for $i \in \lceil \log k \rceil$,

$$Q_{f,i}(\vec{p_\ell}, \vec{p_r}) = \sum_{(w,y,z) \in [k]^3} [\mathrm{digit}(2,w,i) = 1][f(y,z) = w] \cdot \prod_{i' \in [\lceil \log k \rceil]} (p_{\ell,i'} + \mathrm{digit}(2,y,i') + 1)(p_{r,i'} + \mathrm{digit}(2,z,i') + 1)$$

  (degree $2\lceil \log k \rceil$).

- d-hot encoding: for $(i,a) \in [d] \times [b]$,

$$Q_{f,i,a}(\vec{p_\ell}, \vec{p_r}) = \sum_{(w,y,z) \in [k]^3} [\mathrm{digit}(b,w,i) = a][f(y,z) = w] \cdot \prod_{i' \in [d]} p_{\ell,i',\mathrm{digit}(b,y,i')} p_{r,i',\mathrm{digit}(b,z,i')}$$

  (degree $2d$).

3.2 Products and Clean Computation

Here, we show how to build an algorithm for $\mathsf{TreeEval}$ using the encoding from the previous section. The methods introduced in this part are a straightforward adaptation of past work [CM20].

In this section we show how to do efficient space-bounded computation with polynomials, using a protocol called *clean computation* [BCK+14]. In the setting of clean computation, we imagine memory starts out filled with some initial values that are beyond our control. Where an ordinary algorithm computing a function f would be expected to overwrite certain registers with $f(x)$, at the end of a clean computation we must have *added* $f(x)$ to the values of the output registers (over $\mathcal{R}$), and *restored all other registers to their initial states.*

Definition 6 (Clean computation)**.** Let $\mathcal{R}$ be a ring, $f : [k]^n \to \mathcal{R}^m$ a function and $S \subseteq [m]$ a set of output indices. Let P be a register program over $\mathcal{R}$ with s registers, $s \geq m$. Let $\tau_i \in \mathcal{R}$ denote the initial value of register R_i. We say P *cleanly computes* bits S of f if for all inputs $\vec{x}$ and initial register values $\vec{\tau}$, the final values after running P are

$$R_i = \tau_i + f(\vec{x})_i \quad \forall i \in S$$

and

$$R_j = \tau_j \quad \forall j \in [s] \setminus S$$

Using a clean computation as a subroutine lets us save space: rather than allocating new registers for the subroutine's scratch work, we can re-use registers the parent computation is already using. The following two lemmas show that this can be used to solve $\mathsf{TreeEval}$ recursively.

First, we can cleanly compute the value at any leaf.

Lemma 2. *Fix a height h and alphabet size k, and numbers $b, d \in \mathbb{N}$ such that $b^d \geq k$. Let v be a leaf node in the height-h complete binary tree. For some $\mathsf{TreeEval}_{k,h}$ input, let $\vec{p_v} \in {\mathbb{F}_2}^{d \cdot b}$ denote the d-hot encoding of the value at node v.*

Then for every subset $T \subseteq [d \cdot b]$, there is a register program $P_v(T)$ which cleanly computes bits T of $\vec{p_v}$ in space $d \cdot b$ and time at most $d \cdot b$.

Proof. For each $(i, a) \in [d] \times [b]$, define the function $g_{i,a} : [k] \to \{0, 1\}$ so that $g_{i,a}(x)$ is the (i, a)-th coordinate of the d-hot encoding of x (Definition 4).

The value at leaf v is directly encoded as a single input variable x_v Therefore, the (i, a)-th coordinate of $\vec{p_v}$ can be computed as $p_{v,i,a} = g_{i,a}(x_v)$. Our program is as follows:

1: **for** $(i, a) \in T$ **do**
2: $\quad R_{i,a} \leftarrow R_{i,a} + g_{i,a}(x_v)$
3: **end for**

Note that $R_{i,a} \leftarrow R_{i,a} + g_{i,a}(x_v)$ is an allowed register program instruction: $g_{i,a}$ takes the role of the function f_j in Definition 3, and the single input x_v takes the role of x_i.

There is one register for each index $(i, a) \in [d] \times [b]$ of the d-hot encoding, so this program has space $d \cdot b$. The number of instructions is $|T| \leq d \cdot b$. $\square$

Remark 3.1. The careful reader might notice an inefficiency in the algorithm that appears in Lemma 2. In the end, our goal is to produce a branching program. Under Observation 2, the above register program will become a sequence of $|T|$ layers of a branching program, each of which queries the same input x_v. This is wasteful because one layer of a branching program can compute an arbitrary function as long as it depends on only one input index. However, the factor of $|T|$ is insignificant; we gladly pay it to keep our presentation simpler by sticking with the register program point of view.

