Simulating time with square-root space

Talk by Ryan Williams

Notes by Sanjana Das

April 8, 2025

§1 Introduction

The question we'll address in this talk is:

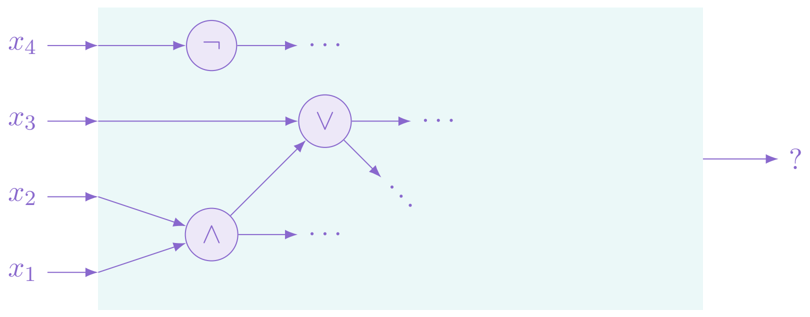
Question 1.1. How space-efficiently can one simulate time-efficient computations?

Specifically, suppose we have an algorithm $\mathcal{A}$ for some problem using $t(n)$ time; we want an algorithm $\mathcal{B}$ for the same problem using as little space as possible. Of course achieving $O(t(n))$ space is trivial (we can just use the same algorithm); we want to know if we can use less.

A canonical problem to try understanding the difficulty of this is the circuit evaluation problem (CVP).

Problem 1.2 (CVP)

- **Input:** A description of some logical circuit with s gates (of fan-in 2) and one output gate, and an input to evaluate it on.
- **Output:** Whether the circuit outputs 1 on the given input.



It takes roughly s time to solve CVP (in most interesting computational models). One might think that it requires $\Omega(s)$ space, to store the intermediate gate values — the natural solution is dynamic programming (where you go through the circuit and label the value at each gate as you go), and along the way you could easily pick up $\frac{s}{10}$ gates that need to be evaluated at once before you can get to the rest.

But in fact, you don't need $\Omega(s)$ space — it follows from results of PV 1976 and Borodin 1977 that CVP can actually be solved in $O(s/\log s)$ space. This algorithm has exponential runtime, but it saves over the naive space bound. And more generally, HPV 1975, PR 1981, and HLMW 1986 showed that *any* problem that can be solved in time t can be solved in space $t/\log t$ (the original result was for multi-tape Turing machines, but it was later extended to basically all interesting models). So you can always shave $\log t$ off the trivial bound.

The way these results work is a 'pebbling' approach (which we'll discuss soon). We've known since 1979 that this approach requires $t/\log t$ space; so as far as people knew, you could shave $\log t$, but you probably couldn't do better than that.

The model we'll work with is multi-tape Turing machines (MTMs): We have a Turing machine with a finite control and k tapes, each with its own tape-head (where k is a constant). The input arrives on tape 1. On each step, each one of the tape-heads can read and write at its current cell, and move left, right, or stay still. (The tape heads could be in very different places — one could be very far down its tape and one could be at the beginning — and they can all read and write and do head-movements in one step.)

Definition 1.3. We define $\mathsf{TIME}[t]$ as the set of problems solvable in $O(t(n))$ time on a multi-tape TM.

This is an old model, but it's very robust. For example, CVP, sorting, FFT, and all your good subcubic time matrix algorithms — all of these can be implemented on multitape TMs with $\mathsf{polylog}$ overhead (in fact, they can be implemented on 2-tape TMs — any multi-tape TM can be implemented with just two tapes, with $O(\log t(n))$ overhead). In fact, it's a major open problem to find *any* problem solvable in RAM in $O(n)$ time, but not on MTMs in $O(n \cdot \mathsf{polylog}(n))$ time — so as far as we know, MTMs are as good as anything else, up to $\mathsf{polylog}$ factors. Despite 60 years of research, there's been no significant lower bounds on them other than those obtained by the time hierarchy themselves, which makes the following theorem shocking.

