Electronic Colloquium on Computational Complexity, Report No. 44 (2026)

Polynomial-Time Almost Log-Space Tree Evaluation by Catalytic Pebbling

Vahid R. Asadi Richard Cleve

*University of Waterloo**

Abstract

The Tree Evaluation Problem (TreeEval) is a computational problem originally proposed as a candidate to prove a separation between complexity classes P and L. Recently, this problem has gained significant attention after Cook and Mertz (STOC 2024) showed that TreeEval can be solved using $O(\log n \log\log n)$ bits of space. Their algorithm, despite getting very close to showing TreeEval $\in$ L, falls short, and in particular, it does not run in polynomial time.

In this work, we present the first polynomial-time, almost logarithmic-space algorithm for TreeEval. For any $\varepsilon > 0$, our algorithm solves TreeEval in time poly(n) while using $O(\log^{1+\varepsilon} n)$ space. Furthermore, our algorithm has the additional property that it requires only $O(\log n)$ bits of free space, and the rest can be catalytic space. Our approach is to trade off some (catalytic) space usage for a reduction in time complexity.

1 Introduction and statement of results

The Tree Evaluation Problem (TreeEval) is a computational problem that was originally proposed in [CMW+12] as a candidate to separate the complexity classes P and L. The problem consists of a d-ary tree of height h where an arbitrary function is assigned to each internal node that maps $d\ell$ bits to ℓ bits, and there is a ℓ bit string assigned to each leaf node as input. The output is the value of the root when the tree is evaluated in a bottom-up way from the leaves. The intuition behind the proposal to separate P and L using this problem was that, in the setting where $d = 2$ and $h = \ell = \log n$ for some $n \in \mathbb{N}$, the most natural algorithm for TreeEval is to evaluate the tree by *pebbling* it, and while this process is polynomial-time, it needs $\Omega(\log^2 n)$ space ($\log n$ pebbles, each consisting of $\log n$ bits). The fact that the functions at the nodes are arbitrary suggests that there is no special structure that can be exploited to permit a strategy much better than pebbling.

Recently, a novel construction of Cook and Mertz [CM24] improved this bound significantly. In particular, they showed that TreeEval can be solved in space $O(\log n \cdot \log\log n)$. Although this does not quite show TreeEval $\in$ L, it invalidated the intuition that $\widetilde{\Omega}(\log^2 n)$ is the minimum amount of space needed to solve it. Goldreich [Gol25b] slightly improved the space bound to $O(\log n \cdot \log\log n/\log\log\log n)$. The results in [CM24] were employed in the acclaimed result in [Wil25], showing that every multi-tape Turing machine running in time t can be simulated by another multi-tape Turing machine running in space $O(\sqrt{t \log t})$.

A drawback of the Cook-Mertz construction [CM24] (and that of [Gol25b]) is that it does not run in polynomial time. Intuitively, this is because, to keep the workspace as small as possible, they do all of the operations in a *catalytic* way. The catalytic model of computation, defined in [BCK+14], is a model in which the algorithm has access to an additional amount of space, but it must return that space to its original configuration; hence the term catalytic. The Cook-Mertz

*Emails: vrasadi@uwaterloo.ca, cleve@uwaterloo.ca

ISSN 1433-8092

construction uses methodologies from catalytic computation in order to keep the space usage small; however, this approach requires a lot of recalculations, resulting in a super-polynomial runtime.

A natural question is: what kind of space efficiency is possible for procedures that run in polynomial time? Recently, [HPR26] asked whether a polynomial-time procedure can use less than $\Omega(\log^2 n)$ space (stated as an open question), and they obtained results about a different question:

> *What is the minimum amount of catalytic space required to solve* TreeEval *in polynomial time using* $O(\log n)$ *free space?*

Catalytic space is apparently a weaker resource than free space. From the known results that $\mathsf{TreeEval} \in \mathsf{LOGCFL} \subseteq \mathsf{TC}^1$ and the main result in [BCK+14], polynomial time using $O(\log n)$ free space and *polynomial* catalytic space is sufficient for TreeEval. [HPR26] improved on this by showing that, for all $\varepsilon > 0$, TreeEval can be solved in polynomial time, using $O(\log n)$ free space and only $2^{O(\log^\varepsilon n)}$ catalytic space. Their algorithm employs results from *locally decodable codes* and *private information retrieval systems*. Note that, for small $\varepsilon > 0$, $2^{O(\log^\varepsilon n)}$ is asymptotically less than n^δ, for all $\delta > 0$; however, it is asymptotically larger than $\log^c n$, for all $c \in \mathbb{N}$.

Our main result is the following.

Theorem 1 (Main Theorem)**.** *For any* $\varepsilon > 0$*, we have that* TreeEval *can be solved in time* $n^{1/\varepsilon + O(1)}$*, while using* $O(\log n)$ *bits of free space and* $\widetilde{O}(\log^{1+\varepsilon} n)$ *bits of catalytic space.*

Although this implies TreeEval can be solved in polynomial-time using asymptotically less than $\log^2 n$ total space, our result is a polynomial-time algorithm in a more restrictive space model, with only $O(\log n)$ free space, supplemented by $O(\log^{1+\varepsilon} n)$ catalytic space.

