CS 860: Space Bounded Computation **University of Waterloo: Winter 2026**

Lecture 2: Tree Evaluation and Simulating Time with Space

January 20, 2026

Instructor: Sepehr Assadi

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

Topics of this Lecture

We will examine the breakthrough work of Cook and Mertz [CM24] on space-efficient algorithms for the *tree evaluation* problem, and one of its more important consequences due to Williams [Wil25] for simulating time on restricted Turing Machines in nearly square root space.

1 The Tree Evaluation Problem

The tree evaluation problem defined below—introduced first by [CMW+12]—is generalization of the circuit evaluation we saw in the last lecture.

Problem 1. In the tree evaluation problem for parameters $d, t \in \mathbb{N}$, we have a rooted full binary tree T of depth d. The root of the tree is named 0 and we recursively name left-child and right-child of a node named y with $y0$ and $y1$. Each internal node v is assigned a function $f_v : \{0,1\}^t \times \{0,1\}^t \to \{0,1\}^t$. Every node u in the tree is assigned a t-bit string, called its *value*, denoted by $val_u \in \{0,1\}^t$. For a leaf-node, the value is given explicitly as input. For an internal node v, its value is defined as

$$val_v := f_v(val_{v0}, val_{v1}),$$

namely, the string obtained by applying the function f_v to the values of its child-nodes.

The goal is that given the tree T and values of the leaf-nodes and the truth-tables of functions $\{f_v\}$ for internal-nodes, evaluate the value of the root.

This problem sometimes is called *word circuit evaluation* [Vio25] since it is basically evaluating a circuit with fan-in two and arbitrary gates on t-bit words. Our goal in this lecture is to study the space complexity of this problem.

Exactly as in the last lecture, there is an obvious algorithm for this problem:

```
Algorithm 1. Eval(node $v$): (returns the value $v_v$ of $v$)

  1. If $v$ is a leaf-node, return $val_v$;
  2. If $v$ is an internal node, recursive let $val_{v0} = $ Eval($v0$) and $val_{v1} = $ Eval($v1$) and return $f_v(v_0, v_1)$.
```

Let $S(d,t)$ denote the space complexity of `Eval` on a depth-d tree with labels in $\{0,1\}^t$. We claim that

$$S(d,t) \leqslant S(d-1,t) + O(t),$$

because the space the algorithm needs is to compute `Eval`$(v0)$ which is $S(d-1,t)$, then store the answer in $O(t)$ bits, then reuse the space of `Eval`$(v0)$ to now compute `Eval`$(v1)$ (while hanging on to the $O(t)$-bit answer of `Eval`$(v0)$) and finally use the two answers to look up the value of f_v and return the answer. Solving the above recursion implies that the space complexity of the algorithm is

$$S(d,t) = O(d \cdot t).$$

Thus, the naive solution for this problem uses $O(d \cdot t)$ space. Our goal is now to present a much more efficient algorithm for this problem that uses close to—but not entirely—$O(d+t)$ space.

1.1 Warm-Up: Linear Functions

Let us start with a warm-up example when the functions $\{f_v\}$ on the internal-nodes of the tree are *linear* functions, meaning that for $x, y, x', y' \in \{0,1\}^t$ and all internal-node v,

$$f_v(x \oplus x', y \oplus y') = f_v(x,y) \oplus f_v(x', y'),$$

where $\oplus$ is the bit-wise XOR operation. The solution in this case is going to be quite similar to the `Magic` algorithm we saw in Lecture 1 (for Boolean circuit evaluation).

Let R_1, R_2, R_3 be three *global* registers holding values in $\{0,1\}^t$. We define a function `LinearEval` that given a node v of the tree and an integer $i \in [3]$, has the following effect on these global registers:

- `LinearEval`(node v, integer $i \in [3]$): assuming the original values of the registers is (r_1, r_2, r_3), after running this function, we will have

$$R_i \leftarrow r_i \oplus val_v \qquad \text{and} \qquad R_j \leftarrow r_j \text{ for } j \neq i. \tag{1}$$

Equipped with this function, we can simply start with the assignment $(0^t, 0^t, 0^t)$ to the registers, call `LinearEval`$(root, 1)$ and return the content of R_1 as the answer. It remains to implement `LinearEval`.

```
Algorithm 2. LinearEval(node $v$, integer $i \in [3]$): (implements the transition stated in Eq (1))

  1. If $v$ is a leaf-node, update $R_i \leftarrow r_i \oplus val_v$.
  2. From hereon, for simplicity, we assume $i = 1$; the other two cases can be handled analogously.
  3. Run LinearEval($v0, 2$) and LinearEval($v1, 3$).
  4. Update $R_1 \leftarrow R_1 \oplus f_v(R_2, R_3)$.
  5. Run LinearEval($v0, 2$) and LinearEval($v1, 3$).
  6. Update $R_1 \leftarrow R_1 \oplus f_v(R_2, R_3)$.
```

We now analyze the correctness and space complexity of `LinearEval`.