Theorem 1.4

For all $t : \mathbb{N} \to \mathbb{N}$ with $t(n) \geq n$, we have $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[\sqrt{t \log t}]$.

It's hard to convey how deeply disturbing this is — Ryan would have bet on the conjecture that $\mathsf{TIME}[t] \not\subseteq \mathsf{SPACE}[t^{1-\varepsilon}]$. This is wild because he's a lower bound skeptic in general, but he would have believed this one is true. And this conjecture was actually used in some prominent works in the literature — Sipser showed that this conjecture plus some expanders (which are now known to exist) would imply $\mathsf{P} = \mathsf{RP}$. Thankfully, though, we now know that we only need *circuit* lower bounds for derandomization, not space lower bounds on time (due to NW 1994 and IW 1997).

Here are some more consequences of this.

Corollary 1.5

The circuit evaluation problem is in $\mathsf{SPACE}[\sqrt{n} \cdot \mathsf{polylog}(n)]$.

Corollary 1.6

Size-s circuits have branching programs of subexponential (specifically, $2^{\sqrt{s} \cdot \mathsf{polylog}(s)}$) size.

(A branching program is a decision tree which is a DAG.)

Corollary 1.7

For space-constructible $s(n) \geq n$, we have $\mathsf{SPACE}[s] \not\subseteq \mathsf{TIME}[s^2/(\log s)^2]$.

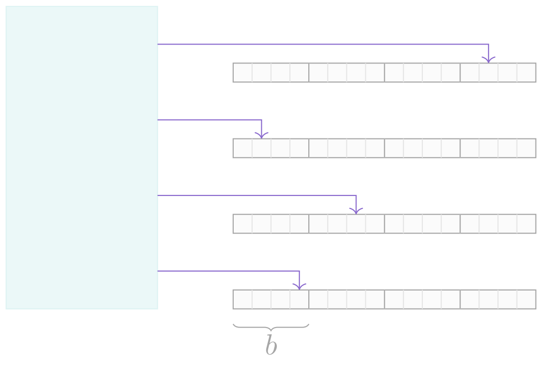
This one is very unsurprising — we believe $\mathsf{SPACE}[s] \not\subseteq \mathsf{TIME}[s^k]$ for all k — but it follows from a very surprising thing (Theorem 1.4, using diagonalization and hierarchy theorems).

Remark 1.8. If we could improve the square root and get that $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[t^{\varepsilon}]$ for all ε, then the exponent on the right-hand side would keep going up, and you'd get $\mathsf{P} \neq \mathsf{PSPACE}$. This would be the weirdest way ever to separate P from PSPACE, but maybe the world is weird.

§2 A computation graph

Some conceptual tools we use come directly from prior work, so we'll start by discussing that work; here we want to show $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[t/\log t]$.

Suppose we're given a MTM $\mathcal{M}$ and an input x of length n. First, we partition the k tapes of $\mathcal{M}$ into blocks of b cells each. (Throughout the talk, you should think of b as $\sqrt{t}$. For simplicity we'll assume $t(n) \geq n^2$, so that the input x fits in one block b. If this isn't true you could split the input into blocks; it's just a bit more technical.)



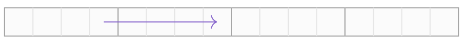
Then we define a *computation graph* $\mathcal{G}_{\mathcal{M},x}$. This will be a DAG with nodes named 0, 1, ..., t/b, where each node represents a time-interval of b steps of time — node 0 represents the initial configuration, 1 represents the first b time-steps, 2 represents the next b time-steps, and so on.

Edges in this graph are basically supposed to record how the information computed in different time-intervals relates — intuitively, for intervals $i < j$, we want to put in an edge $i \to j$ if the information computed in interval i is needed to compute information in interval j. To be more precise:

- For all i, we draw an edge $(i-1) \to i$. The reason for this is that we need to know the last state of interval $i-1$ to know the starting state for interval i.
- For $i < j$, we draw an edge $i \to j$ if there's some tape block which is visited in interval i and again visited in interval j, but wasn't visited in intervals $i+1, \ldots, j-1$. So you were in this tape block in interval i; then you went away and were off doing something else in other tape-blocks; and when you got to interval j, you finally entered this tape block again. The point is that we'll potentially need what we computed on this tape-block in interval i in order to compute in interval j (now that we've come back, we need to know what was in this tape-block).