Continuing our construction of a recursive algorithm, the next lemma shows that we can cleanly compute the value at any internal node v using subroutines that cleanly compute the values at v's children.

Lemma 3. *Fix a height h and alphabet size k, and numbers $b, d \in \mathbb{N}$ such that $b^d \geq k$. Let v be an internal node in the height-h complete binary tree, and ℓ and r the children of v. For some $\mathsf{TreeEval}_{k,h}$ input, let $\overrightarrow{p_v}, \overrightarrow{p_\ell}, \overrightarrow{p_r} \in {\mathbb{F}_2}^{d \cdot b}$ denote the d-hot encodings of the values at nodes v, ℓ and r respectively.*

Suppose that for all subsets $S, S' \subseteq [d] \cdot [b]$, there exist register programs $P_\ell(S)$ and $P_r(S')$ which cleanly compute bits S of $\overrightarrow{p_\ell}$ and bits S' of $\overrightarrow{p_r}$, respectively, in space s and time t.

Then for every subset $T \subseteq [d \cdot b]$, there is a register program $P_v(T)$ which cleanly computes bits T of $\overrightarrow{p_v}$ in space $\max(s, 3db)$ and time $2^{2d}(2t + dbk^2)$.

Proof. We will show how to use the polynomial representation of the function f_v at node v to compute $\overrightarrow{p_v}$ using the subroutines $P_\ell(S)$ and $P_r(S')$. The proof is similar to the proofs of Lemmas 8 and 9 in [CM20].

Warm-up. As a warm-up, consider the following problem. There are d functions $f_1, \ldots, f_d$, and on input x our goal is to cleanly compute $\prod_{i \in [d]} f_i(x)$ into register R^{out}.[4] In order to access the functions $f_i(x)$, for every subset $S \subseteq [d]$, we have a program $P^{in}(S)$ which cleanly computes bits S of $(f_1(x), \ldots, f_d(x))$ into registers $(R^{in}_1, \ldots, R^{in}_d)$.

Define $S \Delta S'$ to be the symmetric set difference $(S \setminus S') \cup (S' \setminus S)$. We claim the following program P^{out} cleanly computes $\prod_{i \in [d]} f_i(x)$ into R^{out}:

[4] In the notation of Definition 6, $m = 1$ and $S = [m]$.

1: Initialize $S_{\text{old}} = \emptyset$
2: **for** $S \subseteq [d]$ **do**
3: Execute $P^{in}(S \Delta S_{\text{old}})$ on $\overrightarrow{R^{in}}$
4: $R^{out} \leftarrow R^{out} + (-1)^{d-|S|} \cdot \prod_{i \in [d]} R_i^{in}$
5: $S_{\text{old}} \leftarrow S$
6: **end for**

The for loop can be executed in any order as long as each subset $S \subseteq [d]$ is considered once. The factors $(-1)^{d-|S|}$ are moot in our chosen ring $\mathbb{F}_2$, but are included so that the algorithm works over any ring.

To understand how it works, first note that the call to $P^{in}(S \Delta S_{\text{old}})$ ensures that for each $i \in [d]$, register R_i^{in} holds its original value τ_i if $i \notin S$ and $\tau_i + f_i(x)$ otherwise. Consider first the iteration where $S = [d]$. We add $\prod_{i \in [d]}(\tau_i^{in} + f_i(x))$ to R^{out}, which gives us the term $\prod_{i \in [d]} f_i(x)$ plus a sum of junk terms $\sum_{S' \neq \emptyset}(\prod_{i \in S'} \tau_i^{in} \prod_{i \notin S'} f_i(x))$. The remaining iterations cancel these terms out, in a way reminicscent of the inclusion-exclusion principle for counting:

$$\begin{aligned}
R^{out} =& \tau^{out} + \sum_{S \subseteq [d]} (-1)^{d-|S|} \prod_{i \in [d]} (\tau_i^{in} + [i \in S] f_i(x)) \\
=& \tau^{out} + \sum_{S \subseteq [d]} (-1)^{d-|S|} \sum_{U \subseteq S} \left(\prod_{i \in U} f_i(x) \right) \left(\prod_{i \in [d] \setminus U} \tau_i^{in} \right) \\
=& \tau_0 + \sum_{U \subseteq [d]} \left(\sum_{\substack{S \subseteq [d] \\ U \subseteq S}} (-1)^{d-|S|} \right) \left(\prod_{i \in U} f_i(x) \right) \left(\prod_{i \in [d] \setminus U} \tau_i^{in} \right) \\
=& \tau_0 + \prod_{i \in [d]} f_i(x)
\end{aligned}$$

