Borrowing memory that's being used: catalytic approaches to the Tree Evaluation Problem

James Cook, Ian Mertz

April 6, 2020

Section 1

The Tree Evaluation Problem

Pebbles and Branching Programs for Tree Evaluation [S. Cook, P. McKenzie, D. Wehr, M. Braverman, R. Santhanam 2010]
New Results for Tree Evaluation [S. Chan, J. Cook, S. Cook, P. Nguyen, D. Wehr 2010]

$$\mathrm{AC}^0(6) \subseteq \mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PH}$$

$$\mathsf{AC}^0(6) \subseteq \mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH}$$

We don't know whether $\mathsf{AC}^0(6) = \mathsf{PH}$.

$$\mathsf{AC}^0(6) \subseteq \mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH}$$

We don't know whether $\mathsf{AC}^0(6) = \mathsf{PH}$.

P $=$ "polynomial time": $O(n^{O(1)})$ time.

P = "polynomial time": $O(n^{O(1)})$ time.

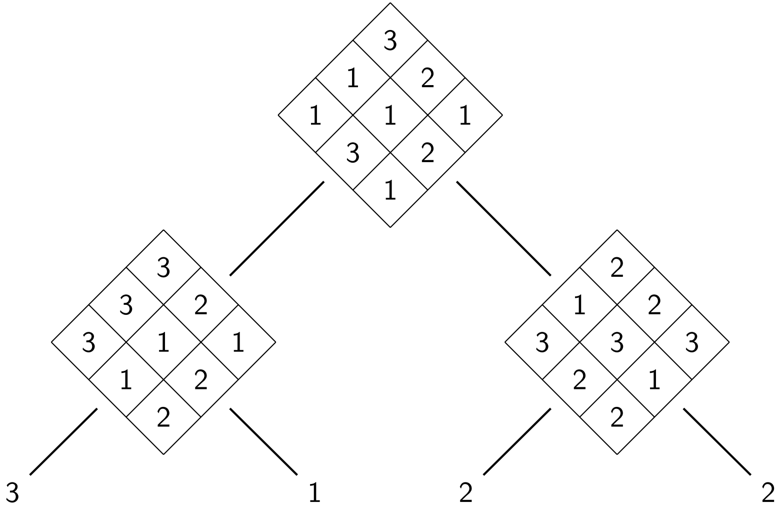
L = "logarithmic space": $O(\log n)$ memory.
$2^{O(\log n)} = n^{O(1)}$ configurations, so L $\subseteq$ P.

P = "polynomial time": $O(n^{O(1)})$ time.

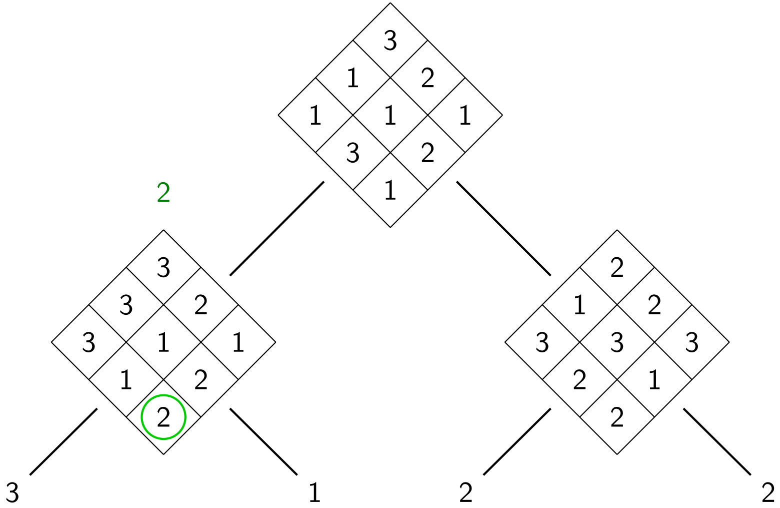
L = "logarithmic space": $O(\log n)$ memory.
$2^{O(\log n)} = n^{O(1)}$ configurations, so L $\subseteq$ P.

TEP $\in$ P.
Goal: prove TEP $\notin$ L, so L $\neq$ P.