Our approach is self-contained, using the standard methodologies of catalytic computation, similar to those employed in [CM24], in a manner that trades off some (catalytic) space usage for a reduction in time complexity. One way of viewing our results is as an approach that is complementary to that of Cook-Mertz for analyzing the conjecture that $\mathsf{TreeEval} \in \mathsf{L}$. While their approach prioritizes minimizing workspace at the cost of super-polynomial time, we focus on maintaining polynomial-time efficiency and logarithmic free space while reducing the required catalytic space.

2 Preliminaries

We fix the following notation throughout this work. We have $[n] = \{1, 2, \ldots, n\}$. For a fixed prime $p \in \mathbb{N}$ and some parameter $n \in \mathbb{N}$, we use the term *register* for elements in $\mathbb{Z}_p$, and denote them by lowercase letters $x_1, x_2, \ldots$. We also use the term *array* for a collection of $\log n$ registers. We denote these by uppercase letters $X_1, X_2, \ldots$. In this work, log is always the base-2 logarithm.

For a d-ary tree T, we denote the root by $\emptyset$, and children of node $u \in \{0, 1, \ldots, d\}^{\leq h}$ are named by the string ui where $i \in \{0, 1, \ldots, d\}$. Now, we formally define the Tree Evaluation Problem.

Definition 2 (Tree evaluation problem)**.** Let $d, \ell, h \in \mathbb{N}$. The tree evaluation problem $\mathsf{TreeEval}^d_{h,\ell}$ is defined by a d-ary rooted tree T, such that an ℓ-bit value in $\{0,1\}^\ell$ is assigned to every leaf node $u \in \{0, 1, \ldots, d-1\}^h$ of T, and a function $f_u : \{0,1\}^{d\ell} \to \{0,1\}^\ell$ is assigned to every internal node $u \in \{0, 1, \ldots, d-1\}^{<h}$. The goal is to evaluate the function corresponding to the root in a bottom-up manner.

Remark 3. In this work, we will exclusively focus on the case where $d = 2$, and $h = \ell = \log n$ for some $n \in \mathbb{N}$. In this case, we will simplify the notation by dropping the subscripts and the superscript, denoting it by TreeEval. Extending our results to the general case is fairly simple, but it is not necessary to obtain the main result.

The following equation plays a principal role in the Cook-Mertz construction:

Lemma 4 ([CM24]). *Let $p \geq 2$ be a prime, and $d < m = p - 1$. Let ω_m be a fixed primitive m-th root of unity in $\mathbb{Z}_p$, and let τ_i, x_i for $i \in [d]$ be arbitrary values in $\mathbb{Z}_p$. Then, it holds that*

$$m^{-1} \sum_{j \in [m]} \prod_{i \in [d]} (\omega_m^j \tau_i + x_i) = \prod_{i \in [d]} x_i \ .$$

Based on Lemma 4, one can construct algorithms that perform computations using a full memory. A simple example is the circuit in Fig. 1, that adds the monomial $x_1 x_2 x_3$ to the last register.

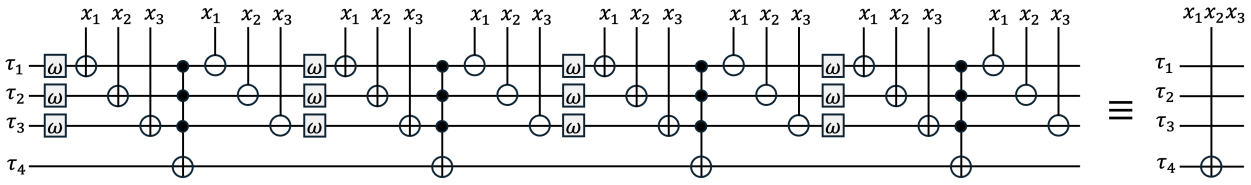

Figure 1: A circuit corresponding to Lemma 4, illustrated for $p = 5$. A simple pattern of gates is repeated $m = p - 1$ times. The $\oplus$ and $\ominus$ symbols denote addition and subtraction (modulo p). For any initial values of $\tau_1, \tau_2, \ldots, \tau_m \in \mathbb{Z}_p$, the first $m - 1$ registers incur no net change and the output of the last register is $\tau_m + x_1 x_2 \cdots x_{m-1} \bmod p$. The gate labeled ω, multiplies by ω_m, a fixed primitive m-th root of unity in $\mathbb{Z}_p$. The gate acting on m registers is an arithmetic generalization of a Toffoli gate, that maps each $(a_1, \ldots, a_{m-1}, a_m) \in \mathbb{Z}_p^m$ to $(a_1, \ldots, a_{m-1}, a_m + a_1 a_2 \ldots a_{m-1} \bmod p)$.

3 Overview of construction in terms of circuit diagrams

Here we demonstrate the basic idea behind the algorithm in Theorem 1 in the language of circuit diagrams (an algorithmic description and it's analysis are in Section 4). As in [CM24], the function $f_u : \{0,1\}^{2 \log n} \to \{0,1\}^{\log n}$ associated with each internal node u in the tree is extended to a function $F_u : (\mathbb{Z}_p)^{2 \log n} \to (\mathbb{Z}_p)^{\log n}$, where p is set to a prime larger than $2 \log n + 1$. The $\log n$ output values of F are each polynomials of degree $\leq 2 \log n$. We begin by introducing in Fig. 2 a condensed circuit notation where each wire carries an array consisting of $\ell = \log n$ elements of $\mathbb{Z}_p$.

