Catalytic approaches to the Tree Evaluation Problem

James Cook, Ian Mertz

STOC 2020

Outline

The Tree Evaluation Problem

New algorithm

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]

The Tree Evaluation Problem

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

New algorithm

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]

The Tree Evaluation Problem

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

New algorithm

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]

The Tree Evaluation Problem (TEP)

Motivation

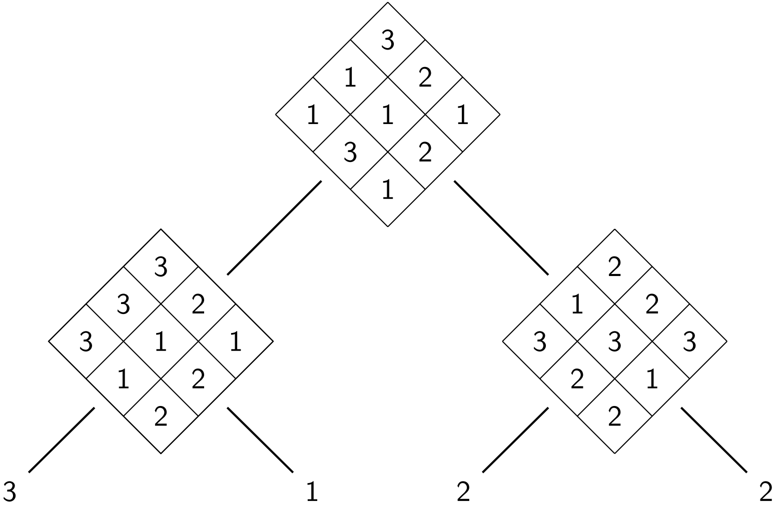
Fact

TEP $\in$ P

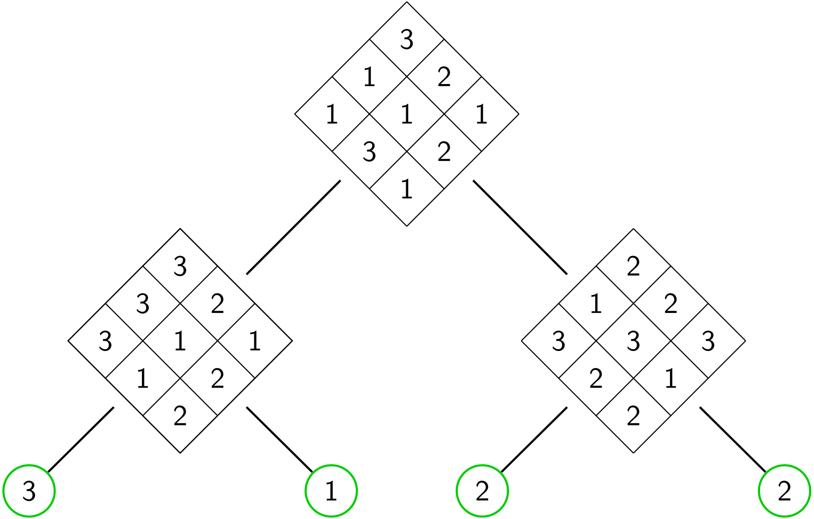
Conjecture

TEP $\notin$ L

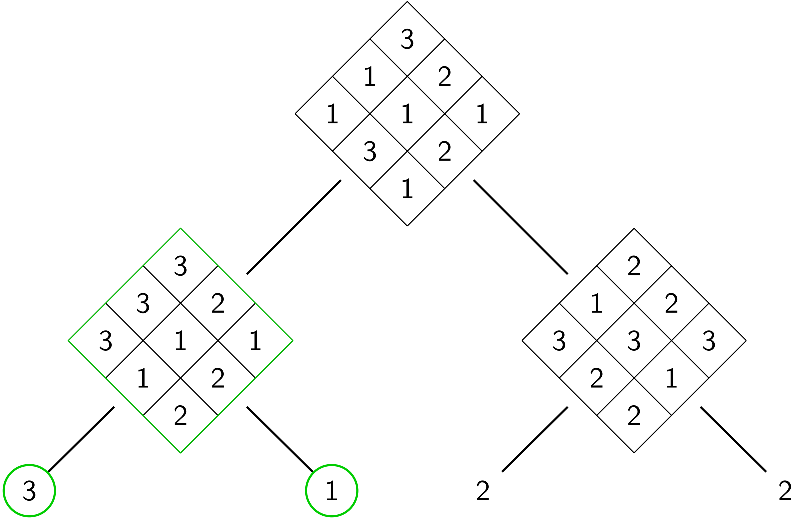
The Tree Evaluation Problem (TEP)