where the last equality follows from the fact that $\sum_{\substack{S \subseteq [d] \\ U \subseteq S}} (-1)^{d-|S|}$ is zero except when $U = [d]$. P^{out} uses $d+1$ registers $R^{out}, R_1^{in}, \ldots, R_d^{in}$ not counting those used by P^{in}. Because P^{in} is a clean computation, it is free to re-use P^{out}'s working memory (R^{out} in this case), and so our total space usage is $\max(s, d+1)$. P^{out} runs in time $2^d(t+1)$ where t is the runtime of $P^{in}(S)$ (assuming for simplicity it does not depend on S).

Real implementation. Now we consider the general problem. Let $\overrightarrow{Q_{f_v}}$ be the polynomial representation (Definition 5) of f_v. (Recall that the value at node v is f_v applied to the values at nodes ℓ and r.) Let $\overrightarrow{R^{\text{out}}}, \overrightarrow{R^{\ell}}, \overrightarrow{R^{r}}$ be

vectors of $d \cdot b$ registers holding values in $\mathbb{F}_2$ (for a total of $3db$ registers), and recall that $P_\ell(S)$ is a register program which cleanly computes $p_{\ell,i,a}$ into $R^\ell_{i,a}$ for all $(i,a) \in S$ (and likewise for $P_r(S')$). We will give a program $P^{out}(T)$ which cleanly computes $p_{v,i,a} = Q_{f_v,i,a}(\overrightarrow{p_\ell}, \overrightarrow{p_r})$ into $R^{out}_{i,a}$ for all $(i,a) \in T$.

Our program differs from the warm-up in three ways:

1. Instead of a single product, we compute the whole polynomial $Q_{f_v,i,a}$ from Definition 5. To do this, we compute all its monomials in parallel, using the following trick. Partition the $[d] \times [b]$ coordinates of $\overrightarrow{p_\ell}$ into d sets $\{1\} \times [b], \{2\} \times [b], \ldots, \{d\} \times [b]$, and partition the coordinates of $\overrightarrow{p_r}$ similarly. Note that each monomial in $Q_{f_v,i,a}$ includes exactly one variable from each of these $2d$ sets. Therefore, if we replace the main loop in our warm-up with a loop over all $V, V' \subseteq [d]$, and in each case set $S = V \times [b], S' = V' \times [b]$, then from the point of view of any particular monomial m our program will act exactly like the warmup. To parallelize across all monomials at once, we simply add the entire polynomial $Q_{v,i,a}(\overrightarrow{R^\ell}, \overrightarrow{R^r})$ to $R^{out}_{i,a}$.

2. Our inputs come from two different programs $P_\ell(S)$ and $P_r(S')$. Thus if we range over all pairs (V, V'), our runtime for the recursive calls will be $(2^d)^2 \cdot 2t$.

3. We need to cleanly compute $Q_{v,i,a}$ into $R^{out}_{i,a}$ for every $(i,a) \in T$, not just a single one. Again we can simply do each in parallel inside the loop, and the analysis for each (i,a) will be separate.

Concretely, our program P^{out} works as follows:

1: Initialize $S_{\text{old}} = \emptyset$, $S'_{\text{old}} = \emptyset$.
2: **for** $V, V' \subseteq [d]$ **do**
3: $S \leftarrow V \times [b]$, $S' \leftarrow V' \times [b]$
4: Execute $P_\ell(S \Delta S_{\text{old}})$ with output to $\overrightarrow{R^\ell}$.
5: Execute $P_r(S' \Delta S'_{\text{old}})$ with output to $\overrightarrow{R^r}$.
6: **for** $(i,a) \in T$ **do**
7: $R^{\text{out}}_{i,a} \leftarrow R^{\text{out}}_{i,a} + (-1)^{2d-|V|-|V'|} Q_{f_v,i,a}(\overrightarrow{R^\ell}, \overrightarrow{R^r})$
8: **end for**
9: $S_{\text{old}} \leftarrow S$
10: $S'_{\text{old}} \leftarrow S'$
11: **end for**

Line 7 cannot be executed as a single register program instruction, because the polynomial $Q_{f_v,i,a}$ depends on several values of the function f_v. (Recall

that in a TreeEval input, the function f_v is encoded as a table of k^2 separate values.) Instead, the line can be executed by k^2 register program instructions, by grouping the terms $[f(y,z)=w]$ according to the pair (y,z).