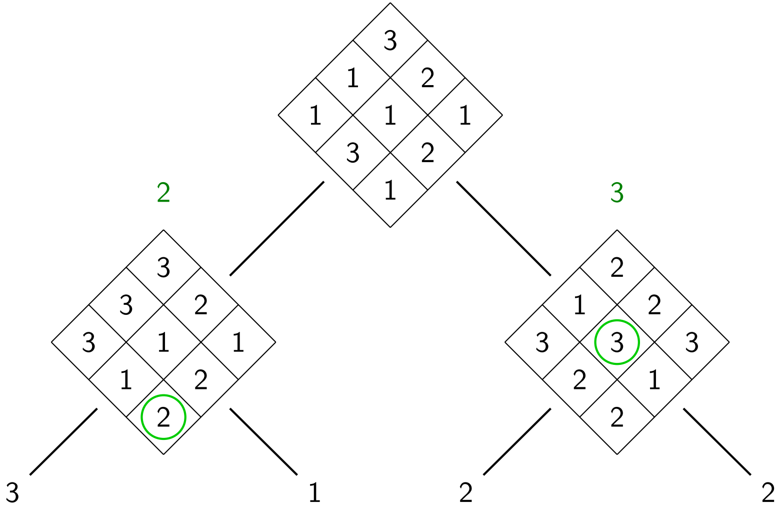
The Tree Evaluation Problem



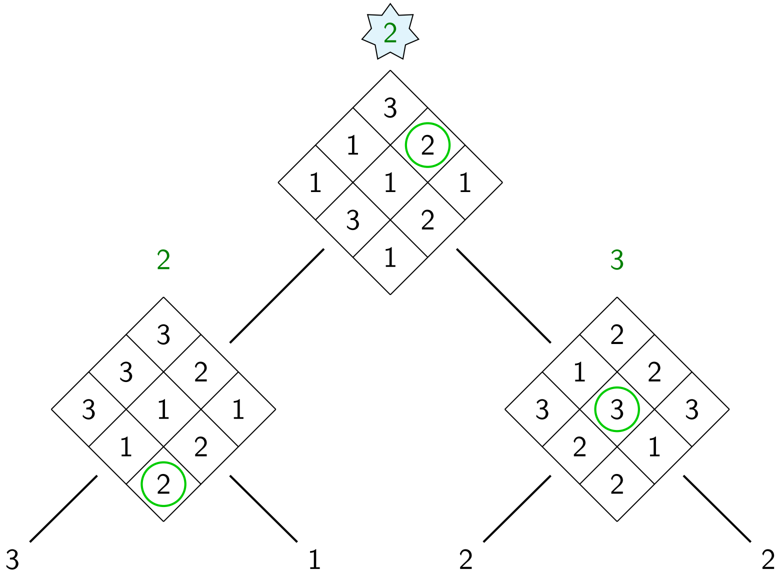
The Tree Evaluation Problem



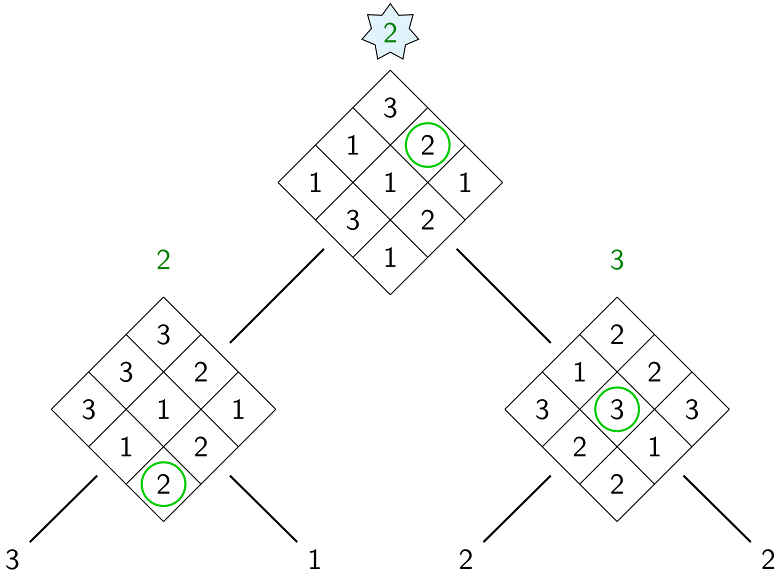
The Tree Evaluation Problem



The Tree Evaluation Problem



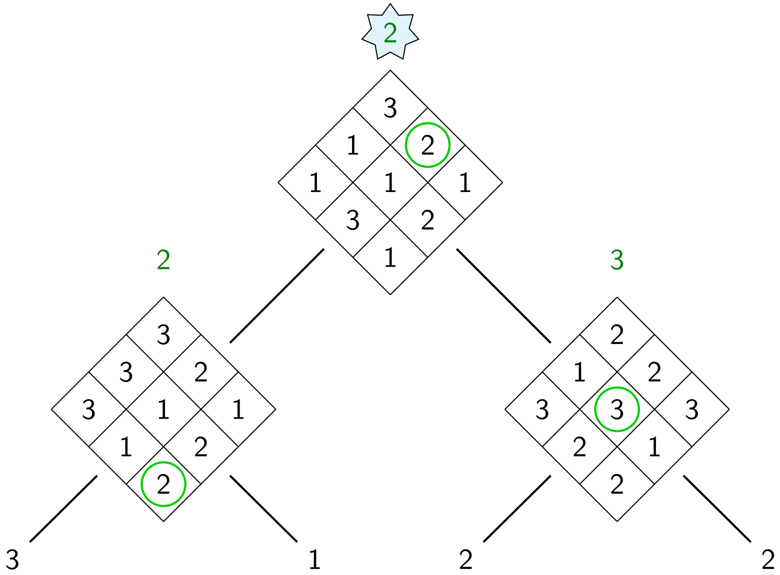
The Tree Evaluation Problem



Parameters:

- height $= 3$
- k $= 3$

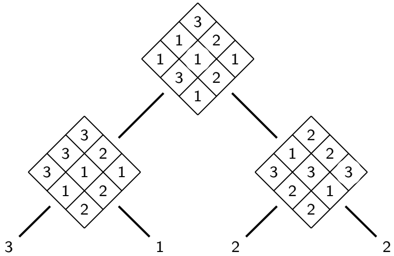
The Tree Evaluation Problem

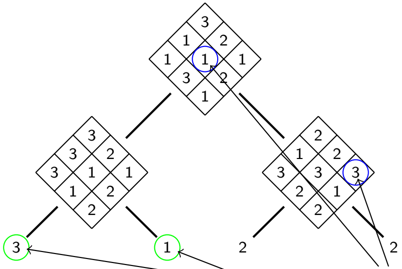


Parameters:

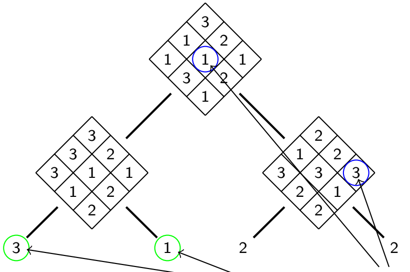
- height = 3
- k = 3

Input size:
$n = \Theta(2^h k^2 \log k)$.


3
1 2
1 1 1
3 2
1
3
3 2
3 1 1
1 2
2
2
1 2
3 3 3
2 1
2
3 1 2 2



A *query* is either a leaf or a cell in a table of an internal node.

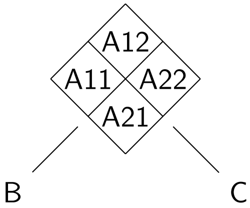


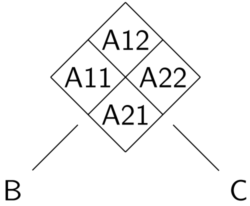
A *query* is either a leaf or a cell in a table of an internal node.