Correctness. The correctness of the base case of the algorithm is immediate. We prove the rest inductively and again focus on the $i = 1$ case. We simply need to keep track of the values of registers (R_1, R_2, R_3):

- Initially:
$$(\ r_1 \ , \ r_2 \ , \ r_3 \ ).$$
- After Line (3):
$$(\ r_1 \ , \ r_2 \oplus val_{v0} \ , \ r_3 \oplus val_{v1} \ ),$$
by the induction hypothesis and the guarantees dictated by Eq (1).
- After Line (4):
$$(\ r_1 \oplus f_v(r_2 \oplus val_{v0}, r_3 \oplus val_{v1}) \ , \ r_2 \oplus val_{v0} \ , \ r_3 \oplus val_{v1} \ ).$$
- After Line (5):
$$(\ r_1 \oplus f_v(r_2 \oplus val_{v0}, r_3 \oplus val_{v1}) \ , \ r_2 \ , \ r_3 \ ),$$
by the induction hypothesis and the guarantees dictated by Eq (1), and since taking $\oplus$ again cancels its first application.
- After Line (6):
$$(\ r_1 \oplus f_v(r_2 \oplus val_{v0}, r_3 \oplus val_{v1}) \oplus f_v(r_1, r_2) \ , \ r_2 \ , \ r_3 \ ),$$
which, *by the linearity of f_v*, can be simplified to
$$(\ r_1 \oplus f_v(val_{v0}, val_{v1}) \ , \ r_2 \ , \ r_3 \ ),$$
which in turn is
$$(\ r_1 \oplus val_v \ , \ r_2 \ , \ r_3 \ ),$$
as desired.

Space complexity. Let $S(d,t)$ denote the space complexity of `LinearEval` *except for* the three global variables R_1, R_2, R_3 it uses. We also assume that there is one other global register *temp* that the algorithm each time uses to compute $f_v(R_1, R_2)$ in Lines (4) and (6), but does not need to maintain any particular invariant on its value (unlike R_1, R_2, R_3 as dictated by Eq (1))[1]. Thus, the space complexity of `LinearEval` is $4t + S(d,t)$. Finally, in the language of the last lecture, $S(d,t)$ practically only measures the space needed to "follow the code" in `LinearEval`. We have,
$$S(d,t) = S(d-1,t) + O(1),$$
because we only need to remember which line of `LinearEval` we are currently on and the name of the next node in the recursive call (which is obtained by concatenating a single bit to the name of the current node). Thus, we have $S(d,t) = O(d)$ implying that the total space complexity of the algorithm is $O(d+t)$ bits.

This concludes an algorithm with space complexity $O(d+t)$ for Tree Evaluation in the special case of all internal-node functions being linear. Of course, for the general problem, there is no such guarantee and more needs to be done to achieve done. We do so in the next section.

2 The Main Algorithm for Tree Evaluation

We have the following theorem due to the breakthrough work of [CM24], and subsequent simplifications presented in [Gol24].

Theorem 1 ([CM24])**.** *There is an algorithm for the Tree Evaluation problem of depth d and t-bit values, with arbitrary input functions, that uses $O((d+t)\log t)$ bits of space.*

The proof of Theorem 1 is based on a quite clever use of *algebraization* techniques (using elementary properties of bounded-degree polynomials on finite fields). We first review the background on polynomial tools we need and then present the algorithm.

[1]We note that the use of this *temp* register is not necessary and the algorithm can simply update R_1 on the fly; we thus only use *temp* here for convenience of the analysis.

2.1 Background: Multi-linear Extensions and Polynomial Interpolation

Multi-linear extensions. Our goal is to, for a Boolean function $f : \{0,1\}^n \to \{0,1\}$, assign a *polynomial* in n variables, from a larger domain of elements, that agree with this function on Boolean inputs. Specifically, let $\mathbb{F}$ be some integer fields (wherein calculations are done modulo some fixed prime p). We define a polynomial $F : \mathbb{F}^n \to \mathbb{F}$ which has degree one on each of its variables and agrees with f on $\{0,1\}^n$, namely,

$$F(x) = f(x) \qquad \text{for all } x \in \{0,1\}^n. \tag{2}$$

The polynomial can be defined as follows

$$F(x_1,\ldots,x_n) = \sum_{a_1,\ldots,a_n \in \{0,1\}^n} \mathbb{I}(a_1 = x_1 \wedge a_2 = x_2 \wedge \cdots \wedge a_n = x_n) \cdot f(a_1,\ldots,a_n),$$