Fact 2.1 — During each time-interval, each tape has at most 2 accessed blocks.

Proof. Each tape-head moves a distance of at most b (a time-interval consists of b steps), and a line of length b intersects at most 2 blocks. □



This means every node in $\mathcal{G}_{\mathcal{M},x}$ has constant in-degree:

Claim 2.2 — We have $\text{indeg}(i) \le 2k+1$ for all i.

Proof. There's k tapes, and each accesses at most 2 blocks (and the $+1$ is from $i-1$). □

Now we're going to define strings that tell us what's the information computed in each interval. First, we define $\mathsf{content}(0)$ as the initial configuration of $\mathcal{M}$ on x, which is $O(n)$ bits (we think of $t(n)$ as large enough that $\sqrt{t(n)} \ge O(n)$). Then for all $i > 0$, we define $\mathsf{content}(i)$ as the 'information computed in interval i' — more specifically, we include the state, the head-positions of all k tape-heads at the end of the interval, and all the contents of the tape blocks we accessed during this interval. This takes $O(b)$ bits to encode.

Claim 2.3 — Suppose that for some j, I give you $\mathsf{content}(i)$ for all i with an edge $i \to j$. Then you can compute $\mathsf{content}(j)$ in $O(b)$ time.

Proof. This is because I'm telling you all the information you need to compute interval j — I'm telling you the state to start in and all the contents of the tape blocks you'll try to access. So you can simulate the machine directly for b steps, which will take $O(b)$ time. □

So we have this graph $\mathcal{G}_{\mathcal{M},x}$ where every node can be evaluated in $O(b)$ time given all its predecessor nodes, and we want to determine the content of the last node — this has the state at the end (accept or reject). So the whole game now is to try to evaluate this computation graph.

§3 The pebbling approach

The result $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[t/\log t]$ uses a similar setup . (They think of b as $\sqrt[3]{t}$, but this isn't super important — you just need $\log b$ to be a constant times $\log t$, and you need it to be significantly smaller than t. For various reasons, they apply Savitch at some point, so they need the cube root. But that's not super important for this discussion; the point is just that b should be small but not too small.)

Now we have a problem about DAGs:

Problem 3.1 (Pebble game)

You're given a v-node DAG (and its topological ordering). You can put a pebble on the source; and if you have pebbles on all i which have an edge to j, then you can pebble j. You can also remove pebbles at any time.

How many pebbles are needed to put a pebble on the last node?

Here, the vertices of the DAG correspond to our time-intervals $0, \dots, t/b$ (in that order). Putting a pebble on the source corresponds to computing the initial configuration, which we can do whenever we want. The second rule tracks the observation that given $\mathsf{content}(i)$ for all i with $i \to j$, you can compute $\mathsf{content}(j)$. And we want to remove pebbles because we want a space-efficient simulation, so we don't want there to be too many pebbles.

This question has been very rigorously studied; and for DAGs of constant in-degree, the answer is $v/\log v$. (There are graphs on which you can't do any better.)

But the point is, suppose we have such an upper bound, and suppose the strategy for pebbling can be computed efficiently (in small space). Then we have this computation graph with $O(t/b)$ nodes, and we only have to store $(t/b)/\log(t/b)$ content-strings at any time; each of these strings is $O(b)$ bits, so this takes $O(t/\log(t/b))$ bits. So as long as b is roughly t^{ε} (for some constant ε), this will be $O(t/\log t)$.

So that's the super high-level idea of the prior work (though we're not going to go into how you win the pebble game, which is quite complicated).