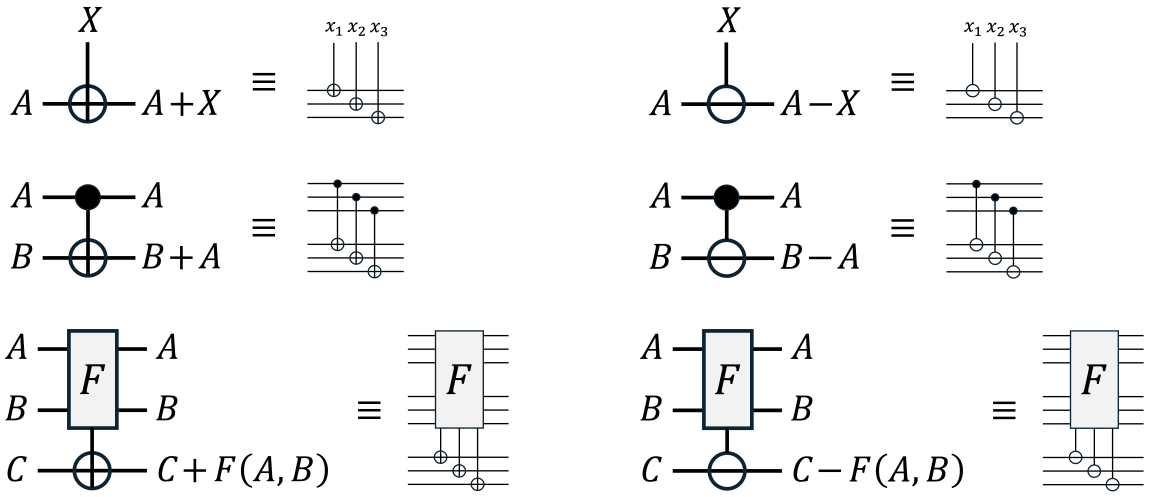

Figure 2: Condensed circuit notation, where the wires carry arrays consisting of ℓ registers, illustrated for $\ell = 3$. The input to F is 2ℓ elements of $\mathbb{Z}_p$ and the output is ℓ elements of $\mathbb{Z}_p$.

In Fig. 2 (for $\ell = \log n$) the gate labeled F represents a function $(\mathbb{Z}_p)^{2\log n} \to (\mathbb{Z}_p)^{\log n}$ (associated with some node in the TreeEval instance). The inputs to the gate labeled F are the first two arrays, and the output of F is added point-wise to the third array.

Fig. 3 contains circuits that illustrate the recursive structure of the Cook-Mertz construction. This figure is for the case $h = 2$, but it is not too difficult to see how it generalizes to larger height h.

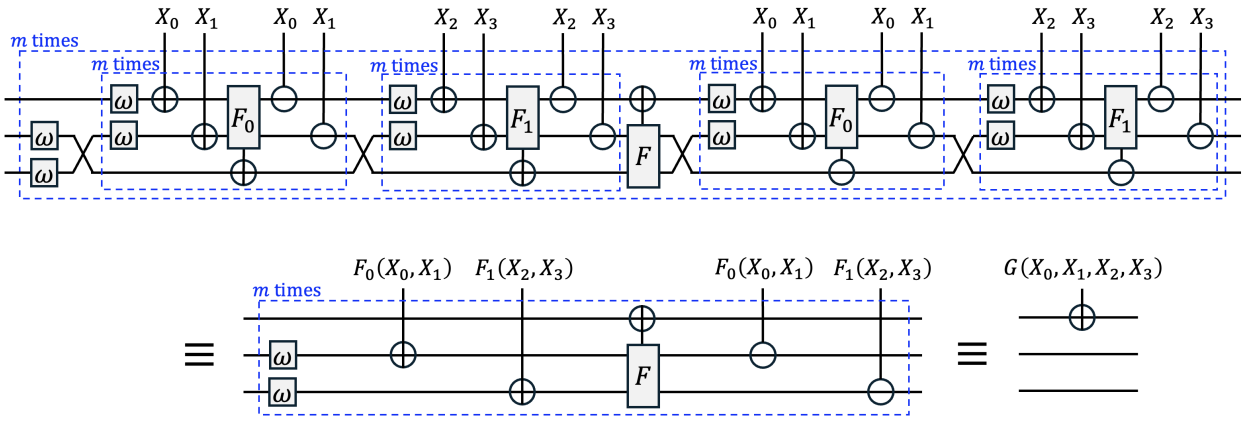

Figure 3: Depiction of two levels of the Cook-Mertz recursive tree-evaluation algorithm to compute $G(X_0, X_1, X_2, X_3) = F(F_0(X_0, X_1), F_1(X_2, X_3))$. Each dashed-line box labelled "m times" is shorthand for m sequential repetitions of its contents.