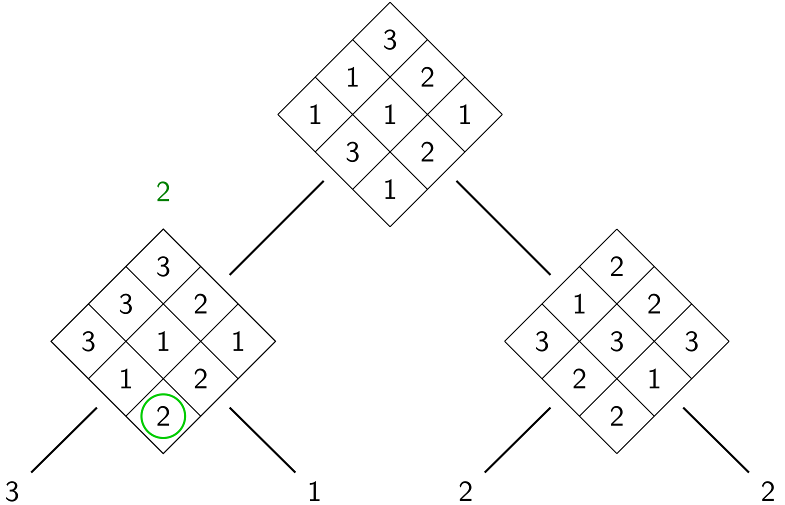
The Tree Evaluation Problem (TEP)



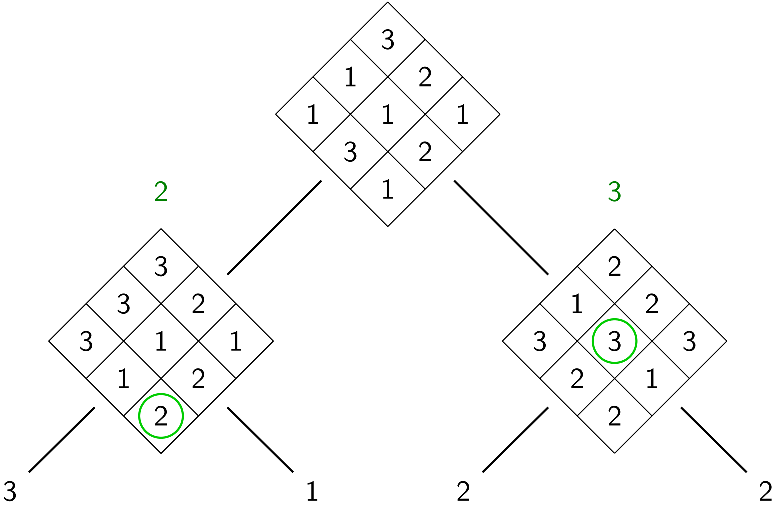
The Tree Evaluation Problem (TEP)



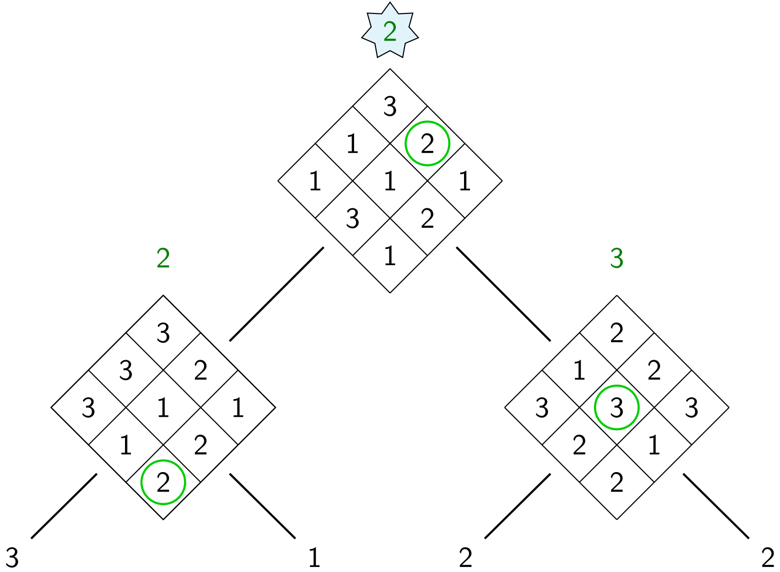
The Tree Evaluation Problem (TEP)