where $\mathbb{I}(B)$ for a Boolean predicate B returns 1 if the predicate is true and 0 otherwise. It is immediate to see that F this way satisfies Eq (2). We are not done yet though as $\mathbb{I}(\cdot)$ is still not a polynomial. However, that can be easily fixed since

$$\mathbb{I}(a_1 = x_1 \wedge a_2 = x_2 \wedge \cdots \wedge a_n = x_n) = \prod_{i=1}^{n} \mathbb{I}(a_i = x_i) = \prod_{i=1}^{n} \left(a_i \cdot x_i + (1-a_i)\cdot(1-x_i)\right),$$

where we obtained the last term by considering the truth table of the equality function on two bits. Thus, the entire polynomial F can now be written as

$$F(x_1,\ldots,x_n) = \sum_{(a_1,\ldots,a_n) \in \{0,1\}^n} f(a_1,\ldots,a_n) \cdot \prod_{i=1}^{n} \left(a_i \cdot x_i + (1-a_i)\cdot(1-x_i)\right). \tag{3}$$

It is easy to see that F is linear in each individual variable x_i for $i \in [n]$.

Finally, we note that F is called the **multi-linear extension** of f and in fact is the unique polynomial satisfying the above properties (although we do not need the latter property in this lecture).

Polynomial interpolation. Suppose $\mathbb{F}$ is again some integer field and $P : \mathbb{F} \to \mathbb{F}$ is a single-variable polynomial with degree n less than $|\mathbb{F}|$. Thus, we have,

$$P(x) = \sum_{i=0}^{n} c_i \cdot x^i,$$

for $n+1$ coefficients $(c_0,\ldots,c_n)$ in $\mathbb{F}$.

Recall that specifying any $n+1$ points of a degree n polynomial fixes the polynomial. We can write this for points $a_0,\ldots,a_n$ as the following matrix equation:

$$\begin{bmatrix} 1 & a_0 & a_0^2 & \cdots & a_0^n \\ 1 & a_1 & a_1^2 & \cdots & a_1^n \\ & & & \ddots & \\ 1 & a_n & a_n^2 & \cdots & a_n^n. \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} P(a_0) \\ P(a_1) \\ \vdots \\ P(a_n) \end{bmatrix}.$$

The matrix on the left is called the **Vandermonde matrix** and is reversible as long as $a_0,\ldots,a_n$ are distinct. This implies that we can recover the coefficients of the polynomial as:

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 1 & a_0 & a_0^2 & \cdots & a_0^n \\ 1 & a_1 & a_1^2 & \cdots & a_1^n \\ & & & \ddots & \\ 1 & a_n & a_n^2 & \cdots & a_n^n. \end{bmatrix}^{-1} \cdot \begin{bmatrix} P(a_0) \\ P(a_1) \\ \vdots \\ P(a_n) \end{bmatrix}.$$

Finally, if our goal is to evaluate the polynomial on some point $b \in \mathbb{F}$, we have

$$P(b) = \begin{bmatrix} 1 & b & b^2 & \dots & b^n \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 1 & b & b^2 & \dots & b^n \end{bmatrix} \cdot \begin{bmatrix} 1 & a_0 & a_0^2 & \cdots & a_0^n \\ 1 & a_1 & a_1^2 & \cdots & a_1^n \\ & & & \ddots & \\ 1 & a_n & a_n^2 & \cdots & a_n^n. \end{bmatrix}^{-1} \cdot \begin{bmatrix} P(a_0) \\ P(a_1) \\ \vdots \\ P(a_n) \end{bmatrix}. \quad (4)$$

This allows us to compute the value of the polynomial P on any point b as a *linear combination* of its values on $P(a_0), \dots, P(a_n)$ using the above formula. In other words, we can interpolate $P(b)$ *linearly* from P on any $n + 1$ points.

We also state, without proof, the following fact about matrix inversion in bounded space.

Fact 2. *Fix any integer $n \geqslant 1$ and any field $\mathbb{F}$ with $|\mathbb{F}| > n$. For any invertible matrix $A \in \mathbb{F}^{n \times n}$, and any $i, j \in [n]$, we can calculate $(A^{-1})_{i,j}$ in $\mathrm{poly}(\log |\mathbb{F}|)$ space.*

A quick note about Fact 2 is that "linear algebra is in NC" which by that we mean computing determinant, matrix inversion, and matrix multiplication on N-bit inputs admits circuits of size $\mathrm{poly}(N)$ and depth $\mathrm{poly}(\log N)$. Fact 2 follows from this and the circuit evaluation we saw in the previous lecture that simulates a depth-d circuit in $O(d)$ space.