§4 The tree evaluation problem

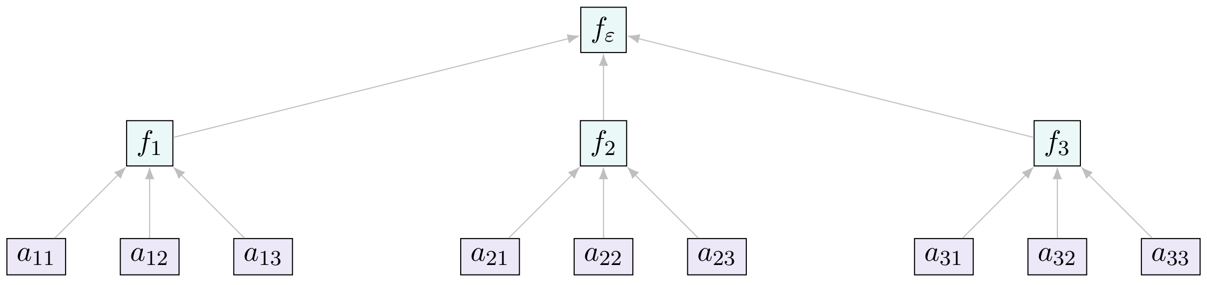
To discuss how we get a low-space algorithm, we'll introduce the tree evaluation problem (TreeEval), which was defined by BCMSW 2009.

Problem 4.1 (TreeEval)

Fix parameters $d, b, h > 0$ (which are integers).

- **Input:** a d-ary tree of height h, where every node is named by a string over $\{1, \dots, d\}$ of length at most $h - 1$. Each leaf ℓ is labelled by a b-bit string $a_\ell \in \{0, 1\}^b$, and each inner node u is labelled by a function $f_u : \{0, 1\}^{db} \to \{0, 1\}^b$.
- **Goal:** Suppose that for each node u (with children $u1, \dots, ud$), we define $a_u = f_u(a_{u1}, \dots, a_{ud})$. Find the value of the root.

So as a picture, we start with a function f_ε at the root (where ε denotes the empty string); then below it we have $f_1, \dots, f_d$; and at the bottom we have leaves labelled $a_{111\dots1}, \dots, a_{ddd\dots d}$. (You can imagine these functions are given as tables of length 2^{db}.)



And each node v takes a b-bit string from each of its d children, computes f_v on those db bits, and sends this value up to its parent.

(If $d = 1$, then this is just Boolean formula evaluation.)

Question 4.2. How much space is needed to solve TreeEval?

There's a simple algorithm that the authors studied, and they wanted to know if it's optimal:

Algorithm 4.3

Suppose we want to compute a_v for some node v.

- If v is a leaf, just return its value.
- If not, then recursively compute the values of all its children (i.e., $a_{v1}, \dots, a_{vd}$), compute $a_v = f(a_{v1}, \dots, a_{vd})$, and then erase the values of these children.

You can imagine doing this in a depth-first way, where you recurse as soon as possible when you need to. This takes $O(hdb)$ space — you have a recursion stack of height h, and you're storing db bits on each level (you could be storing $d - 1$ b-bit values, for all the children except the one you're currently recursing on).

ε	a_1	a_2	a_3
4	a_1		
24	a_1	a_2	
$\vdots$		$\vdots$	

This takes $(\log n)^2$ bits for an input of length n.

BCMSW showed lots of interesting lower bounds in restricted settings, and they conjectured TreeEval wasn't in LOGSPACE. You can solve it in P (just by storing everything that you compute), but they conjectured that it's not in LOGSPACE; so studying TreeEval was a way of possibly separating LOGSPACE from P.

It was a big surprise last year when Cook and Mertz showed that you can actually do much better — there's a surprisingly space-efficient algorithm for TreeEval.

Theorem 4.4 (Cook–Mertz 2024)
We can solve TreeEval in $O(h \log db + db)$ space.

So instead of a stack of height h with db bits on each level, you just need a stack of height h with $\log db$ bits on each level. And you also have d b-bit registers, and somehow you reuse these over and over at every level. Before, you had an enormous number of registers (you had h sets of d b-bit registers); but here you somehow replace them with *one* set that you keep overriding and overriding.

At a high level, they produce an algorithm that XORs the value of any node u into existing space (i.e., one of these b-bit registers), assuming an oracle that can XOR the value of any child of u into existing space. So if you have an oracle that XORs the value of any child into the registers, I'll give you a way of XORing the function evaluated on all those children.

