Tree Evaluation in $O(\log n \log\log n)$ Space

Notes by Linus Tang.

These notes have not been thoroughly reviewed. Any errors below are my own responsibility.

Sources:

- The original paper by James Cook and Ian Mertz
  - https://dl.acm.org/doi/10.1145/3618260.3649664
- A version by Oded Goldreich I found easier to understand
  - https://www.wisdom.weizmann.ac.il/~oded/VO/tree-eval.pdf
- A stunning result by Ryan Williams that uses space-efficient tree evaluation
  - https://arxiv.org/pdf/2502.17779

Problem

You have a complete binary tree of height h. Each leaf carries a b-bit string. Each internal node represents a gate, which is an aribtrary function $\{0,1\}^b \times \{0,1\}^b \to \{0,1\}^b$.

To give things names, the internal nodes are indexed by $U = \bigcup_{i=0}^{h-1} \{0,1\}^i$, and for $u \in U$ the children of u are $u0$ and $u1$. Each leaf $u \in \{0,1\}^h$ is assigned a b-bit value v_u.

The desired output is v_λ such that for ever $u \in U$ we have $v_u = f_u(v_{u0}, v_{u1})$. We want an algorithm that produces this output using a minimal amount of space.

Note: everything here works just as wb for complete m-ary trees of height h, for constants $m \geq 2$.

History

Note that the size of an instance is $n = \Theta\big(b \cdot 2^{h+2b}\big)$.

The solution by the natural "DFS"-like algorithm uses $\Theta(hb) = O(\log^2 n)$ space, and was long believed to be optimal. To the surprise of many complexity theorists, Cook and Mertz in 2024 reduced this to

$$\Theta((h+b)\log b) = O(\log n \log\log n).$$

Solution

Framework

Consider a global memory that stores 3 values, or registers, $(x, y, z) \in \{0,1\}^b \times \{0,1\}^b \times \{0,1\}^b$.

Goal: Design a (recursive) procedure that, when invoked with argument u, modifies global memory according to $(x, y, z) \to (x, y, z \oplus v_u)$.

Clearly, if we have such a procedure, we initialize the global memory to zeros and invoke the procedure with argument $u = \lambda$ to get the correct output in the third register.

The base case is when u is a leaf, and you simply XOR the desired register with v_u, which is specified in the instance.

The general plan for the recursive procedure (when u is not a leaf) is to update the memory in 5 steps as follows:

$$\begin{aligned}
(x, y, z) &\to (x \oplus v_{u0}, y, z) \\
&\to (x \oplus v_{u0}, y \oplus v_{u1}, z) \\
&\to (x \oplus v_{u0}, y \oplus v_{u1}, z \oplus v_u) \\
&\to (x \oplus v_{u0}, y, z \oplus v_u) \\
&\to (x, y, z \oplus v_u).
\end{aligned}$$

Steps 1, 2, 4, and 5 are recursive calls to the procedure, except modifying the first and second registers instead of the third (which we can do by symmetry).

Step 3 is the miraculous step, and we show how to do it below.

To summarize the plan so far: we make recursive calls to XOR x and y with the values of the children, (somehow) XOR z with the value of the parent while keeping the other registers intact, then repeat the initial calls to restore x and y.

Miraculous step

Let $\mathcal{K}$ be a prime field and let $f_{u,i}(x, y)$ denote the i-th bit of $f_u(x, y)$. Then the multilinear extension of $f_{u,i}$ is $\widehat{f_{u,i}} : \mathcal{K}^b \times \mathcal{K}^b \to \mathcal{K}^b$ given by

$$\widehat{f_{u,i}}(p, q) = \sum_{x,y \in \{0,1\}^b} \prod_{i=1}^{2n} \begin{cases} (p\|q)_i & \text{if } (x\|y)_i = 1 \\ 1 - (p\|q)_i & \text{if } (x\|y)_i = 0\text{'} \end{cases}$$

where $\|$ denotes concatenation and i indexes the bits of $(x\|y)$ or the digits of $(p\|q)$.

In other words, $\widehat{f_{u,i}}$ is the unique extension of $f_{u,i}$ to $\mathcal{K}^b \times \mathcal{K}^b$ which agrees with $f_{u,i}$ on $\{0, 1\}^i$ and is multilinear (i.e. linear in each of the $2b$ input "digits").

Choose $\mathcal{K}$ such that m-th roots of unity $1, \omega, ..., \omega^{m-1}$ exist, for some $m > 2b$ and $|\mathcal{K}| = O(b)$.