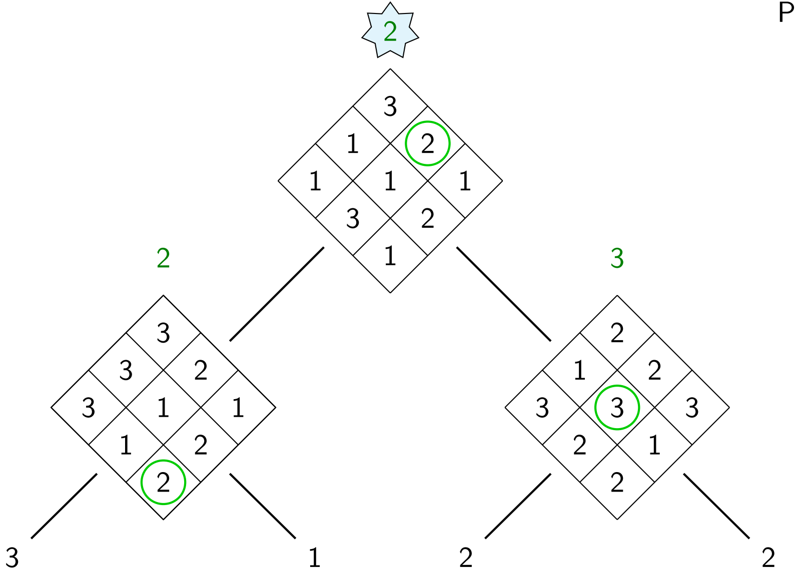
The Tree Evaluation Problem (TEP)



The Tree Evaluation Problem (TEP)



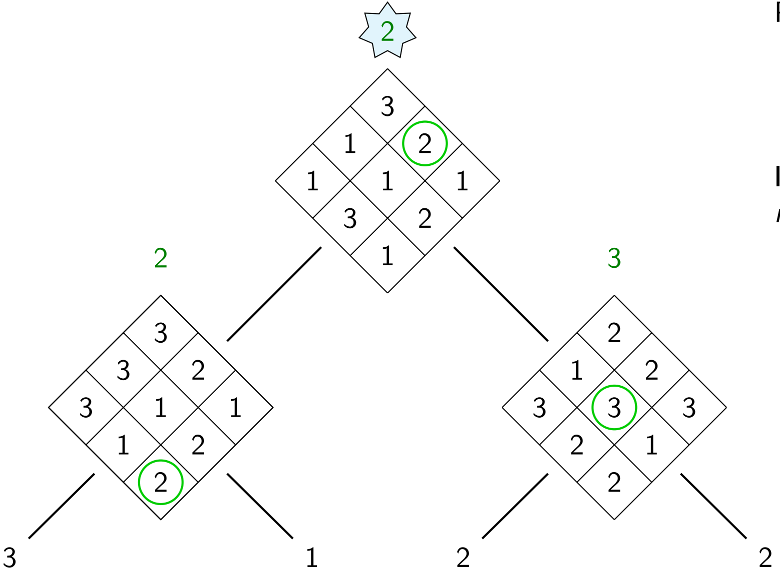
The Tree Evaluation Problem (TEP)



Parameters:

- height = 3
- k = 3

The Tree Evaluation Problem (TEP)



Parameters:

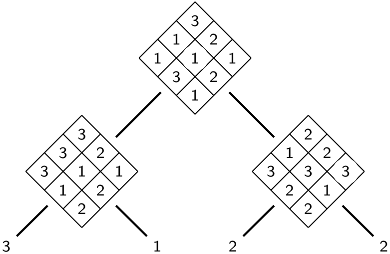
- height $= 3$
- k $= 3$

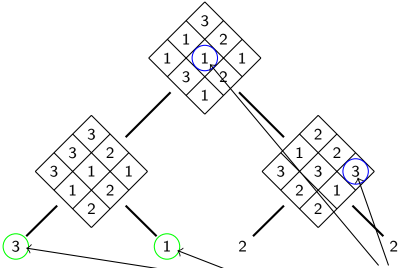
Input size:
$n = \Theta(2^h k^2 \log k)$ bits.