If we expand the condensed notation (explained in Fig. 2) that occurs in Fig. 3 and also expand the repetitions, then we obtain circuit structures similar to the circuit shown in Fig. 1, but for polynomials rather than monomials. Since this kind of circuit structure works for each monomial, it follows by linearity (over the field $\mathbb{Z}_p$) that it works for each polynomial. Therefore, the net effect of the circuit on the left side is to add the polynomials of $G(X_0, X_1, X_2, X_3)$ to the first array.

In Fig. 3, it can be seen from (two level) nested m-fold repetitions that this makes $(2m)^2$ recursive calls to each of the inputs X_0, X_1, X_2, X_3. More generally, for the case where the depth of G is r, this approach makes $(2m)^r$ recursive calls to each of the input arrays $X_0, X_1, \ldots, X_{2^r-1}$. Setting $r = \log n$, we can recover the result in [CM24], where $3 \log n$ registers that are $\mathbb{Z}_p$-valued are used for the data storage,[1] which amounts to $O(\log n \log\log n)$ bits and the length of the computation (resulting from unraveling the recurrence) is

$$(2m)^{\log n} n^{O(1)} = (4 \log n)^{\log n} n^{O(1)} = n^{\log\log n + O(1)} \quad .$$

Now, we are ready to explain the basic idea behind our new catalytic pebbling method. The idea is to evaluate $r = \varepsilon \log\log n$ levels of the tree as our recursive step, in a manner that balances (calatytic) space usage with run-time efficiency. Fig. 4 (on the next page) shows the case where $r = 2$, but it's not too difficult to see how it generalizes to larger r.

For each depth-r subtree, its 2^r input arrays are stored so as to avoid having to recursively recompute them; however, for space efficiency, these input arrays are stored *catalytically*, as offsets to registers that already store data (and we refer to this as *calalytic pebbling*).

The benefit of this approach is that it makes only $2m$ recursive calls to each of the inputs $X_0, X_1, \ldots, X_{2^r-1}$, instead of the $(2m)^r$ recursive calls that would result from a direct application of the Cook-Mertz approach. The price paid for this savings in recursive cost is the additional space required to store the catalytic pebbles, which is $2^r = \log^\varepsilon n$ arrays, each of which contains $\log n$ registers that are $\mathbb{Z}_p$-valued; the total number of bits is $O(\log^{1+\varepsilon} n \log\log n)$.

[1] In addition to the data storage, the algorithm also requires $O(\log n \log\log n)$ free bits for the program control.

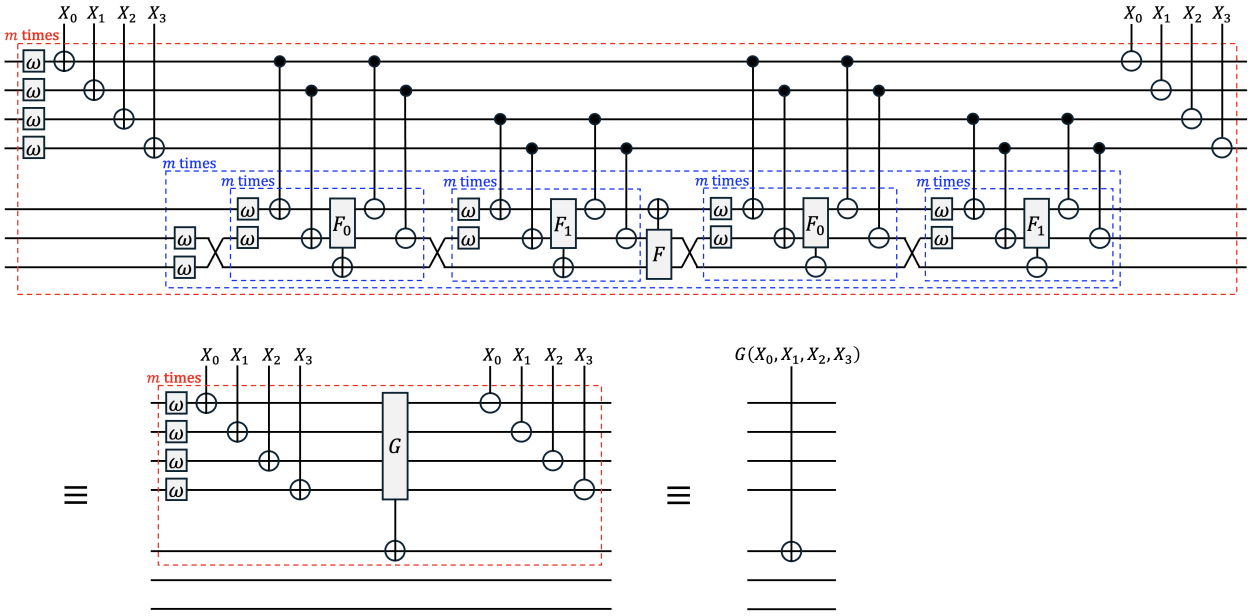

Figure 4: Computing $G(X_0, X_1, X_2, X_3) = F(F_0(X_0, X_1), F_1(X_2, X_3))$ by the catalytic pebbling method (the $r = 2$ case of a subtree of depth r). The first four (in general, 2^r) registers store the catalytic pebbles.