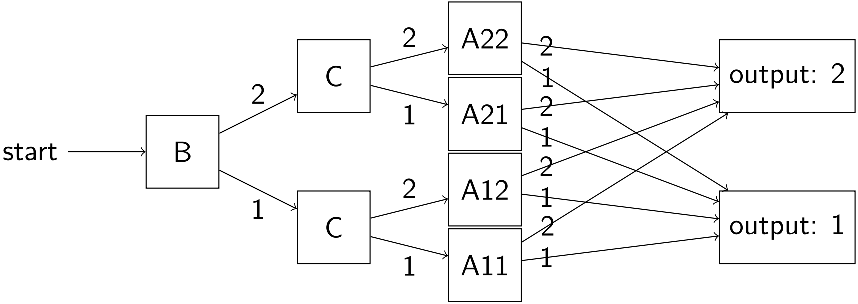
A *branching program* is a directed graph of *states*. There are two kinds of state:

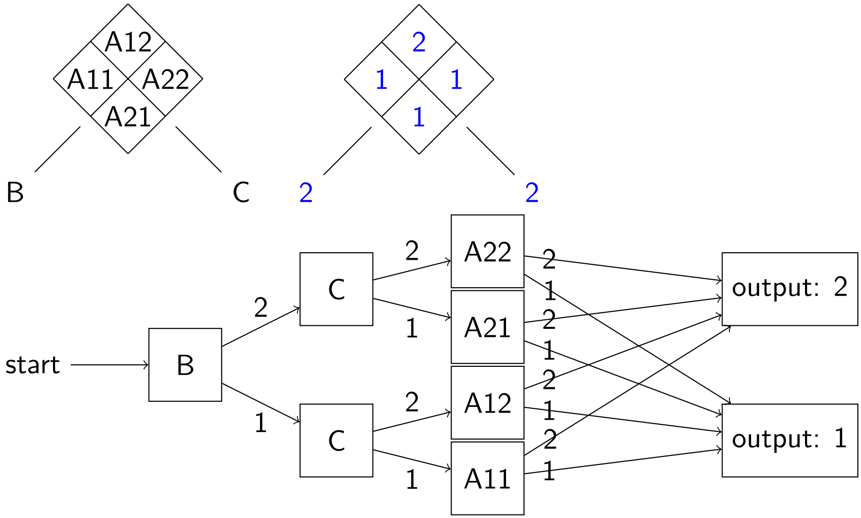
- *query state*: labelled with a query and has k outgoing edges labelled with the possible answers.
- *final state*: labelled with a number $1..k$.

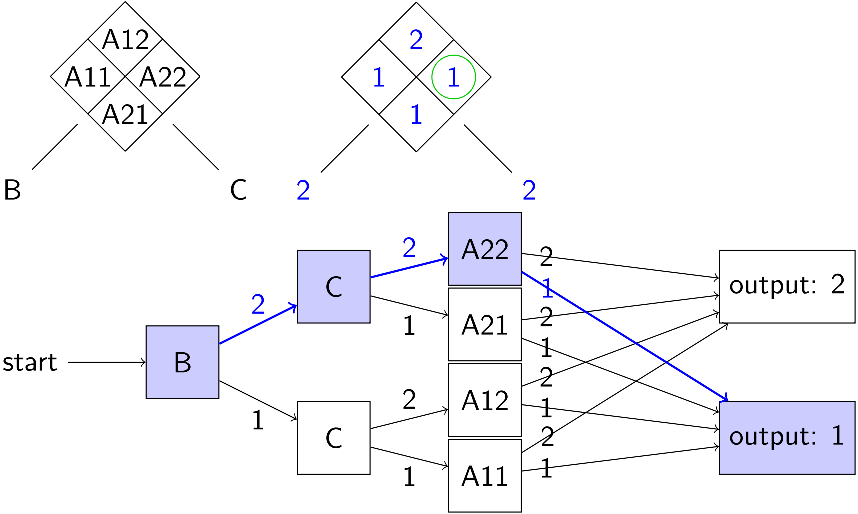
One state is the starting state.


A12
A11
A22
A21
B
C

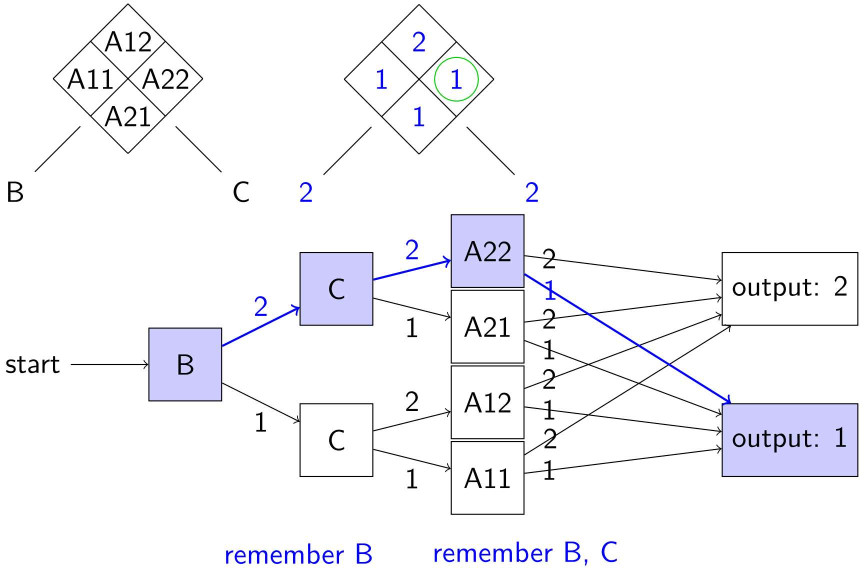

A12
A11
A22
A21
B
C


start
B
2
1
C
C
2
1
2
1
A22
A21
A12
A11
2
1
2
1
2
1
2
1
output: 2
output: 1


A12
A11
A22
A21
B
C
2
1
1
1
2
2
start
B
C
C
A22
A21
A12
A11
output: 2
output: 1
2
1
2
1
2
1
2
1
2
1
2
1
2
1



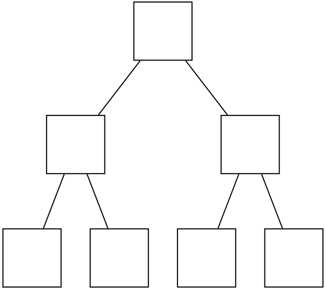
A12
A11
A22
A21
B
C
2
1
1
1
2
2
start
B
C
C
A22
A21
A12
A11
output: 2
output: 1
2
1
2
1
2
1
2
1
2
1
2
1
2
1