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

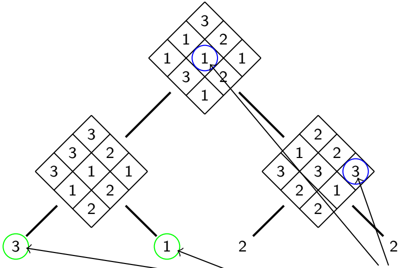
Conjecture

TEP $\notin$ L
In other words, it can't be solved in $O(h + \log k)$ space.


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.

Conjecture

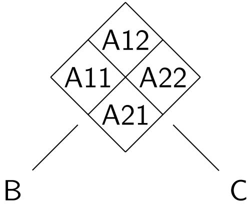
TEP $\notin$ L
In other words, it can't be solved in $O(h + \log k)$ space.

Conjecture

$\mathrm{TEP} \notin \mathrm{L}$

In other words, it can't be solved in $O(h + \log k)$ space.

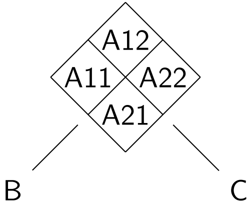
In other words, it can't be solved by a uniform family of branching programs with $2^{O(h)} k^{O(1)}$ states.

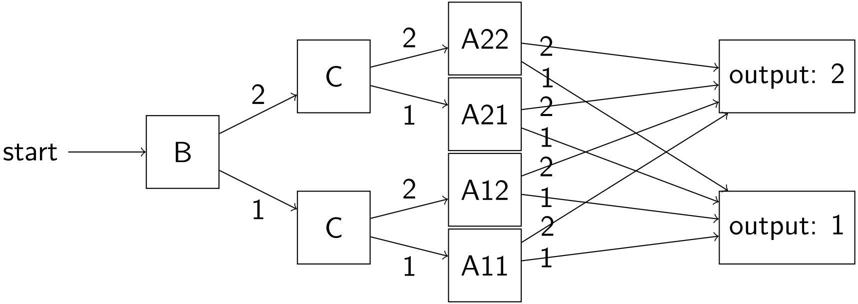

A12

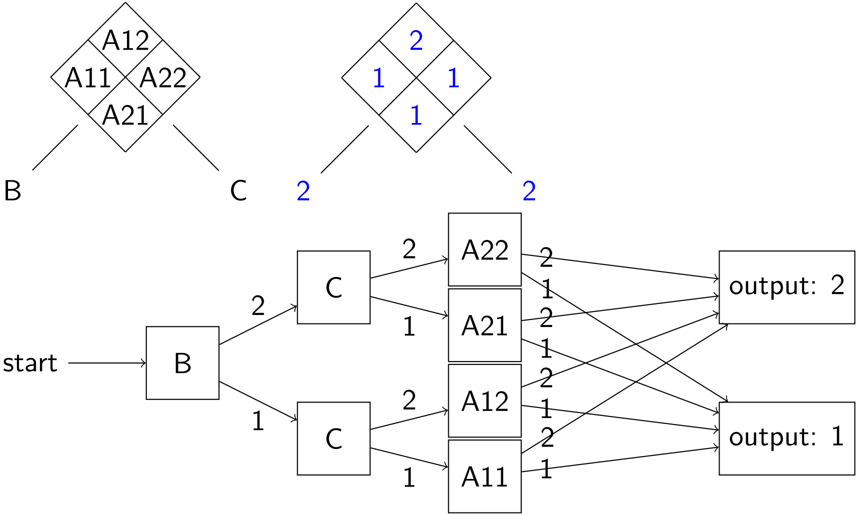
A11 A22