As a bit of intuition for what's happening in these $\log db$ bits, consider the low-degree extension $\widetilde{f}_u$ of f_u (you can even imagine the multilinear extension; this won't give you exactly the right bound, but it'll get close). This multilinear extension agrees with the actual function on $\{0,1\}$-valued points; so we want to XOR the value of $\widetilde{f}_u$ (on the children of u) into existing space.

Roughly, what happens is that we evaluate it over $\mathsf{poly}(db)$ values, each of which involves the values of your children combined with the content previously stored in the registers. And you somehow use polynomial interpolation to cancel out the old content and get just what you want, by summing over these $\mathsf{poly}(db)$ values. (The height-h stack of $\log db$ bits is used to store which of these $\mathsf{poly}(db)$ things you're currently on.)

§5 From the computation graph to tree evaluation

First we'll outline how to show that $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[\sqrt{t} \cdot \log t]$ (with the $\log t$ outside rather than inside the square root), which is still shocking.

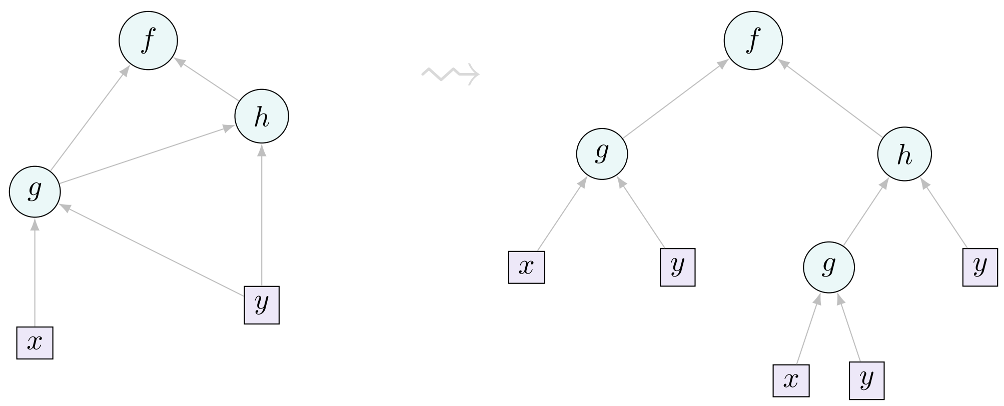
The idea is that given a time-t TM $\mathcal{M}$ and input x, we want to reduce to an exponential-sized TreeEval instance. We're not going to hold this TreeEval instance in memory (it's too big); we'll just access pieces of it in small space as we go (we'll access nodes and edges in some implicit way).

We'll start by assuming that we know $\mathcal{G}_{\mathcal{M},x}$ (either that it's stored in memory, or that we can compute its edges efficiently — that we can compute whether $i \to j$ is an edge in small space, so that given a node j, we can always recover its predecessors). (We'll discuss how to fix this assumption later.)

We'll set d (the fan-in of the tree) to $2k+1$, and h (the height of the tree) to $t/b+1$. And we'll define b so that all the content-strings in our computation graph are of length at most b. (This is slightly different from the b we used for block length when defining the computation graph; but they differ by just a constant factor.) (Recall that b is the number of bits in the value at each node; and we want to fit our content-strings in the computation graph into these b-bit values.)

We have to extend Cook–Mertz to work for trees where each node has *at most d* children and the tree has height *at most h*; this is easy (their algorithm is recursive, and you can just stop if you reach a leaf early).

And then there's a folklore result that a Boolean circuit of depth h can be transformed into a Boolean formula of depth h. The idea is that given a circuit, you can create a formula by starting from the output and working backwards. This will give you a tree-like circuit of the same depth, where each gate here represents a path in the original circuit from some gate to the output.



This was proved for Boolean circuits and formulas, but it works just as well for circuits and formulas with b-bit values on the wires, which is exactly what we're looking at in the computation graph and in $\mathsf{TreeEval}$: Computing the content of a node in $\mathcal{G}_{\mathcal{M},x}$ is just evaluating a circuit over b-bit values ($\mathcal{G}_{\mathcal{M},x}$ is a DAG, so we can think of it as a circuit). And $\mathsf{TreeEval}$ can be thought of as evaluating a formula with depth h and fan-in d over b-bit values.