Note that, since $G(X_0, X_1, \ldots, X_{2^r-1})$ is of depth $r = \varepsilon \log\log n$, each output register is a polynomial of degree $\log^{1+\varepsilon} n$ in terms of the registers in the arrays $X_0, X_1, \ldots, X_{2^r-1}$. Therefore, for the construction in Fig. 4 to work correctly (specifically the repetitions in the red dashed-line box), we need to set $p \approx \log^{1+\varepsilon} n$ (slightly larger than $\log n$, which is sufficient for a direct application of the Cook-Mertz construction). This does not affect the asymptotic space usage, because it suffices to use $(1+\varepsilon)\log\log n$ bits to store elements of $\mathbb{Z}_p$.

For depth $r = \varepsilon \log\log n$, the number of steps of the Cook-Mertz construction (in the blue dashed-line boxes in Fig. 4) is bounded by

$$\left(8\log^{1+\varepsilon} n\right)^{\varepsilon \log\log n} n^{O(1)} = 2^{O((\log\log n)^2)} n^{O(1)} \ ,$$

which is polynomial. Note that this internal Cook-Mertz construction does not act directly on the inputs to the subtree $X_0, X_1, \ldots, X_{2^r-1}$; rather, it acts indirectly on these inputs by the process delineated by the red dashed-line box.

If we evaluate the entire tree in terms of subtrees that are each of depth $r = \varepsilon \log\log n$ (where the computation *within* each subtree is along the lines of the construction in Fig. 4) then how many such subtree-evaluations do we have to perform? The structure of the subtrees is an $(\log^{\varepsilon} n)$-ary tree of depth $(\log n)/(\varepsilon \log\log n)$. For each node, every one of it's $\log^{\varepsilon} n$ subtrees must be evaluated $\log^{1+\varepsilon} n$ times (as in the middle part of Fig. 4), so that's $\log^{1+2\varepsilon} n$ recursive subtree evaluations at each level. Unraveling the recursion, we get that the total number of subtree evaluations is

$$\left(\log^{1+2\varepsilon} n\right)^{(\log n)/(\varepsilon \log\log n)} = 2^{\log n \left(\frac{\log\log^{1+2\varepsilon} n}{\varepsilon \log\log n}\right)} = n^{\frac{1+2\varepsilon}{\varepsilon}} = n^{\frac{1}{\varepsilon}+2} \ .$$

The above does not account for rounding $\varepsilon \log\log n$ to an integer and for setting p to a prime such that $p = O(\log^{1+\varepsilon} n)$. It is straightforward to show that, accounting for these, as well as the cost associated with each subtree evaluation, results in a circuit-size (and run-time) that is $n^{1/\varepsilon + O(1)}$.

4 Formal analysis in terms of algorithms

We provide the proof of Theorem 1 in this section. Specifically, we state our algorithm in terms of pseudocode, prove its correctness, and analyze its complexity. Let an instance of TreeEval parameterized by n be given, and let $\varepsilon > 0$ be an arbitrary constant. The starting point is to fix a prime p, such that $\log^{1+\varepsilon} n + 1 \le p \le 2\log^{1+\varepsilon} n$, and let $m = p - 1$. Throughout the following, all arithmetic operations are performed modulo p, and we treat all registers as elements of $\mathbb{Z}_p$. Also, we use ω as a shorthand notation a fixed primitive m-th root of unity ω_m.

We now present the formal algorithm. Our approach employs a subroutine on subtrees of height $r = \varepsilon \log\log n$, which adapts the Cook-Mertz construction. We begin by recalling this construction, hereafter referred to as the CM Subroutine.

In this algorithm, X_u denotes the output of the node $u \in \{0,1\}^{\le r}$. If $u \in \{0,1\}^r$ is a leaf node, X_u is simply the value assigned to that node. For an internal node $u \in \{0,1\}^{<r}$, recall that we use $f_u : \{0,1\}^{2\log n} \to \{0,1\}^{\log n}$ for the function assigned to u, and let $F_u : (\mathbb{Z}_p)^{2\log n} \to (\mathbb{Z}_p)^{\log n}$ denote its extension to $\mathbb{Z}_p$. With this notation established, we have the recurrence $X_u = F_u(X_{u0}, X_{u1})$. The goal of the subroutine is to take the workspace registers $(\tau_1, \tau_2, \tau_{\text{out}})$ and a node u as input, and return $(\tau_1, \tau_2, \tau_{\text{out}} + X_u)$ as output.

CM Subroutine: Cook-Mertz construction for instances of height $\varepsilon \log\log n$

Input: A TreeEval instance t of height $r = \varepsilon \log\log n$ with workspace arrays $(\tau_1, \tau_2, \tau^{(1)}_{\text{out}})$.