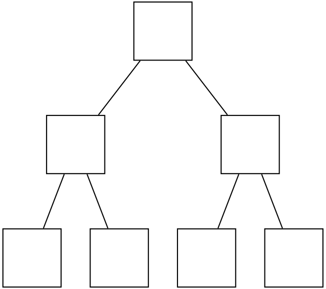
A12
A11
A22
A21
B
C
2
1
1
1
2
2
start
B
C
C
A22
A21
A12
A11
output: 2
output: 1
remember B
remember B, C

Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]

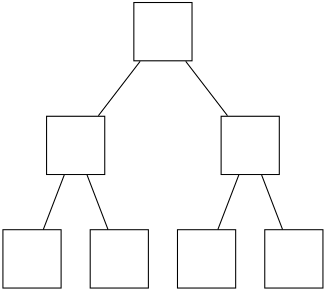


Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

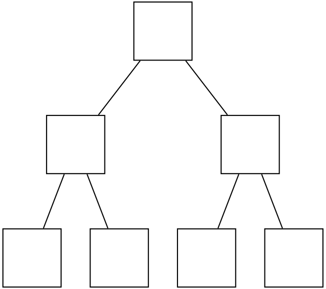
Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Pebbling game [Paterson Hewitt 1970]

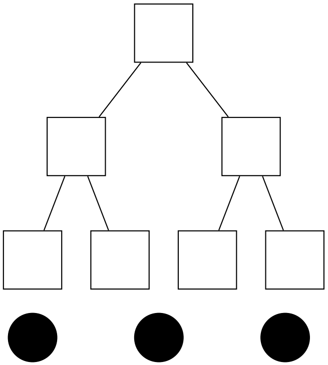


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

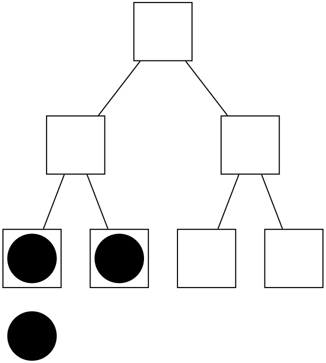


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

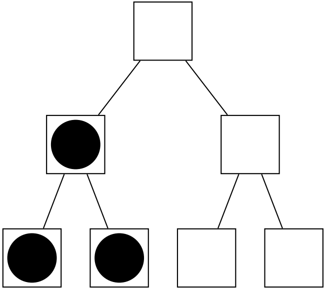


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

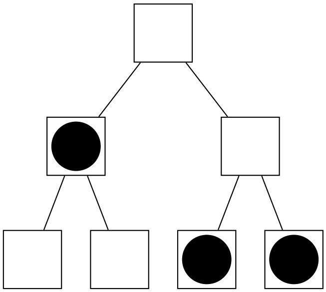


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

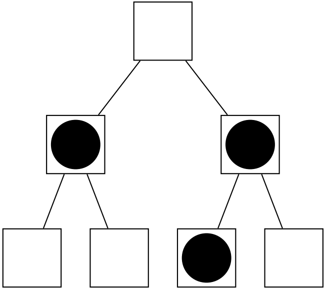


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

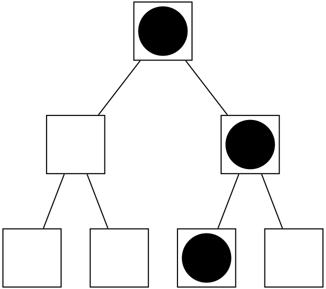


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]

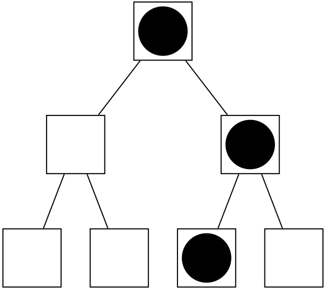


Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Pebbling game [Paterson Hewitt 1970]



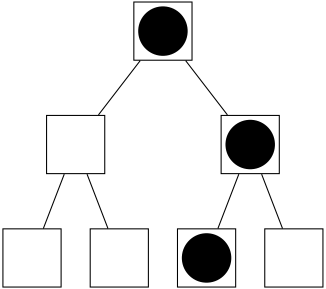
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.

Pebbling game [Paterson Hewitt 1970]



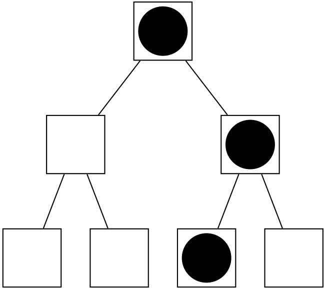
Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).
Two kinds of move:

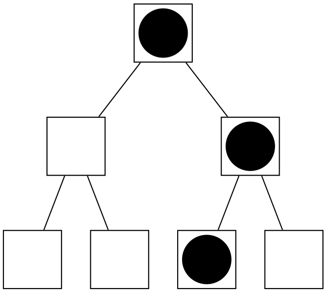
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Theorem: h pebbles are needed.

Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).
Two kinds of move:

- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

Theorem: h pebbles and $2^h - 1$ steps are enough.
Corollary: A branching program with $2^h k^h$ states can solve TEP.

Theorem: h pebbles are needed.
Conjecture (false): To solve TEP, a branching program needs $\Omega(k^h)$ states.

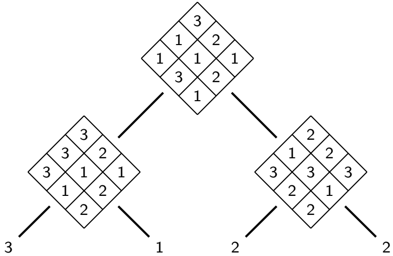
The Tree Evaluation Problem

- Motivation and definition
- Branching programs and pebbling games
- Lower bounds

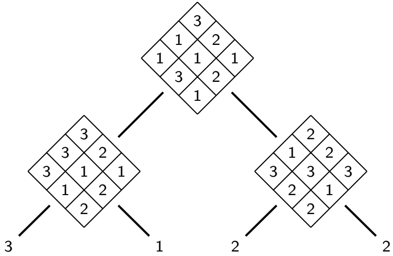
New algorithm

- Reversible computation
- Solving TEP

Input size: $\Theta(2^h k^2 \log k)$.



Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.



Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

New algorithm: $(\frac{k}{h} + 1)^{\Theta(h)} k^{\Theta(1)}$ states. (Beats pebbling when $h \geq k^{4/5+\epsilon}$.)

Input size: $\Theta(2^h k^2 \log k)$. So, log space $= O(h + \log k)$ memory.

If TEP $\in$ L, then it can be solved by a family of branching programs with $2^{O(h+\log k)} = 2^{O(h)} k^{O(1)}$ states.

Pebbling algorithm (previous best):

- 2^h layers.
- Up to k^h states per layer.
- Total $\Theta((k+1)^h)$ states.

New algorithm: $(\frac{k}{h} + 1)^{\Theta(h)} k^{\Theta(1)}$ states. (Beats pebbling when $h \geq k^{4/5+\epsilon}$.)

Neither algorithm fits in $2^{O(h)} k^{O(1)}$ states, so TEP $\notin$ L is still possible.

Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume. . .

Lower bounds

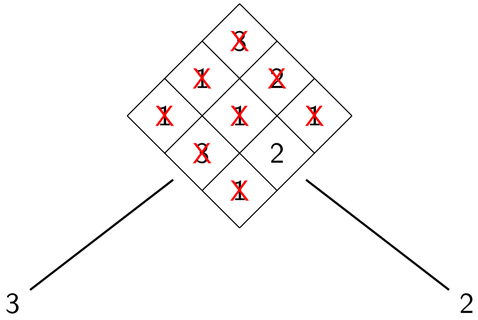
Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

- the algorithm is *read-once*

Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

- the algorithm is *read-once*
- or the algorithm is *thrifty*: never reads an irrelevent piece of the input.



Lower bounds

Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume...

- the algorithm is *read-once*
- or the algorithm is *thrifty*: never reads an irrelevent piece of the input.

New algorithm: $(\frac{k}{h}+1)^{\Theta(h)}k^{\Theta(1)} \notin \Omega(k^h)$.

Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state

Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state

Result: with $O(\log n)$ ordinary memory and $n^{O(1)}$ extra memory, can compute things not known to be in L, e.g. matrix determinant, NL, ...

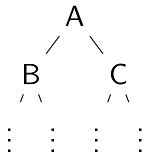
Catalytic space

Computing with a full memory: catalytic space [BCKLS 2014].

Given:

- Small ordinary memory
- Large memory that must be returned to its original state

Result: with $O(\log n)$ ordinary memory and $n^{O(1)}$ extra memory, can compute things not known to be in L, e.g. matrix determinant, NL, ...



This rules out the following lower bound argument:

- At some point, you need to compute B.
- You need to remember B ($\log k$ bits) while computing C.
- So, every level of the tree adds $\log k$ bits you need to remember.

Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1*.* [D. Barrington 1989]

Computing algebraic formulas using a constant number of registers. [M. Ben-Or, R. Cleve 1992]

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Ring R
Inputs $x_1, \ldots, x_n \in R$
Work registers $r_1, \ldots, r_m \in R$

Reversible instructions:

- Example: $r_5 \leftarrow r_5 + r_4 \times x_1$.
- Inverse is $r_5 \leftarrow r_5 - r_4 \times x_1$.

Notation: τ_j denotes the starting value of register r_j.

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Invert the whole sequence by running the inverse of each instruction in reverse order. (Computes $-f$.)

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Lemma

Suppose there is a sequence of ℓ *instructions that cleanly computes* f*, and each instruction has the form:*

$$(r_1, \ldots, r_m) \leftarrow g(x_j, r_1, \ldots, r_m)$$

Then there is a branching program that computes f *with* $\ell |R|^m$ *states.*

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Example

Cleanly compute $x_1 + x_2$ into r_1:

- $r_1 \leftarrow r_1 + x_1$
- $r_1 \leftarrow r_1 + x_2$

Definition

A sequence of reversible instructions *cleanly computes* f into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Example

Cleanly compute $x_1 + x_2$ into r_1:

- $r_1 \leftarrow r_1 + x_1$ $\quad [r_1 = \tau_1 + x_1]$
- $r_1 \leftarrow r_1 + x_2$

Definition

A sequence of reversible instructions *cleanly computes f* into r_i if, once it finishes:

- $r_i = \tau_i + f(x_1, \ldots, x_n)$
- all other registers are unchanged ($r_j = \tau_j$ for $j \neq i$)

Example

Cleanly compute $x_1 + x_2$ into r_1:

- $r_1 \leftarrow r_1 + x_1$ $\quad [r_1 = \tau_1 + x_1]$
- $r_1 \leftarrow r_1 + x_2$ $\quad [r_1 = \tau_1 + x_1 + x_2]$

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\quad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- $P_1 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\qquad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $\qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $\qquad [r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $\qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $\qquad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- $P_1 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 - f_1 \times \tau_2]$
- $P_2 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- $P_1^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 + f_1 \times f_2]$
- P_2^{-1}

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2]$
- P_1 $[r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 - f_1 \times \tau_2]$
- P_2 $[r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$ $[r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- P_1^{-1} $[r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$ $[r_3 = \tau_3 + f_1 \times f_2]$
- P_2^{-1} $[r_1 = \tau_1, r_2 = \tau_2]$

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2]$
- $P_1 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 - f_1 \times \tau_2]$
- $P_2 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2 \qquad [r_3 = \tau_3 + \tau_1 \times \tau_2 + \tau_1 \times f_2 + f_1 \times f_2]$
- $P_1^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2 \qquad [r_3 = \tau_3 + f_1 \times f_2]$
- $P_2^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2]$

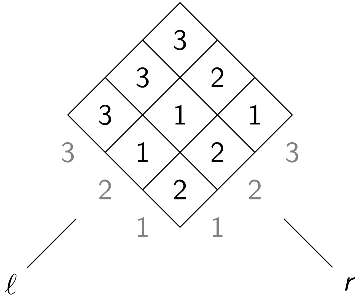
Cost: need to run P_1 and P_2 twice each. But: no memory needs to be reserved.

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.