So if we know $\mathcal{G}_{\mathcal{M},x}$ (e.g., if we somehow have it in memory), then we can simply run the Cook–Mertz algorithm — our goal was to evaluate the last node in the computation graph. And if we know the graph, then we can unroll it into this tree and use Cook–Mertz to evaluate the content of the last node.

Cook–Mertz takes space $db + h \log db$. We set $h = t/b$ (and d is constant), so this is $O(b + (t \log b)/b)$. We set $b = \sqrt{t \log t}$ to minimize this, and we get $O(\sqrt{t \log t})$ space.

However, we haven't actually proved this bound because we don't yet know what the computation graph $\mathcal{G}_{\mathcal{M},x}$ is — all this assumes we can get our hands on it.

§6 Approach 1: Olivious TMs

Here's the easiest fix, which gets space usage $\sqrt{t} \cdot \log t$. The idea is that we want to program $\mathcal{M}$ so that we can calculate all the head positions quickly — if I know where all the head positions are at any point in time, then I can tell you if there should be an edge $i \to j$ (there's an edge if we're in some block at i, we go away, and we come back at j; and we can just calculate this). So if we can calculate all the head positions quickly in advance, then we're happy. (We really only need to be able to calculate the head positions in low *space*, but it turns out we can do it in low time as well.)

For this, we use the notion of *oblivious TMs*:

Definition 6.1. A TM $\mathcal{M}$ is oblivious if for all n and all inputs x of length n, the head movements of $\mathcal{M}$ on x only depend on n (and don't depend further on x).

Theorem 6.2

For any TM $\mathcal{M}$ running in $t(n)$ time, there is an equivalent two-tape oblivious TM $\mathcal{M}'$ running in $O(t(n)\log t(n))$ time. Furthermore, given n and i in binary, you can compute the head positions of $\mathcal{M}'$ on step i in $\mathsf{polylog}(t(n))$ time.

So I can tell you at any point in time what the head positions are, which means I can quickly figure out which tape blocks are accessed when, and then I can just compute $\mathcal{G}_{\mathcal{M}',x}$. But we lose a little, because time blew up by a log factor; this results in the slightly worse space bound of $\sqrt{t}\cdot\log t$.

But also, this is a black box; and we'd like a proof without black boxes. There's actually another way to do this which is more direct and gets the slightly better bound of $\sqrt{t\cdot\log t}$; and it has the nice property that if $\mathsf{TreeEval}$ is actually in $\mathsf{LOGSPACE}$, then we'd get $\mathsf{TIME}[t]\subseteq\mathsf{SPACE}[\sqrt{t}]$.

§7 Approach 2: Guessing the computation graph

The whole game now is to get our hands on the computation graph $\mathcal{G}_{\mathcal{M},x}$ — if we can do this, then we can compute its tree and run Cook–Mertz. The idea is to *guess* the computation graph and check that it's valid! (This is the 'nondeterminism paradigm.')

We can definitely do this in low space (by trying all possible computation graphs one at a time). But there are a few concerns. First, what if there's too many possible guesses? (We need some way of encoding where the edges go to describe possible guesses.) And we also have to be careful how we check the graph is valid. The Cook–Mertz procedure is really weird and you might not know the values of intermediate nodes, so you have to be careful.

But it turns out you can encode the computation graph very efficiently, in just $O(t/b)$ bits (where t/b is the number of nodes — so we're just using $O(1)$ bits per node). For each tape and each time-interval i, if we're starting at a certain tape block, I tell you which tape blocks adjacent to this one are accessed in interval i, with 0, -1, and $+1$ (corresponding to this tape block, the one to its left, and the one to its right). So we can have $\{-1,0\}$ (if this block and the one to its left are accessed), $\{0\}$, or $\{0,1\}$. And I can also tell you where the head ends up — how the tape block containing the head changes from this interval to the next — with another 0, -1, or 1.