We now analyze the values of the output registers once the algorithm finishes, by a close examination of the effect of line 7 using the definition of $Q_{f_v,i,a}$ from Definition 5. For any $(i,a)\in T$,

$$\begin{aligned}
R^{\text{out}}_{i,a} =& \tau^{\text{out}}_{i,b} + \sum_{V,V'\subseteq[d]} (-1)^{2d-|V|-|V'|} \cdot \sum_{(w,y,z)\in[k]^3} [\text{digit}(b,w,i)=a][f_v(y,z)=w]\cdot \\
& \left(\prod_{i'\in[d]} (\tau^{\ell}_{i',\text{digit}(b,y,i')} + [i'\in V] p_{\ell,i',\text{digit}(b,y,i')})\right)\cdot \\
& \left(\prod_{i'\in[d]} (\tau^{r}_{i',\text{digit}(b,z,i')} + [i'\in V'] p_{r,i',\text{digit}(b,z,i')})\right) \\
=& \tau^{\text{out}}_{i,a} + \sum_{U,U'\subseteq[d]} \left(\sum_{\substack{V,V'\subseteq[d]\\ U\subseteq V, U'\subseteq V'}} (-1)^{2d-|V|-|V'|}\right)\cdot \\
& \sum_{(w,y,z)\in[k]^3} [\text{digit}(b,w,i)=a][f_v(y,z)=w]\cdot \\
& \left(\prod_{i'\in U} p_{\ell,i',\text{digit}(b,y,i')}\right)\left(\prod_{i'\in[d]\setminus U} \tau^{\ell}_{i',\text{digit}(b,y,i')}\right)\cdot \\
& \left(\prod_{i'\in U} p_{r,i',\text{digit}(b,z,i')}\right)\left(\prod_{i'\in[d]\setminus U} \tau^{r}_{i',\text{digit}(b,z,i')}\right) \\
=& \tau^{\text{out}}_{i,a} + \sum_{(w,y,z)\in[k]^3} [\text{digit}(b,w,i)=a][f_v(y,z)=w]\cdot \\
& \left(\prod_{i'\in[d]} p_{\ell,i',\text{digit}(b,y,i')}\right)\left(\prod_{i'\in[d]} p_{r,i',\text{digit}(b,z,i')}\right) \\
=& \tau^{\text{out}}_{i,a} + Q_{f_v,i,a}(\vec{p_\ell},\vec{p_r})
\end{aligned}$$

The algorithm uses $2\cdot 2^{2d}$ calls to P_ℓ and P_r and an additional $2^{2d}dbk^2$ basic instructions (line 7). Our algorithm uses the $3db$ registers $\vec{p_v},\vec{p_\ell},\vec{p_r}$, not counting the space required to compute P_ℓ and P_r. However, since P_ℓ and P_r are promised to be clean computations, our algorithm can lend *all*

$3db$ registers to whichever program is currently being executed, and so each call to P_ℓ or P_r needs no more than s registers including the $3db$ already defined. □

From this we can recursively compute $\mathsf{TreeEval}$, which will give our main result when applied to the d-hot encoding.

Theorem 4 ($\mathsf{TreeEval}$ algorithm)**.** *For any subset $T \subseteq [d \cdot b]$ there is a register program with $3db$ registers and length $O(2^{(2d+1)(h-1)}dbk^2)$ that cleanly computes bits T of the d-hot encoding of $\mathsf{TreeEval}_{k,h}$. (That is, given an input to $\mathsf{TreeEval}_{k,h}$, the program cleanly computes the encoding of the output.)*