A formula for TEP

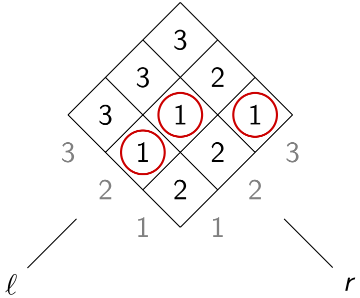
Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.
Suppose node v has children ℓ and r:



$$[v = 1] =$$

A formula for TEP

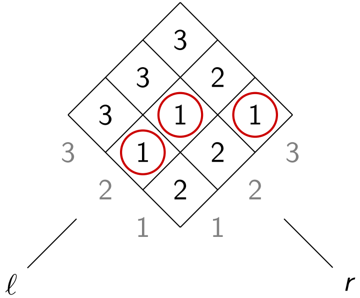
Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.
Suppose node v has children ℓ and r:



$$[v = 1] =$$

A formula for TEP

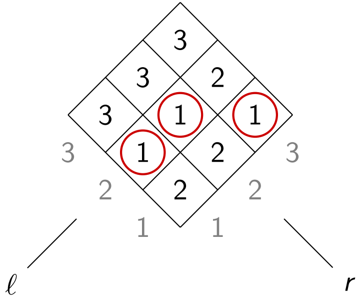
Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.
Suppose node v has children ℓ and r:



$[v = 1] =$
$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$

A formula for TEP

Let $R = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. Define $[x = y] = 1$ if $x = y$, 0 otherwise.
Suppose node v has children ℓ and r:



$$[v = 1] =$$
$$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$$

Let f_v denote v's table. In general,

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y, z) = x] \times [\ell = y] \times [r = z]$$

First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm

CheckNode(v, x, i)
Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm

CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i

Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.

First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm

CheckNode(v, x, i)
Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
    using multiplication lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm

CheckNode(v, x, i)

Parameters: node v, value $x \in [k]$, target register i

Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$
    using multiplication lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

Needs three registers.

First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm

CheckNode(v, x, i)
Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$ using multiplication lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

Needs three registers. Gives branching program with width 8 and length $(k^2)^{h-1}$.

First attempt

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm

CheckNode(v, x, i)
Parameters: node v, value $x \in [k]$, target register i
Computes $r_i \leftarrow r_i + [v = x]$

- If v is a leaf:
  - $r_i \leftarrow r_i + [v = x]$ is one instruction.
- else: for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$ using multiplication lemma: 2 calls each to CheckNode(ℓ, y, j) and CheckNode(r, z, j'), where j and j' are two registers other than i.

Needs three registers. Gives branching program with width 8 and length $(k^2)^{h-1}$.
Worse than pebbling, which uses $\Theta((k+1)^h)$ states.

- for $(y, z) \in [k]^2$:
    - $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$

- for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$

$r_j \leftarrow r_j + [\ell = 1]$

$r_i \leftarrow r_i - r_j \times r_{j'}$

$r_{j'} \leftarrow r_{j'} + [r = 1]$

$r_i \leftarrow r_i + r_j \times r_{j'}$

$r_j \leftarrow r_j - [\ell = 1]$

$r_i \leftarrow r_i - r_j \times r_{j'}$

$r_{j'} \leftarrow r_{j'} - [r = 1]$

$r_i \leftarrow r_i + r_j \times r_{j'}$

- for $(y, z) \in [k]^2$:
  - $r_i \leftarrow r_i + [f_v(y,z) = x] \times [\ell = y] \times [r = z]$

$$r_j \leftarrow r_j + [\ell = 1]$$
$$r_i \leftarrow r_i - r_j \times r_{j'}$$
$$r_{j'} \leftarrow r_{j'} + [r = 1]$$
$$r_i \leftarrow r_i + r_j \times r_{j'}$$
$$r_j \leftarrow r_j - [\ell = 1]$$
$$r_i \leftarrow r_i - r_j \times r_{j'}$$
$$r_{j'} \leftarrow r_{j'} - [r = 1]$$
$$r_i \leftarrow r_i + r_j \times r_{j'}$$

$$r_j \leftarrow r_j + [\ell = 1]$$
$$r_i \leftarrow r_i - r_j \times r_{j'}$$
$$r_{j'} \leftarrow r_{j'} + [r = 2]$$
$$r_i \leftarrow r_i + r_j \times r_{j'}$$
$$r_j \leftarrow r_j - [\ell = 1]$$
$$r_i \leftarrow r_i - r_j \times r_{j'}$$
$$r_{j'} \leftarrow r_{j'} - [r = 2]$$
$$r_i \leftarrow r_i + r_j \times r_{j'}$$

$$r_j \leftarrow r_j + [\ell = 1]$$
$$r_i \leftarrow r_i - r_j \times r_{j'}$$
$$r_{j'} \leftarrow r_{j'} + [r = 3]$$
$$r_i \leftarrow r_i + r_j \times r_{j'}$$
$$r_j \leftarrow r_j - [\ell = 1]$$
$$r_i \leftarrow r_i - r_j \times r_{j'}$$
$$r_{j'} \leftarrow r_{j'} - [r = 3]$$
$$r_i \leftarrow r_i + r_j \times r_{j'}$$

$\cdots$

$\cdots$

$\cdots$

One-hot encoding

Given a value $x \in [k]$, define $\mathrm{OneHot}(x) = ([x = 1], [x = 2], \ldots, [x = k]) \in \{0, 1\}^k$.

E.g. for $k = 3$, $\mathrm{OneHot}(2) = (0, 1, 0)$.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

Algorithm

ComputeOneHot(v, i)   Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \text{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \text{OneHot}(r)$

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \text{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \text{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \text{OneHot}(r)$

$$F(\vec{r_j}, \vec{r_{j'}})_x = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times (\vec{r_j})_y \times (\vec{r_{j'}})_z$$

Note:

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Algorithm

ComputeOneHot(v, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^k$.