And then we can check whether there's an edge $i\to j$ by keeping counters that store the tape block indices for every tape. These counters just take $O(\log t)$ space. To determine edges, we just want to know, if we're currently in some tape block, when do we leave it and when do we come back? So if we have counters that keep track of the tape block indices over all intervals, that's enough.

So this graph can be encoded in a very succinct way.

The next thing we need to do is figure out how to check:

Goal 7.1. Given a graph $\mathcal{G}$, check whether $\mathcal{G}=\mathcal{G}_{\mathcal{M},x}$.

We need to be careful here — we're using $\mathsf{TreeEval}$ in a black-box way, and the Cook–Mertz algorithm reuses space so much that the only thing we might actually know in the end is the value at the root.

There are several ways to try implementing this, but here's one: We're going to bake the checking into the functions in our $\mathsf{TreeEval}$ instance. The encoding of $\mathcal{G}$ asserts exactly which tape blocks are accessed during each interval i. And we'll define our $\mathsf{TreeEval}$ function for the node i so that its behavior depends on $\mathcal{G}$. The idea is that if I'm simulating $\mathcal{M}$ on the interval i, if a tape head ever tries accessing a block that $\mathcal{G}$ says it doesn't, or it doesn't access all the tape blocks that $\mathcal{G}$ says it does, then I just say FAIL (as the function output for that node). We also propagate FAIL's up the tree — if a node receives any input of FAIL from its children, then that node also outputs FAIL. So if there's any issue with $\mathcal{G}$, then the tree will output FAIL at the root. This means the resulting instance doesn't output FAIL at the root if and only if you got the right graph $\mathcal{G}$.

So we just go over all these possible graphs (i.e., all possible t/b-bit strings encoding some graph) and run TreeEval on each, until we get an Accept or Reject state (rather than FAIL) at the root; and that's the answer.

§8 Open problems

To finish, here's some open problems.

Question 8.1. Can we extend this to RAM models, i.e., show $\mathsf{TIME}_{\mathsf{RAM}}[t] \subseteq \mathsf{SPACE}[t^{1-\varepsilon}]$?

Question 8.2. Can we extend to parallel machines, i.e., show $\mathsf{TIME}[t] \subseteq \mathsf{ATIME}[t^{1-\varepsilon}]$?

If we could do this, it would imply superlinear time lower bounds for even more natural problems like QBF.

Question 8.3. Can we improve the exponent from $\frac{1}{2}$ to $\frac{1}{2} - \varepsilon$, i.e., show $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[t^{1/2-\varepsilon}]$?

This would actually resolve Questions 8.1 and 8.2 — the simulations needed to go between these models have a quadratic overhead, so if we can improve over $\frac{1}{2}$ for space, we'd get $1 - \varepsilon$ for those.

Question 8.4. If $\mathsf{P} = \mathsf{PSPACE}$, could you get a better simulation (e.g., $\mathsf{TIME}[t] \subseteq \mathsf{SPACE}[t^{\varepsilon}]$)?

Question 8.5. Are there barriers?

Remark 8.6. Ryan thinks he's very lucky that he didn't think about TreeEval until after Cook–Mertz — if he had found this argument first, he would have thought it was a barrier to TreeEval (that you can't solve TreeEval in low space, because if you could then you'd improve the 50-year-old simulation of $\mathsf{TIME}[t]$ in $\mathsf{SPACE}[t/\log t]$). So this is kind of disturbing.

§9 More about Cook–Mertz

Finally, here's a bit more detail on the Cook–Mertz TreeEval algorithm. (*Note: This part is from a separate talk Ryan gave at the Graph Simplification Reading Group on May 9, 2025.*)

In our case d is a constant; then you can without loss of generality assume $d = 2$. Let's pick a big field $\mathbb{F}$, with $|\mathbb{F}| = 2^q \geq 2b^2$. Recall that we defined

$$a_u = f_u(a_{u1}, a_{u2}),$$

where f_u takes in $2b$ bits and outputs b. We can break f_u up into b functions $f_{u,j}$ that output the jth bit (for each $1 \leq j \leq b$).