The recursive step (input node $u \in \{0,1\}^{\le r}$):

1. **If** $u \in \{0,1\}^r$ is a leaf node, update the workspace arrays to
$$(\tau_1, \tau_2, \tau^{(1)}_{\text{out}} + X_u) \ .$$

2. **Otherwise, for** $i \in [m]$:

   a) Multiply each register of the first and second arrays of workspace by ω, so that their state becomes
   $$(\omega^i \tau_1, \omega^i \tau_2, \tau^{(i)}_{\text{out}}) \ .$$

   b) **Call** the recursive step[a] first on $u0$ and then on $u1$, so that the state of the workspace arrays updates to
   $$(\omega^i \tau_1 + X_{u0}, \omega^i \tau_2 + X_{u1}, \tau^{(i)}_{\text{out}}) \ .$$

   c) Apply F_u on the first and second arrays, and define $\tau^{(i+1)}_{\text{out}}$ to be
   $$\tau^{(i+1)}_{\text{out}} := \tau^{(i)}_{\text{out}} + m^{-1} F_u\left(\omega^i \tau_1 + X_{u0}, \omega^i \tau_2 + X_{u1}\right) \ .$$
   Then update the workspace to be
   $$\left(\omega^i \tau_1 + X_{u0}, \omega^i \tau_2 + X_{u1}, \tau^{(i+1)}_{\text{out}}\right) \ .$$

   d) **Call** the recursive step[a] again first on $u0$ and then on $u1$, this time subtracting the values so that the state of the workspace arrays updates to
   $$\left(\omega^i \tau_1, \omega^i \tau_2, \tau^{(i+1)}_{\text{out}}\right) \ .$$

[a] After proper rearrangement of arrays.

Our Main Algorithm proceeds by partitioning the tree into subtrees of height $r = \varepsilon \log\log n$. Each subtree is evaluated using the CM Subroutine and their values are stored catalytically.

We first establish the notation for the tree decomposition. Let T be the given TreeEval instance partitioned into subtrees of height r. Define $w = \log n/r = \log n/(\varepsilon \log\log n)$ as the depth of the decomposition, and let $k = 2^r = \log^{\varepsilon} n$ as the branching factor. We identify subtree t by a string of elements of $\{0,\dots,k-1\}^{\le r}$. For each subtree $t \in \{0,\dots,k-1\}^{<w}$, and any $j \in \{0,\dots,k-1\}$, let the concatenation tj denote the$j$-th child subtree of t.

Following the same logic, if $t \in \{0,\dots,k-1\}^{w}$, we treat it as a leaf node of the decomposition, where X_t is simply the input value assigned to that node. Otherwise, for any internal node $t \in \{0,\dots,k-1\}^{<w}$, we let $F_t : (\mathbb{Z}_p)^{k\log n} \to (\mathbb{Z}_p)^{\log n}$ be the polynomial extension of the function assigned to that node, when we abstract away the inner structure of the subtree and view it as a function of its k children. Therefore, the values satisfy the recurrence $X_t = F_t(X_{t0}, X_{t1}, \dots, X_{t(k-1)})$.

Main Algorithm: The catalytic pebbling algorithm.

Input: A TreeEval instance T of height $\log n$, pebble arrays and workspace arrays $(\rho_1, \rho_2, \dots, \rho_k, \tau_1, \tau_2, \tau^{(1)}_{\text{out}})$.

The recursive step (input subtree t where $t \in \{0,\dots,k-1\}^{\le w}$):

1. **If** $t \in \{0,\dots,k-1\}^{w}$, therefore it is a leaf, so update the output array by adding the corresponding input. The state of the arrays becomes
$$\left(\rho_1, \rho_2, \dots, \rho_k, \tau_1, \tau_2, \tau^{(1)}_{\text{out}} + X_t\right) .$$

2. **Otherwise, for** $i \in [m]$:
   a) Multiply each register of the pebble arrays by ω, so that their state becomes
   $$\left(\omega^i\rho_1, \omega^i\rho_2, \dots, \omega^i\rho_k, \tau_1, \tau_2, \tau^{(i)}_{\text{out}}\right) .$$
   b) **For** $j \in \{0,\dots,k-1\}$:
      i. **Call** the recursive step[a] on tj and update the corresponding pebble array to get
      $$\left(\omega^i\rho_1 + X_{t0}, \dots, \omega^i\rho_{j+1} + X_{tj}, \omega^i\rho_{j+2}, \dots, \omega^i\rho_k, \tau_1, \tau_2, \tau^{(i)}_{\text{out}}\right) .$$
   c) **Run** the CM Subroutine on the subtree t, while the inputs to the subroutine are the pebble arrays, and the workspace registers of the subroutine are $(\tau_1, \tau_2, \tau^{(i)}_{\text{out}})$. By setting $\tau^{(i+1)}_{\text{out}}$ to be
   $$\tau^{(i+1)}_{\text{out}} := \tau^{(i)}_{\text{out}} + m^{-1}F_t\left(\omega^i\rho_1 + X_{t0}, \omega^i\rho_2 + X_{t1}, \dots, \omega^i\rho_k + X_{t(k-1)}\right) ,$$
   we can say that the state of the arrays will change to
   $$\left(\omega^i\rho_1 + X_{t0}, \omega^i\rho_2 + X_{t1}, \dots, \omega^i\rho_k + X_{t(k-1)}, \tau_1, \tau_2, \tau^{(i+1)}_{\text{out}}\right) .$$
   d) **For** $j \in \{0,\dots,k-1\}$:
      i. **Call** the recursive step[a] again on tj and update the corresponding pebble array, this time by subtracting the output, so that we get
      $$\left(\omega^i\rho_1, \dots, \omega^i\rho_{j+1}, \omega^i\rho_{j+2} + X_{t(j+1)}, \dots, \omega^i\rho_k + X_{t(k-1)}, \tau_1, \tau_2, \tau^{(i+1)}_{\text{out}}\right) .$$

[a] After proper rearrangement of arrays.