A corollary of Fact 2 and Eq (4) is that we can compute a value of degree n polynomial on any point, given input access to $n + 1$ points of the polynomial, in $\mathrm{poly}(\log |\mathbb{F}|)$ space

2.2 The Algorithm

We are now ready to present the algorithm `TreeEval`. Throughout, let $\mathbb{F}$ be an integer field modulo a fixed prime in $(2t + 1, 4t + 2)$, whose existence is guaranteed by Bertrand's postulate (aka Chebyshev's theorem). Yet again, the algorithm uses three *global* registers R_1, R_2, R_3, but now holding values in $\mathbb{F}^t$. Given a node v of the tree, an integer $i \in [3]$, and an operator $\circ \in \{+, -\}$, running the `TreeEval` function has the following effect on the registers:

- `TreeEval`(node v, integer $i \in [3]$, $\circ \in \{+, -\}$): assuming the original values of the registers is (r_1, r_2, r_3), after running this function, we will have

$$R_i \leftarrow r_i \circ val_v \qquad \text{and} \qquad R_j \leftarrow r_j \text{ for } j \neq i. \quad (5)$$

We go over the definitions and high level ideas of the algorithm first and then proceed to formalize it.

Definitions and High Level Ideas

We follow a similar strategy as in `LinearEval` by "linearizing" the functions $f_v : \{0,1\}^t \times \{0,1\}^t \to \{0,1\}^t$, one output bit at a time, using multi-linear extensions and polynomial interpolation introduced earlier. Firstly, define, for any $j \in [t]$,

$$f_{v,j} : \{0,1\}^{t+t} \to \{0,1\} \quad \text{where } f_{v,j}(xy) \text{ returns the } j\text{-th bit of } f_v(x,y).$$

Consider the multi-linear extension of $f_{v,j}$ over $\mathbb{F}$ denoted by

$$F_{v,j} : \mathbb{F}^{t+t} \to \mathbb{F} \qquad \text{(the multi-linear extension of } f_{v,j}).$$

Moreover, there are (at least) three operations we can do on the registers that allows us to later on bring back their state to their original form:

- $R_j \leftarrow z \cdot R_j$ for any $z \neq 0$ because we can simply run $R_j \leftarrow z^{-1} \cdot R_j$ to reverse this, given any $z \neq 0$ has a unique inverse in the field $\mathbb{F}$.

- $R_j \leftarrow R_j \pm val_{v0}$ or $R_j \leftarrow R_j \pm val_{v1}$ by recursively running `TreeEval` to obtain this and running it again with the other operator to reverse it.
- $R_j \leftarrow R_j \pm \alpha \cdot F_{v,i}(R_\ell, R_k)$ for distinct values of $j, \ell, k \in [3]$, $i \in [t]$, and $\alpha \in \mathbb{F}$, by calculating $F_{v,i}$ using the formula of the multi-linear extension and input access to $f_{v,i}$.

Our goal is thus to add val_v to, say, R_1, using a combination of these operations. To proceed, we need one more definition. Let (r_1, r_2, r_3) be the initial values of the registers (R_1, R_2, R_3). For a fixed $j \in [t]$, consider the single-variate polynomial

$$g_j : \mathbb{F} \to \mathbb{F} \quad \text{such that} \quad g_j(z) := F_{v,j}(z \odot r_2 + val_{v0}, z \odot r_3 + val_{v1}),$$

where $\odot : \mathbb{F} \times \mathbb{F}^t \to \mathbb{F}^t$ multiplies z to each of the t-indices of r_2, r_3 separately, i.e.,

$$z \odot (r_{2,1}, \cdots, r_{2,t}) = (r_{2,1} \cdot z\ ,\ \cdots\ ,\ r_{2,t} \cdot z).$$

Now, notice two things:

- All we want is to compute $R_1 \leftarrow R_1 + \sum_{j=1}^{t} 2^{j-1} \cdot g_j(0)$ to place $f_{v,j}(val_{v0}, val_{v1})$ in the j-th bit of R_1.
- But, computing $g_j(0)$ directly involves computing $R_2 \leftarrow z \cdot R_2 + val_{v0}$ (and similarly for R_3) for $z = 0$ which is not a reversible operation. Nevertheless, we can compute any other $g_j(z)$ for $z \neq 0$ in a similar manner (we will see this formally soon).

Here is where polynomial interpolation comes to our rescue. By Eq (4), we know that

$$g_j(0) = \sum_{z=1}^{deg(g_j)+1} c_z \cdot g_j(z) \tag{6}$$

where $deg(g_j)$ is the degree of the polynomial g_j and we simply picked the points $[deg(g_j)+1]$ as the interpolation points (the choice of the points is completely arbitrary and can be anything non-zero), and $c_1, \ldots, c_{deg(g_j)+1}$ are only functions of the points $\{0\} \cup [deg(g_j)+1]$ all of which are fixed and input-independent once we bound $deg(g_j)$. But bounding the degree is also pretty easy since

$$\begin{aligned} g_j(z) &= F_{v,j}(z \odot r_2 + val_{v0}, z \odot r_3 + val_{v1}) \\ &= \sum_{(a_1,\ldots,a_{2t}) \in \{0,1\}^{2t}} f(a_1, \ldots, a_{2t}) \cdot \prod_{i=1}^{2t} \left(a_i \cdot (z \cdot r_{*,i} + (val_{v*})_i) + (1-a_i) \cdot (1 - (z \cdot r_{*,i} + (val_{v*})_i)\right). \end{aligned}$$

(by the definition of $F_{v,j}$ Eq (3) where $* = 2$ for $i \leqslant t$ and $* = 3$ for $t < i \leqslant 2t$)

Specifically, this means that the largest power of z in $g_j(z)$ is $2t$ and thus $deg(g_j) = 2t$ as well.