Due to linearity and roots-of-unity filter, we have the crucial identity:

$$\sum_{j=0}^{m-1} L(\omega^j s + v) = m \cdot L(v)$$

for all multilinear functions L over $\mathcal{K}^{2b}$, where $s, v \in \mathcal{K}^{2b}$ and scalar multiplication and addition of arguments are done componentwise.

Remark. Using $1, \omega, ..., \omega^{m-1}$ instead of arbitrary distinct elements of K is a choice made for convenience. The equation above is a form of polynomial interpolation, and choosing roots of unity happens to make the coefficients of the interpolation all equal 1.

Putting it together

Modify the earlier framework to work over $\mathcal{K}$ instead of $\mathbb{F}_2$. Our global memory will have 3 registers, each storing a value in $\mathcal{K}^b$. When we call our procedure with arguments u and $\delta \in \{\pm 1\}$, we want to transform the registers according to $(p, q, r) \to (p, q, r + \delta v_u)$.

- There's nothing deep going on with the additional parameter δ, it's just that we're working over $\mathcal{K}$ instead of $\mathbb{F}_2$, so instead of XORing, we have to know whether we're adding or subtracing.

We iterate over $j = 0, ..., m - 1$. At each iteration, our goal is to add

$$\delta \widehat{f_{u,i}}(\omega^j p + v_{u0}, \omega^j q + v_{u1})/m$$

to the i-th digit of r, while keeping the other two registers intact. By the identity above, the net result is adding

$$\delta \sum_{j=0}^{m-1} \delta \widehat{f_{u,i}}(\omega^j p + v_{u0}, \omega^j q + v_{u1})/m = \delta \widehat{f_{u,i}}(v_{u0}, v_{u1}) = \delta v_{u,i}.$$

Iteration j of the procedure now looks like

$$
\begin{aligned}
(p, q, r) &\to (\omega^j p, \omega^j q, r) \\
&\to (\omega^j p + \delta v_{u0}, \omega^j q, r) \\
&\to (\omega^j p + \delta v_{u0}, \omega^j q + \delta v_{u1}, r) \\
&\to (\omega^j p + \delta v_{u0}, \omega^j q + \delta v_{u1}, r + \delta R) \\
&\to (\omega^j p + \delta v_{u0}, \omega^j q, r + \delta R) \\
&\to (\omega^j p, \omega^j q, r + \delta R) \\
&\to (p, q, r + \delta R),
\end{aligned}
$$

where R is a shorthand for the concatenation of the field elements

$$\widehat{f_{u,i}}(\omega^j p + v_{u0}, \omega^j q + v_{u1})/m,$$

recalling that this was our target expression to add to the third register in iteration j.

Now, we see how the miraculous step in the middle can actually be performed, because R is a function of the values stored in the first two registers.

This completes the definition of the key recursive procedure.

Finally, we can initialize the three registers to zeros and call the recursive procedure with arguments $u = \lambda, \sigma = 1$ to get v_λ in the third register, as desired.

Space analysis

The details of the space analysis are not as nice as the algorithm itself, so they are omitted from these notes. It turns out that the algorithm uses $O((h + b) \log b) = O(\log n \log \log n)$ memory.

An Implication

Ryan Williams showed, roughly speaking, that any program that runs in $t(n)$ time can be simulated by another program that uses $O\left(\sqrt{t(n) \log t(n)}\right)$ space, by showing a connection between the tree evaluation problem and the problem of general space-efficient computation. This result can be abbreviated as

$$\mathsf{TIME}(t(n)) \subseteq \mathsf{SPACE}\left(\sqrt{t(n) \log t(n)}\right)$$

This result was surprising to many complexity theorists including Williams himself. Before Williams's result, it was commonly believed that for all $\varepsilon > 0$,

$$\mathsf{TIME}(t(n)) \not\subset \mathsf{SPACE}(t(n)^{1-\varepsilon}).$$

The recent results in space-efficient tree evaluation and space-efficient simulation of time-bound programs are huge steps forward in complexity theory.

References

[1] Cook, J. & Mertz, I. "Tree evaluation is in space $O(\log n \cdot \log \log n)$". *ACM Symposium on Theory of Computing* 56 (2024), pp. 1268–1278. DOI: https://doi.org/10.1145/3618260.3649664

[2] Goldreich, O. "On the Cook-Mertz Tree Evaluation Procedure". Department of Computer Science, Weizmann Institute of Science (2024). URL: https://www.wisdom.weizmann.ac.il/~oded/VO/tree-eval.pdf

[3] Williams, R. "Simulating Time With Square-Root Space". arXiv (2025). DOI: https://doi.org/10.48550/arXiv.2502.17779