4.1 Analysis of the algorithm

Proof of correctness. The correctness follows from an induction on n. The base case $n = 1$ is straightforward: the input consists of a single leaf, and the algorithm correctly adds its value to the output array in Step 1. For the inductive step, assume the algorithm works correctly for all instances of size $< n$. It follows that the recursive calls in Steps 2b and 2d return results consistent with our inductive hypothesis. Therefore, the state of the arrays align with the algorithm's specification. The correctness of the subsequent update in Step 2c then depends on the correctness of the CM Subroutine, which is established by Lemma 4. For a rigorous proof of the CM Subroutine, we refer the reader to [CM24].

It remains to be shown that the final state of the arrays matches the following.

$$\left(\rho_1, \rho_2, \ldots, \rho_k, \tau_1, \tau_2, \tau_{\text{out}}^{(1)} + X_\emptyset\right) \ ,$$

First, observe that for any pebble array, each step results in a multiplication by ω. Therefore, after m steps, each register is multiplied by ω^m. Given that ω is a primitive m-th root of unity, $\omega^m = 1$, and the registers return to their initial state. Regarding the output array we have

$$\begin{aligned}
\tau_{\text{out}}^{(m+1)} &= \tau^{(1)} + \sum_{i\in[m]} \left(m^{-1} F_\emptyset\left(\omega^i\rho_1 + X_0, \omega^i\rho_2 + X_1, \ldots, \omega^i\rho_k + X_{(k-1)}\right)\right) \\
&= \tau^{(1)} + F_\emptyset\left(X_0, X_1, \ldots, X_{(k-1)}\right) \\
&= \tau^{(1)} + X_\emptyset \ ,
\end{aligned}$$

where second equality follows from Lemma 4, and the property that each $F_t^{(j)}$ for $j \in [\log n]$ is a polynomial of degree $k \log n = \log^{1+\varepsilon} n$. Therefore, at the end of the recursive call, the state of the arrays is as expected: all pebble and workspace arrays have been returned to their initial states, and the output has been added to the output array.

Runtime analysis. We analyze the runtime by first bounding the number of recursive calls in the outer subroutine. Note that we can view this process as a tree of height w, where the arity of each node is $k = \log^\varepsilon n$ (we are counting the number of arrays). Each child is invoked $2m$ times. Given our choice of $p \le 2\log^{1+\varepsilon} n$, the total number of recursive calls in the outer subroutine is at most

$$\begin{aligned}
(2mk)^w \le (4\log^{1+\varepsilon} n \log^\varepsilon n)^{\left(\frac{\log n}{\varepsilon \log\log n}\right)} &= (4\log^{1+2\varepsilon} n)^{\left(\frac{\log n}{\varepsilon\log\log n}\right)} \\
&= 2^{(2+(1+2\varepsilon)\log\log n)\left(\frac{\log n}{\varepsilon\log\log n}\right)} \\
&= 2^{\left(\frac{2\log n}{\varepsilon\log\log n} + \frac{(1+2\varepsilon)\log n}{\varepsilon}\right)} \\
&= \left(2^{\log n}\right)^{\frac{2}{\varepsilon\log\log n} + \frac{1+2\varepsilon}{\varepsilon}} \\
&= n^{2+\frac{1}{\varepsilon}+o(1)} \ ,
\end{aligned}$$

where $o(1)$ term in the exponent accounts for setting p to be a prime and rounding $\varepsilon \log\log n$ to the nearest integer.

Each recursive call requires a runtime equivalent to one execution of the CM Subroutine. By similar reasoning, and accounting for the $n^{O(1)}$ overhead required to arithmetize functions of the form $\{0,1\}^{2\log n} \to \{0,1\}^{\log n}$, the runtime of the CM Subroutine is

$$(8\log^{1+\varepsilon} n)^{\varepsilon \log\log n} n^{O(1)} = 2^{(3+(1+\varepsilon)\log\log n)(\varepsilon \log\log n)} \cdot n^{O(1)}$$
$$\leq 2^{O_\varepsilon((\log\log n)^2)} n^{O(1)} .$$

Combining the two, we conclude that the total runtime of Main Algorithm will be

$$T(n) = n^{2+\frac{1}{\varepsilon}+o(1)} \cdot 2^{O_\varepsilon((\log\log n)^2)} n^{O(1)}$$
$$\leq n^{\frac{1}{\varepsilon}+O(1)} .$$

Space analysis. To analyze the space complexity of the Main Algorithm, we partition the space into three components, namely the *workspace* (denoted by τ_i), the *catalytic pebbles* (denoted by ρ_i), and the *control space*. Each array in the workspace and the pebble set consists of $\log n$ registers, where each register stores an element of $\mathbb{Z}_p$, therefore requires $\lceil \log p \rceil = \log\log n + O(1)$ many bits to represent. We have 3 arrays in the workspace and $k = \log^\varepsilon n$ many arrays for pebbles. Summing these, we require a total of

$$(3 + \log^\varepsilon n) \cdot \log n \cdot (\log\log n + O(1)) = \tilde{O}(\log^{1+\varepsilon} n)$$

many bits to represent the workspace and pebble space. This part of the space is catalytic. Regardless of the initial values written on them, they will always have those values at the end of execution.