We now have all the tools needed to formalize the algorithm.

Formalization and the Analysis

The algorithm is formally as follows. We again let (r_1, r_2, r_3) be the initial values of registers (R_1, R_2, R_3). This definition fixes the definition of the polynomial $g_j(\cdot)$ for any $j \in [t]$ also (as defined earlier).

Algorithm 3. `TreeEval`(node v, integer $i \in [3]$, $\circ \in \{+, -\}$): *(implements the transition in Eq (5))*

```
1. If $v$ is a leaf-node, update $R_i \leftarrow r_i \circ val_v$.
2. From hereon, for simplicity, we assume $i = 1$; other cases can be handled analogously.
3. For $j = 1$ to $t$ and $z = 1$ to $2t + 1$:
    (a) Update $R_2 \leftarrow z \odot R_2$ and $R_3 \leftarrow z \odot R_3$.
    (b) Run TreeEval($v0, 2, +$) and TreeEval($v1, 3, +$).
    (c) Update $R_1 \leftarrow R_1 \circ 2^{j-1} \cdot c_z \cdot F_{v,j}(R_2, R_3)$ where $c_z$ is the coefficient of $g_j(z)$ in Eq (6).
    (d) Run TreeEval($v0, 2, -$) and TreeEval($v1, 3, -$).
    (e) Update $R_2 \leftarrow z^{-1} \odot R_2$ and $R_3 \leftarrow z^{-1} \odot R_3$.
```

It is immediate to verify that after each iteration of the inner for-loop, values of R_2 and R_3 are reverted back to r_2 and r_3, respectively. So, the main part is to figure out what happens to r_1. The following lemma is the key part of the proof.

Lemma 3. *For any $j \in [t]$, let $r_1(j)$ denote the initial value of register R_1 at the beginning of the iteration j of the for-loop. Then, at the end of the iteration, we have*

$$R_1 = r_1(j) + 2^{j-1} \cdot f_{v,j}(val_{v0}, val_{v1}).$$

Before proving this lemma, let us see how it finalizes the proof. For the final value of R_1, we have,

$$\begin{aligned} R_1 = r_1(t+1) &= r_1 + \sum_{j=1}^{t} 2^{j-1} \cdot f_{v,j}(val_{v0}, val_{v1}) \\ &\qquad \text{(by writing the sum as a telescoping value and using Lemma 3)} \\ &= r_1 + f_v(val_{v0}, val_{v1}) \\ &\qquad \text{(since } f_{v,j}(val_{v0}, val_{v1}) \text{ is in } \{0,1\} \text{ and multiplying it by } 2^{j-1} \text{ puts it in the } j\text{-th digit in } \{0,1\}^t\text{)} \\ &= r_1 + val_v, \end{aligned}$$

which is the desired value. Thus, it remains to prove Lemma 3.

Proof of Lemma 3. We have

$$f_{v,j}(val_{v0}, val_{v1}) = F_{v,j}(0{\cdot}r_2{+}val_{v0}, 0{\cdot}r_3{+}val_{v1}) = g_j(0) = \sum_{z=1}^{2t+1} c_z g_j(z) = \sum_{z=1}^{2t+1} c_z{\cdot}F_{v,j}(z{\odot}r_2{+}val_{v0}, z{\odot}r_3{+}val_{v1});$$

here, the first equation holds because $F_{v,j}$ is a multi-linear extension of $f_{v,j}$ and val_{v0}, val_{v1} are Boolean value; the second holds by the definition of g_j, the third holds by Eq (6) using polynomial interpolation of $g_j(0)$ using the polynomial's values in $[2t+1]$, and the last one is again by the definition of $g_j(z)$.

By induction (on the depth of the tree when running `TreeEval`), in iteration z of the inner for-loop, when running the update on R_1, values of R_2, R_3 are $z \odot r_2 + val_{v0}$ and $z \odot r_3 + val_{v1}$, implying that in iteration z, the algorithm adds $2^{j-1} \cdot c_z \cdot F_{v,j}(z \odot r_2 + val_{v0}, z \odot r_3 + val_{v1})$ to R_1. Summing up over all choices of z concludes the proof. □

Finally, we also need to analyze the space complexity of the algorithm.