We'll discuss an easier version of Cook–Mertz where we look at the multilinear extension of $f_{u,j}$, defined by

$$\widetilde{f}_{u,j}(x_1, x_2) = \sum_{a_1, a_2 \in \{0,1\}^b} \chi_{a_1,a_2}(x_1, x_2) f_{u,j}(a_1, a_2),$$

where x_1 and x_2 are length-b vectors over $\mathbb{F}$, and χ_{a_1,a_2} is the function that outputs 1 on (a_1, a_2) and 0 everywhere else (so it's multilinear of degree $2b$). So now we're allowing $\{0,1\}$-valued inputs, but also inputs over this big field $\mathbb{F}$. This algorithm will actually get $(h+b)\log b$ space usage, where the extra $b \log b$ comes from this multilinear extension (if you use low-degree extensions correctly, you can get rid of it).

Here's the key lemma behind the whole thing.

Lemma 9.1

Let ω be a generator of $\mathbb{F}^\times$ (i.e., a primitive mth root of unity, where $m = 2^q - 1$). Let $y_1, y_2, z_1, z_2 \in \mathbb{F}^b$. Then for all polynomials P of degree less than m, we have

$$P(z_1, z_2) = -\sum_{i=1}^{m} P(\omega^i y_1 + z_1, \omega^i y_2 + z_2).$$

The way to think about this is to consider any monomial $x_1^k x_2^\ell$ in P and imagine expanding it; we'll get

$$z_1^k z_2^\ell + \omega^{ki} y_1^k z_2^\ell + \omega^{\ell i} z_1^k y_2^\ell + \omega^{(k+\ell)i} y_1^k y_2^\ell$$

(where the first term is the one we want, and the rest are extras). When we sum over i, the extra terms are going to have overall coefficients that look like $\sum_i \omega^i$ (or $\sum_i \omega^{ki}$ for some $0 < k < m$); and this is 0. So all the extra terms cancel out, and we're left with just the things we want.

Why is this useful for us? We said that Cook–Mertz reuses space over and over by XORing the current value into the existing space. Here we think of y_1 and y_2 as the existing space, which we can't control (it's the content of our register when we come in — worst-case space chosen for us). And we think of z_1 and z_2 as values that we can add into the registers.

We'll have $d+1$ registers (here $d = 2$, so we'll have three). We assume inductively that we can add the values a_{u1} and a_{u2} into any register we want; and we want to be able to do the same for a_u. So let's imagine we want to add a_u into the third register. Suppose that the registers started with y_1, y_2, and c, where these are some values not under our control (we want to end with them having y_1, y_2, and $a_u + c$).

$$\boxed{y_1} \qquad\qquad \boxed{y_2} \qquad\qquad \boxed{c}$$

We want to compute $a_u = P(a_{u1}, a_{u2})$; we'll do this by computing the right-hand side of Lemma 9.1. So we'll range over all $i = 1, \ldots, m$. We have y_1 and y_2 in the first two registers. We first multiply these by ω^i; and then we inductively add in a_{u1} into the first register and a_{u2} into the second (so now we have $\omega^i y_1 + a_{u1}$ and $\omega^i y_2 + a_{u2}$ in those two registers). Then we evaluate P on these two values, and add it to a running sum we're keeping in the third register. Then we uncompute in the first two registers (subtracting out a_{u1} and a_{u2} and dividing out ω^i, so that they go back to having y_1 and y_2).

In the end, we'll have added $P(\omega^i y_1 + a_{u1}, \omega^i y_2 + a_{u2})$ to the third register for each i, so that register now has $P(a_{u1}, a_{u2}) + c = a_u + c$ by Lemma 9.1, which is what we wanted.

(The only thing we need to record on our height-h recursion stack is which index i we're currently on; since $m = \mathsf{poly}(b)$, this takes $\log b$ bits.)

Remark 9.2. In this algorithm, you've never actually computed the values of the internal nodes — everything is XORed and smashed together. This is why we had to do the propagate-FAIL thing — if we run Cook–Mertz we can't look at intermediate node values to figure out whether our guessed graph $\mathcal{G}$ has issues (because it doesn't compute those values), so we have to make sure these issues show up at the root.