In addition, we need to take the control space into account. The control space is dominated by the space used to store the counters for all the nested loops. There are $O(\log n/\log\log n)$ loops in the Main Algorithm that each repeat $\log^{O(1)} n$ times, so that's $O\big((\log n/\log\log n)(\log\log n)\big) = O(\log n)$ bits to store these the counters. There are additional counters in the CM Subroutine too, but these consume $O((\log\log n)^2)$ bits, since the depth of these subtrees is $O(\log\log n)$.

Proof of Theorem 1. We showed that the Main Algorithm solved TreeEval while using $O(\log n)$ bits of free space, $\widetilde{O}(\log^{1+\varepsilon} n)$ bits of catalytic space, and a has a running time of $n^{\frac{1}{\varepsilon}+O(1)}$. □

5 Conclusion

In this work, we present the first polynomial-time and almost logarithmic-space algorithm for TreeEval. For any arbitrary small constant $\varepsilon > 0$, we show that it is possible to solve TreeEval in time $n^{1/\varepsilon+O(1)}$ and simultaneous space $O(\log^{1+\varepsilon} n)$. Our construction has the additional property that it only uses $O(\log n)$ bits of free space, and the rest is catalytic. This can be viewed as a hybrid between the well-known *pebbling* technique, used by [CMW+12], and the recent Cook-Mertz construction from [CM24]. To see this, we briefly discuss two extreme scenarios:

- **When $r = 0$.** In this case, our algorithm matches the standard Cook-Mertz construction. This is because when $r = 0$, each subtree is equivalent to a single node, so the algorithm does not make use of the inner subroutine. We note that in this case, although the space is logarithmic, the runtime will be super-polynomial, which in turn will result in the need for super-logarithmic control space. This is essentially what happens in the Cook-Mertz construction, and it is the reason why that construction falls short of showing TreeEval $\in$ L.
- **When $r = \log\log n$.** On the other extreme, setting $r = \log\log n$ (or, equivalently, $\varepsilon = 1$) matches the complexity of the standard pebbling algorithm for TreeEval. In this case, the algorithm has access to $O(\log^2 n)$ bits of pebble space, runs in polynomial time, and makes use of subtrees of height $\log\log n$. The input to these subtrees will be $\log^2 n$ bits long, which we can keep in the pebbling space, and use to evaluate the final output.

5.1 Miscellaneous comments

Low-degree extension. A careful reader may notice that in our work, we did not bother with the better encoding of the inputs (as shown in [Gol25a]) using low-degree extension, which will result in avoiding the $O(\log \log n)$ blowup in the space complexity due to representing field elements in bits. This is intentional, due to a couple of reasons. The first reason is that we believe staying away from the low-degree extension keeps the notation slightly simpler and will make this manuscript more approachable for non-experts in the field. The second reason is that, because we aim for a $O(\log^{1+\varepsilon} n)$ space complexity, shaving an $O(\log \log n)$ term will not improve the results. So it gives us more incentive to keep things simpler by sticking to the slightly less optimal representation of field elements.

Reducing height by increasing arity. As discussed previously, [Gol25b] improves slightly on [CM24] to show an $o(\log n \cdot \log \log n)$ algorithm for TreeEval. Their idea is to think of subtrees of height $\log \log \log n$ as a single mega-node that maps inputs of length $\log \log n \log n$ to $\log n$ bits of output. This can be viewed as a new tree with lower height, at the cost of increased arity. Although both works deal with subtrees of carefully chosen heights, there are fundamental differences in how they handle them, and in particular, we exploit the substructure within the subtrees so as to achieve polynomial time and in a very restrictive space model.

Acknowledgments

This research was partially supported by Canada's NSERC, a Mike and Ophelia Lazaridis Fellowship, and a Cheriton Graduate Scholarship.

References

[BCK+14] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. "Computing with a full memory: catalytic space". *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 857–866, 2014 (pages 1, 2).

[CM24] James Cook and Ian Mertz. "Tree evaluation is in space $O(\log n \cdot \log \log n)$". *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1268–1278, 2024 (pages 1–4, 8–10).

[CMW+12] Stephen Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. "Pebbles and branching programs for tree evaluation". *ACM Transactions on Computation Theory (TOCT)*, 3(2), 2012 (pages 1, 9).

[Gol25a] Oded Goldreich. "On the Cook-Mertz tree evaluation procedure", *Computational Complexity and Local Algorithms: On the Interplay Between Randomness and Computation*, pages 102–112. Springer, 2025 (page 10).

[Gol25b] Oded Goldreich. "Solving tree evaluation in $o(\log n \cdot \log \log n)$ space", *Computational Complexity and Local Algorithms: On the Interplay Between Randomness and Computation*, pages 113–118. Springer, 2025 (pages 1, 10).

[HPR26] Alexandra Henzinger, Edward Pyne, and Seyoon Ragavan. "Catalytic tree evaluation from matching vectors". *Electronic Colloquium on Computational Complexity (ECCC)*, 2026 (page 2).

[Wil25] R. Ryan Williams. "Simulating time with square-root space". *Proceedings of the 57th Annual ACM Symposium on Theory of Computing (STOC)*, pages 13–23, 2025 (page 1).

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il