Lemma 4. `TreeEval` *on a tree of depth d with t-bit values can be implemented in $O((d + t) \cdot \log t)$ space.*

Proof. Firstly, the algorithm uses 3 registers with entries in $\mathbb{F}^t$ and thus each requiring $O(t \log t)$ bits. Besides this, we need to evaluate the space needed for its recursion.

Two non-trivial steps in the algorithm are computing the coefficients c_z for computing the multi-linear extension $F_{v,j}(R_2, R_3)$.

For the former case, as stated in Eq (4), these coefficients are only functions of the numbers $\{0\} \cup [2t+1]$ and are input independent. By Fact 2, since the total bit complexity of the matrix involving these coefficients is $t \log t$, we can calculate each bit of c_z in $\text{poly}\log t = o(t)$ space and subsequently compute and store all of c_z in another $O(\log t)$ space. Note that the space we need for calculating c_z is "local" to this step and for that we can for instance use a temporary global register once across all recursive calls.

For the latter, computing the multi-linear extension using Eq (3) involves going over all the 2^{2t} Boolean values in the sum using $O(t)$ bits of space and for each one additionally using $O(\log t)$ bits to calculate the entire multiplicative term, so overall this step can be implemented in $O(t)$ bits as well. Again, this space is "local" and can be done using a temporary global register.

Finally, to implement the recursive calls, we also need to know which line of code the algorithm currently is which can be specified by writing down j and z in $O(\log t)$ space plus another $O(1)$ step for the inner instruction. This means that at each level of the recursion, we need to store $O(\log t)$ bits and since the depth of the recursion is d, we need $O(d \log t)$ space in this case.

Overall, the algorithm can be implemented in $O((d+t)\log t)$ space. □

This concludes the proof of Theorem 1.

Remark. The tree evaluation problem was first introduced by [CMW+12] as a candidate for of separating logspace from polytime complexity classes, i.e., proving that $\mathsf{L} \neq \mathsf{P}$. Letting n denote the input size in the tree evaluation and setting $t = d$, we get that both $t, d = \Theta(\log n)$. It is easy to see that we can immediately solve tree evaluation in $\text{poly}(n)$ time but the obvious algorithm for this problem requires $O(\log^2 n)$ space. Based on this (and some lower bounds in weaker models of computation), [CMW+12] conjectured that tree evaluation cannot be solved in $o(\log^2 n)$ space, proving which separated L from P. which constitute an amazing milestone in complexity theory. The breakthrough work of [CM24] refutes this conjecture in a very strong sense, showing that tree evaluation can be solved in $O(\log n \cdot \log\log n)$ space. It remains an interesting open question to fully close this gap and achieve an $O(\log n)$ space algorithm for this problem (or of course prove that is not possible although that seems a lot less likely now than what was thought at the time of the conjecture of [CMW+12]).

3 Simulating Time with Space on Multi-Tape Turing Machines

We now briefly sketch an interesting application of Theorem 1 due to [Wil25] for simulating t-time algorithms in nearly $\sqrt{t}$ space (recall that it is trivial to simulate a t-time algorithm in $O(t)$-space since in that much time, the algorithm cannot touch more than t cells in the memory).

An important caveat is that for this result to hold, we need to work with notions of time and space on a more restricted computational model than the RAM model we introduced in the last lecture (and solely focus on in this course, beside this section, when talking about uniform computation).

Multi-tape Turing Machines. We will be working with two-tape Turing Machines instead of RAM machines for the rest of this lecture (for our purpose, power of two tape vs constant tape machines are equivalent). Roughly speaking, in this model, we have two tapes, where one of them holds the input initially. There are also two *heads* of the machine one on each tape. In each time step, the machine can read/write from the position of its current head and move each of the heads to left or right for one cell. The moves of the machine are dictated by a constant size state-space among which are *start* state, *accept* state, and *reject* state. Note that the main difference between this model and RAM machines we formally defined in the last lecture is that the heads of this machine can only move sequentially; for instance, if the first head is in position and wants to read position k next, it is going to take the machine $O(k)$ time steps to do so, unlike on RAM machines which this can be done in $O(1)$ time.

The time complexity and space complexity of the algorithm can be defined similarly to RAM machines using the number of moves in an execution (for time) and the largest index of the cell touched on either of the tapes (for space). For any functions $t(n) : \mathbb{N} \to \|N$ and $s(n) : \mathbb{N} \to \mathbb{N}$, we define

- $\mathsf{Time}^\star(t(n))$: set of all languages in $\{0,1\}^*$ decidable by a two-tape Turing Machine with time complexity $t(n)$ on inputs of length n;
- $\mathsf{Space}^\star(s(n))$: set of all languages in $\{0,1\}^*$ decidable by a two-tape Turing Machine with space complexity $s(n)$ on inputs of length n;

(We use the $\star$ in these definitions to distinguish them with our standard definitions in the last lecture).