Parameters: node v, target register i

Computes $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{OneHot}(v)$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{OneHot}(v)$ is one instruction.
- else:
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} + \mathrm{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} + \mathrm{OneHot}(r)$
  - $\vec{r_i} \leftarrow \vec{r_i} + F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_j} \leftarrow \vec{r_j} - \mathrm{OneHot}(\ell)$
  - $\vec{r_i} \leftarrow \vec{r_i} - F(\vec{r_j}, \vec{r_{j'}})$
  - $\vec{r_{j'}} \leftarrow \vec{r_{j'}} - \mathrm{OneHot}(r)$

$$F(\vec{r_j}, \vec{r_{j'}})_x = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times (\vec{r_j})_y \times (\vec{r_{j'}})_z$$

Note:

$$[v = x] = \sum_{(y,z)\in[k]^2} [f_v(y,z) = x] \times [\ell = y] \times [r = z]$$

Gives branching program with width 2^{3k}, length $\Theta(k^2 4^h)$. Total $2^{\Theta(k+h)}$ states.

Pebbling algorithm: $\Theta((k+1)^h)$
ComputeOneHot: $2^{\Theta(k+h)}$ states.

Pebbling algorithm: $\Theta((k+1)^h) = \Theta(2^{h\log_2(k+1)})$
ComputeOneHot: $2^{\Theta(k+h)}$ states. Better when $h\log(k+1) >> k+h$, i.e. when $h >> \dfrac{k}{\log k}$.

Pebbling algorithm: $\Theta((k+1)^h) = \Theta(2^{h\log_2(k+1)})$
ComputeOneHot: $2^{\Theta(k+h)}$ states. Better when $h\log(k+1) >> k+h$, i.e. when $h >> \dfrac{k}{\log k}$.

Can we do better?

Binary encoding

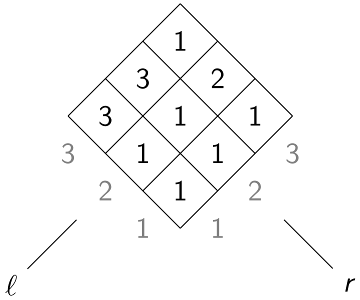
Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.

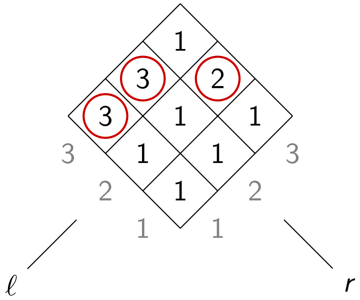


$$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.

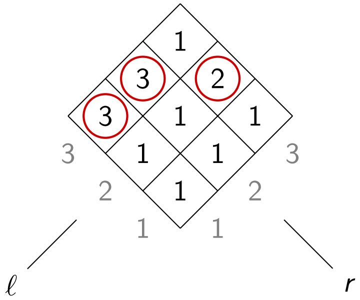


$$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.

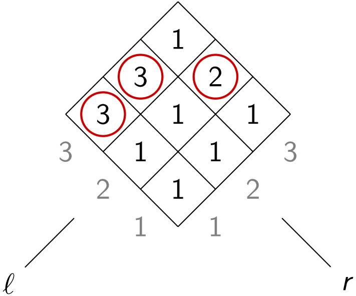


$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$
$[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3]$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.



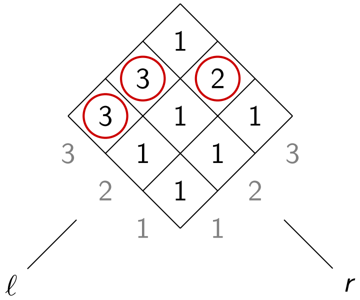
$\mathrm{Bin}(v)_1 = [v = 2] + [v = 3] =$
$[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3]$

$[\ell = 1] = (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.



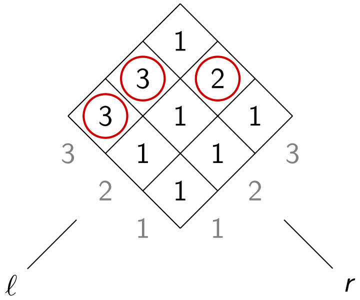
$$\mathrm{Bin}(v)_1 = [v=2] + [v=3] =$$
$$[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3]$$
$$= (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2 \times (1 + \mathrm{Bin}(r)_1) \times \mathrm{Bin}(r)_2$$
$$+ (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2 \times \mathrm{Bin}(r)_1 \times (1 + \mathrm{Bin}(r)_2)$$
$$+ \mathrm{Bin}(\ell)_1 \times (1 + \mathrm{Bin}(\ell)_2) \times \mathrm{Bin}(r)_1 \times \mathrm{Bin}(r)_2$$

$$[\ell = 1] = (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2$$

Binary encoding

Given a value $x \in [k]$, let $\mathrm{Bin}(x) \in \{0,1\}^{\lceil \log k \rceil}$ be its binary encoding.

E.g. for $k = 3$, $\mathrm{Bin}(1) = (0,1)$.



$$\begin{aligned}
&\mathrm{Bin}(v)_1 = [v=2] + [v=3] = \\
&[\ell = 1] \times [r = 1] + [\ell = 1] \times [r = 2] + [\ell = 2] \times [r = 3] \\
&= (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2 \times (1 + \mathrm{Bin}(r)_1) \times \mathrm{Bin}(r)_2 \\
&+ (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2 \times \mathrm{Bin}(r)_1 \times (1 + \mathrm{Bin}(r)_2) \\
&+ \mathrm{Bin}(\ell)_1 \times (1 + \mathrm{Bin}(\ell)_2) \times \mathrm{Bin}(r)_1 \times \mathrm{Bin}(r)_2
\end{aligned}$$

$$[\ell = 1] = (1 + \mathrm{Bin}(\ell)_1) \times \mathrm{Bin}(\ell)_2$$

In general, $\mathrm{Bin}(v)_x$ can be written as a degree-$2\lceil \log k \rceil$ polynomial involving $\mathrm{Bin}(\ell)$ and $\mathrm{Bin}(r)$.