Proof. We will prove by induction on the height h that the program has length at most $\frac{2^{(2d+1)h}-1}{2^{2d+1}-1}2^{2d}dbk^2 = O(2^{(2d+1)(h-1)}dbk^2)$. Lemma 2 solves the base case $h = 1$ using space $db \leq 3db$ and at most $db \leq \frac{2^{(2d+1)0}-1}{2^{2d+1}-1}2^{2d}dbk^2$ instructions. Now, assume for some height h that for every subset $S \subset [s]$, bits S of the encoding of $\mathsf{TreeEval}_{k,h}$ can be cleanly computed for some $h \geq 0$. Given an instance of $\mathsf{TreeEval}_{k,h+1}$, Let v be the root and let ℓ and r be the children. Under the induction hypothesis, there exist programs P_ℓ and P_r which can cleanly compute the d-hot encoding of f_ℓ and f_r in space $3db$ and time $\frac{2^{(2d+1)h}-1}{2^{2d+1}-1}2^{2d}dbk^2$. By Lemma 3 we can use P_ℓ and P_r to compute the d-hot encoding of f_v—and thus the output for the $\mathsf{TreeEval}_{k,h+1}$ instance—in space $3db$ and time at most $2^{2d}\left(2\frac{2^{(2d+1)h}-1}{2^{2d+1}-1}2^{2d}dbk^2 + dbk^2\right) = \frac{2^{(2d+1)(h+1)}-1}{2^{2d+1}-1}2^{2d}dbk^2$ as desired. □

The principle difference between Theorem 4 and previous algorithms that using the "catalytic" approach [CM20, Theorems 1–3] is the choice of encoding. Table 2 summarizes the trade-off between time and space for different encodings.

Proof of Theorem 1. Set $d = \lceil \log k / \log h \rceil$ and $b = h$; note that $b^d \geq k$. If we apply Theorem 4 for $T = [d \cdot b]$ and for a set of registers initialized to 0, then we get a register program computing the d-hot encoding of $\mathsf{TreeEval}_{k,h}$ into some registers while returning all other registers to 0. The register program uses $3dh$ registers and has length $O(2^{(2d+1)(h-1)}dhk^2) \leq 2^{O(dh)}$, and so it can be transformed into a branching program of size $2^{O(dh)}$ (see Observation 2). Note that each reachable output state corresponds to a different possible value of the d-hot encoding of $\mathsf{TreeEval}_{k,h}$ in the output registers plus 0 in all other registers. Since there are only k such values—one for each value in $[k]$—we relabel the k output nodes with the value their output register value corresponds to. Clearly this branching program computes $\mathsf{TreeEval}_{k,h}$.

encoding	one-hot	binary	d-hot
encoding bits (Def. 4)	k	$\lceil \log k \rceil$	$db(\geq d\lceil k^{1/d} \rceil)$
total space	$3k$	$3\lceil \log k \rceil$	$3db$
degree (Def. 5)	2	$\lceil \log k \rceil$	d
time for leaf node (Lem. 2)	$3k$	$3\lceil \log k \rceil$	$3db$
time for rec. step (Lem. 3)	$4(t + k^3)$	$k^2(2t + k^2\lceil \log k \rceil)$	$2^{2d}(2t + dbk^2)$
total time	$\Theta(4^{h-1}k^3)$	$\Theta((2k^2)^{h-1}k^4\lceil \log k \rceil)$	$\Theta(2^{(2d+1)h}dbk^2)$

Table 2: Trade-offs in Theorem 4 if different encodings had been used. The number of registers depends on the encoding (Definition 4). The total number of instructions depends on the number of recursive calls in Lemma 3, which in turn depends on the polynomial degree (Definition 5).

When $k \geq h$ we have $d = O(\log k / \log h)$, and so the size is $2^{O(h \log k / \log h)} = k^{O(h/\log h)}$. When $k \leq h$, we have $d = 1$, so the size is $2^{O(h)}$. □

References

[BCK+14] Harry Buhrman, Richard Cleve, Michal Kouckỳ, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 857–866. ACM, 2014.

[BCM+09a] Mark Braverman, Stephen Cook, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Branching programs for tree evaluation. In *International Symposium on Mathematical Foundations of Computer Science*, pages 175–186. Springer, 2009.

[BCM+09b] Mark Braverman, Stephen Cook, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Fractional pebbling and thrifty branching programs. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[BoC92] Michael Ben-or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, February 1992.

[CBM+09] Stephen Cook, Mark Braverman, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Branching programs: Avoiding barriers. Talk at Barriers Workshop at Princeton, August 2009. URL: https://www.cs.toronto.edu/~sacook/barriers.ps.

[CM20] James Cook and Ian Mertz. Catalytic approaches to the tree evaluation problem. In *Proceedings of the 52nd annual ACM symposium on Theory of computing*. ACM, 2020.

[CMW+12] Stephen 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, January 2012. arXiv version freely available at http://arxiv.org/abs/1005.2642. URL: http://doi.acm.org/10.1145/2077336.2077337, doi:10.1145/2077336.2077337.