The following is a slightly weaker (by polylogarithmic factors[2]) version of the main result of [Wil25].

Theorem 5. *For every function $t(n) \geqslant n$:*

$$\mathsf{Time}^\star(t(n)) \subseteq \mathsf{Space}^\star(\widetilde{O}(\sqrt{t(n)})).$$

The idea behind the proof is to start with a two-tape machine M, write its execution in T steps as an instance of tree evaluation with $O(\sqrt{T})$-depth and $O(\sqrt{T})$-bit values, and then apply Theorem 1 to this tree to compute the answer in only $\widetilde{O}(\sqrt{T})$ space.

The first step to do this is to use a classical result for turning multi-tape Turing machines **oblivious**: this means that on every input x of length n, the head movements of the Turing machines are only a function of n and *not* x (e.g., think of the heads always going from left to right and then looping back again and again).

Proposition 6 (cf. [HS66])**.** *For every $t(n)$-time two-tape Turing Machine, there exists an equivalent oblivious two-tape Turing machine with time complexity $t'(n) = O(t(n)\log(t(n)))$. Moreover given n and any $i \in [t'(n)]$, the position of the heads of M' at time i can be computed in* $\text{polylog}(t'(n))$ *time.*

Equipped with Proposition 6, we can focus on proving Theorem 5 for the oblivious two-tape Turing machines prescribed in Proposition 6. To continue, we need one more definition.

Definition 7. For an oblivious two-tape Turing Machine M and input length n running in $t(n)$ time, we define the following **computation DAG** G:

1. There are $t(n)$ vertices in the DAG corresponding to the $t(n)$ time steps taken by the algorithm.
2. The edges of the DAG can only go from a vertex u to v, if $u < v$ (i.e., u corresponds to an earlier time step than v). Hence, G is a DAG.
3. There is an edge (u, v) in G iff: (1) the position of heads of M at time u and time v intersect, and (2) v is the smallest index for which this is true (i.e., for any $w : u < w < v$, the heads of M at time u and w are disjoint).

Note that since M is oblivious, the computation DAG is only a function of the input length and not the input. Moreover, given vertices/indices $u, v \in [t(n)]$, we can determine if (u, v) is an edge in this DAG in

[2]We use the standard notation that $\widetilde{O}(f) := O(f \cdot \text{poly}\log(f))$ when stating the result.

$\widetilde{O}(t(n))$ time since the position of heads of the machine can be computed in $|v - u| \cdot \text{polylog}\,(t(n))$ time by the guarantees of Proposition 6.

We are now ready to prove Theorem 5.

Proof of Theorem 5. Let M be an oblivious two-tape Turing Machine of the type in Proposition 6 running in $t(n)$ time on inputs of length n. Let G be the computation DAG of M on n-size inputs. Partition the cells in the two tapes of M into *consecutive* **blocks** of length B for some parameter B to be determined later. Similarly, partition the time steps/vertices of G into *consecutive* **phases** of the same length B. We have the following simple observation.

Observation 8. *When running M on any input of length n, during a fixed phase (of time), at most four blocks (of the tapes) are visited by the heads of the machine M.*

Proof. During a phase, the head of the machine on each tape can move at most B cells in each direction and thus cannot move beyond a single block on each tape, meaning at most two blocks are visited on each tape and at most four overall. □

Observation 8 is the main place where restriction of multi-tape Turing Machines (compared to RAM machines) are being exploited[3]. Moreover, notice that to compute the updated content of the at most four visited blocks in a single phase of the algorithm, we only need to know the content of blocks also (we can just simply simulate M on these blocks as it never reads/write to any block outside here).

For any phase p and any of the four (tape) blocks b in this phase, define a function:

$$f_{p,b} : \{0,1\}^B \times \{0,1\}^B \times \{0,1\}^B \times \{0,1\}^B \to \{0,1\}^B \,,$$

which takes as input the content of the four tape blocks involved in phase p and outputs the content of the tape block p at the end of phase p. Note that we can evaluate $f_{p,b}$ using $O(B)$ space and $O(B)$ time by simulating M on the input content of these blocks during the phase.

Additionally, the computation DAG tells us exactly where the input of $f_{p,b}$ comes from the output of which $f_{p',b'}$: whenever there is an edge (p, p') in the DAG. We can thus "unfold" this DAG into a tree T with root corresponding to the last phase and having multiple vertices for each phase of the algorithm to avoid creating cycles. See Figure 1 for an illustration of unfolding a DAG into a tree.