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- P_1
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- P_1^{-1}
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- P_2^{-1}
- $r_3 \leftarrow r_3 + r_1 \times r_2$

Lemma: Multiplication

Suppose P_1 cleanly computes f_1 into r_1 and P_2 cleanly computes f_2 into r_2. Then we can cleanly compute $f_1 \times f_2$ into r_3 as follows:

- $P_1 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- $P_2 \qquad [r_1 = \tau_1 + f_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$
- $P_1^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2 + f_2]$
- $r_3 \leftarrow r_3 - r_1 \times r_2$
- $P_2^{-1} \qquad [r_1 = \tau_1, r_2 = \tau_2]$
- $r_3 \leftarrow r_3 + r_1 \times r_2$

Lemma: d-ary multiplication

Suppose we have d values $f_1, \ldots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

Lemma: d-ary multiplication

Suppose we have d values $f_1, \ldots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

- For every subset $S \subseteq [d]$:
  - Call $P(S')$, choosing S' so that $r_i = \tau_i$ for $i \notin S$, and $r_i = \tau_i + f_i$ for $i \in S$.
  - $r_{d+1} \leftarrow r_{d+1} + c_S \times \prod_{i=1}^{d} r_i$
- Call P once more to ensure $r_i = \tau_i$ for $i = 1, \ldots, d$.

Lemma: d-ary multiplication

Suppose we have d values $f_1, \ldots, f_d$, and a general subroutine P. For any $S \subseteq [d]$, $P(S)$ cleanly computes $r_i \leftarrow r_i + f_i$ for every $i \in S$, and leaves r_j alone for $j \notin S$. Then we can cleanly compute $f_1 \times \cdots \times f_d$ into r_{d+1} as follows:

- For every subset $S \subseteq [d]$:
  - Call $P(S')$, choosing S' so that $r_i = \tau_i$ for $i \notin S$, and $r_i = \tau_i + f_i$ for $i \in S$.
  - $r_{d+1} \leftarrow r_{d+1} + c_S \times \prod_{i=1}^{d} r_i$
- Call P once more to ensure $r_i = \tau_i$ for $i = 1, \ldots, d$.

Uses $d + 1$ registers and 2^d recursive calls.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.
Parameters: node v, set $S \subseteq [\log k]$, target register i
Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \text{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \text{ComputeBin}(v)$ is one instruction.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls.
It gives branching program with width $2^{3 \log k} = k^3$ and length $\Theta((2k)^{2h} k^{O(1)})$. Total $\Theta(k^{2h+O(1)})$ states.

Algorithm

ComputeBin(v, S, i) Uses *vector registers* $\vec{r_i} \in \{0,1\}^{\lceil \log k \rceil}$.

Parameters: node v, set $S \subseteq [\log k]$, target register i

Computes: $\vec{r_i}_b \leftarrow \vec{r_i}_b + \mathrm{Bin}(v)_b$ for all $b \in S$

- If v is a leaf:
  - $\vec{r_i} \leftarrow \vec{r_i} + \mathrm{ComputeBin}(v)$ is one instruction.
- else:
  - for all subsets $T_1, T_2 \subseteq [\log k]$:
    - Call ComputeBin(ℓ, T_1', j) and ComputeBin(r, T_2', j').
    - for all $b \in S$, $(\vec{r_i})_b \leftarrow (\vec{r_i})_b + F(\vec{r_j}, \vec{r_{j'}})$

ComputeBin uses $3 \log k$ bits of memory and makes $2 \times 2^{2 \log k} = 2k^2$ recursive calls. It gives branching program with width $2^{3 \log k} = k^3$ and length $\Theta((2k)^{2h} k^{O(1)})$. Total $\Theta(k^{2h+O(1)})$ states.

Worse than pebbling, which uses $\Theta((k+1)^h)$ states.

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a=1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a = 1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a = \log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)} k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a = 1$	$2^{\Theta(k)}$	$2^{\Theta(h)} k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a = \log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid, $a = \log(\frac{k}{h}+1)$	$\Theta((\frac{k}{h}+1)^{\Theta(h)}$	$\Theta((\frac{k}{h}+1)^{\Theta(h)})$	$(\frac{k}{h}+1)^{5h} k^{\Theta(1)}$

algorithm	width	length	total states
One-hot	$2^{\Theta(k)}$	$\Theta(k^2 4^h)$	$2^{\Theta(k+h)}$
Binary	k^3	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid	$2^{\Theta(\frac{a}{2^a-1}k)}$	$2^{\Theta(ah)}k^{\Theta(1)}$	$2^{\Theta(ah+\frac{a}{2^a-1}k))}$
Hybrid, $a=1$	$2^{\Theta(k)}$	$2^{\Theta(h)}k^{\Theta(1)}$	$2^{\Theta(k+h)}$
Hybrid, $a=\log(k+1)$	$k^{\Theta(1)}$	$k^{\Theta(h)}$	$k^{\Theta(h)}$
Hybrid, $a=\log(\frac{k}{h}+1)$	$\Theta((\frac{k}{h}+1)^{\Theta(h)}$	$\Theta((\frac{k}{h}+1)^{\Theta(h)})$	$(\frac{k}{h}+1)^{5h}k^{\Theta(1)}$

Pebbling uses $\Theta((k+1)^h)$ states. Hybrid is better when $h=\omega(k^{4/5})$.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a - 1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Each bit of $\text{Hybrid}_a(v)$ is a degree-$2a$ polynomial in $\text{Hybrid}_a(\ell)$ and $\text{Hybrid}_a(r)$.

Hybrid algorithm

The Hybrid encoding is broken into $\frac{k}{2^a-1}$ *blocks* that are a bits long.
For example, with $k = 9, a = 2$:

x	block 1	block 2	block 3	full encoding
1	01	00	00	010000
2	10	00	00	100000
3	11	00	00	110000
4	00	01	00	000100
5	00	10	00	001000
6	00	11	00	001100
7	00	00	01	000001
8	00	00	10	000010
9	00	00	11	000011

Each bit of $\mathrm{Hybrid}_a(v)$ is a degree-$2a$ polynomial in $\mathrm{Hybrid}_a(\ell)$ and $\mathrm{Hybrid}_a(r)$.
Using this, we can build an algorithm that uses 3 registers with $\frac{ka}{2^a-1}$ bits each and makes $2^{\Theta(a)}$ recursive calls at each level, for a total of $2^{\Theta(ah)}k^{\Theta(1)}$ layers.

Future work

- Improve the algorithm. (Better ways to compute d-ary products? We're not the first to want them.)
- Find new TEP lower bounds that apply to these algorithms. (Old lower bounds apply only to read-once or "thrifty" algorithms.)