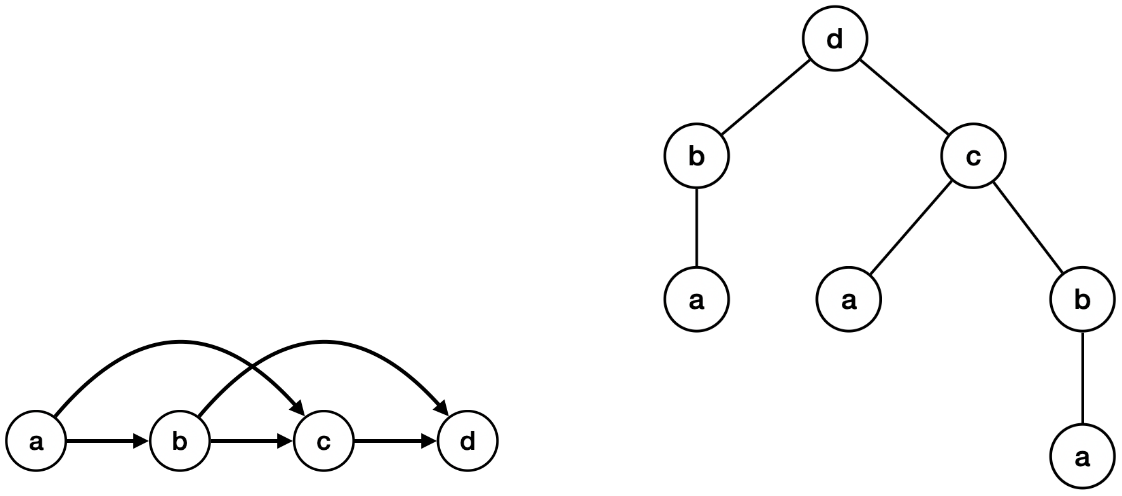
Note that since M is oblivious and the computation DAG is fixed, this tree is input independent and can be created implicitly by determining the child nodes of any node in the tree in $\text{polylog}\,(t(n))$ space.

The problem at this point is that we have a tree with:

- fan-in at most four;
- B-bit values and functions at each level of the tree that can be computed in $O(B)$ space and time;
- depth at most $t(n)/B$ (as each level of the tree goes down at least one phase in the computation).

This is almost an instance of the tree evaluation problem with two minor differences: the degree of vertices is at most 4 instead of exactly 2, and that child-nodes of a node may appear at different levels in the tree. Despite this, it is immediate to verify that the proof of Theorem 1 works verbatim for this problem as well. Thus, we can solve this tree evaluation problem also in $O((d+t)\log t)$ where $d := t(n)/B$ is the depth of the tree and $t := B$ is the size of each value in the tree. Thus, we can simulate the multi-tape Turing Machine M

[3]While an analogue of Proposition 6 is not known for RAM machines, it turns out obliviousness is not that crucial for the proof and can be circuimvanted using an "instance-wise obliviousness" by simply running the algorithm for all choices of computation DAG until we find the right one for the specific input; see [Wil25].



(a) An original DAG. (b) The unfolded tree corresponding to the DAG.

Figure 1: An illustration of unfolding a DAG into a tree.

using $O((t(n)/B+B)\log B)$ space[4]. Setting $B=\sqrt{t(n)}$, we get an algorithm that runs in $O(\sqrt{t(n)}\cdot\log t(n))$ space.

Finally, since we also only lost a logarithmic factor in Proposition 6, we can conclude that

$$\mathsf{Time}^{\star}(t(n)) \subseteq \mathsf{Space}^{\star}(\widetilde{O}(\sqrt{t(n)})),$$

as desired. □

References

[CM24] James Cook and Ian Mertz. Tree evaluation is in space $O(\log n\cdot\log\log n)$. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1268–1278. ACM, 2024. 1, 3, 8

[CMW+12] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *ACM Trans. Comput. Theory*, 3(2):4:1–4:43, 2012. 1, 8

[Gol24] Oded Goldreich. On the cook-mertz tree evaluation procedure. *Electron. Colloquium Comput. Complex.*, TR24-109, 2024. 3

[HS66] F. C. Hennie and Richard Edwin Stearns. Two-tape simulation of multitape turing machines. *J. ACM*, 13(4):533–546, 1966. 9

[Vio25] Emanuel Viola. Mathematics of the impossible: The complexity of computation. *2023-2025 Emanuel Viola, to be published by Cambridge University Press*, 2025. 1

[Wil25] R. Ryan Williams. Simulating time with square-root space. In Michal Koucký and Nikhil Bansal, editors, *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025, Prague, Czechia, June 23-27, 2025*, pages 13–23. ACM, 2025. 1, 8, 9, 10

[4]It is worth pointing out that the solution to tree evaluation we presented was in the RAM model and not multi-tape Turing Machines. However, when talking about space and not time, the distinction between the two models is entirely inconsequential one can simulate one model in another using the same space complexity.