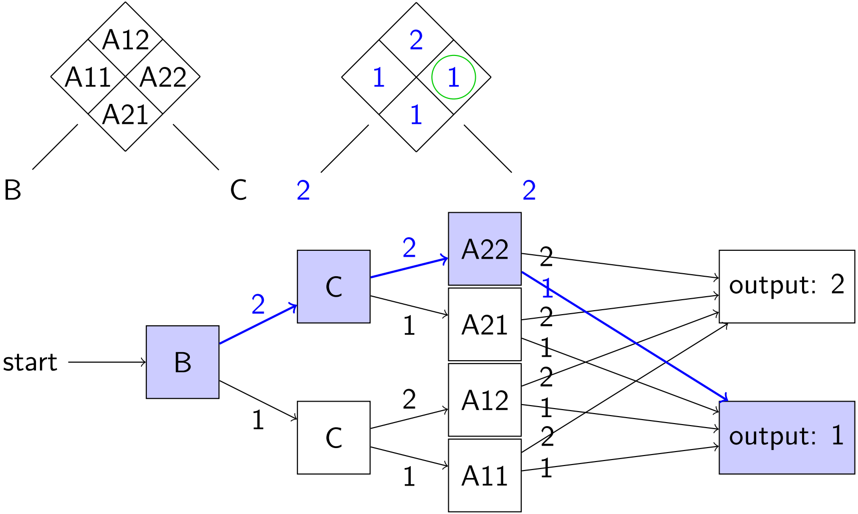
A21

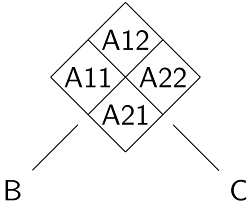
B C

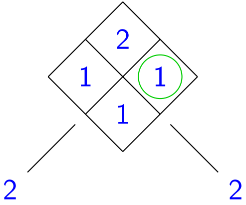

A12
A11
A22
A21
B
C

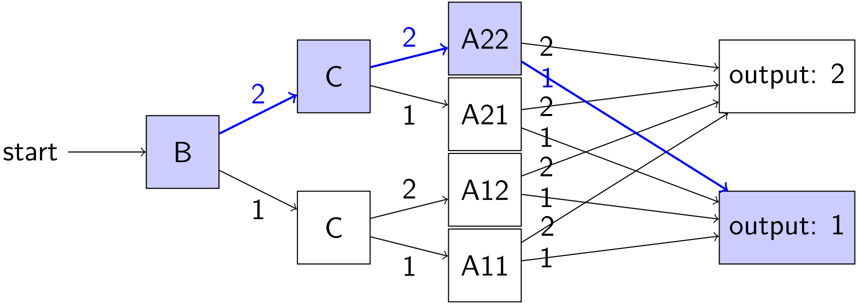

start
B
2
1
C
C
2
1
2
1
A22
A21
A12
A11
2
1
2
1
2
1
2
1
output: 2
output: 1


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





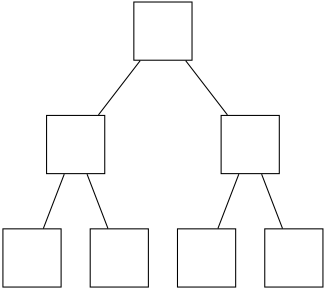


remember B

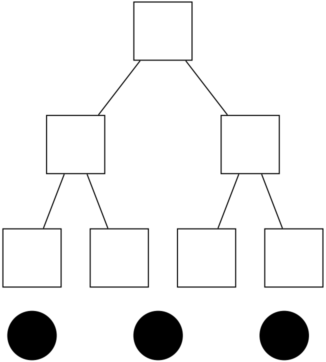
remember B, C

Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]

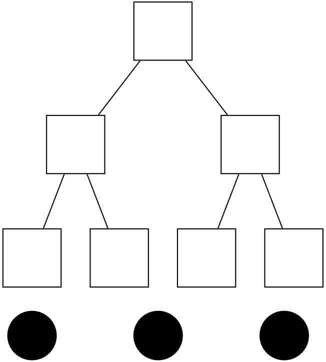


Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

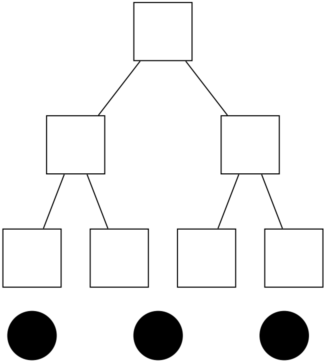
Pebbling game [Paterson Hewitt 1970]



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]

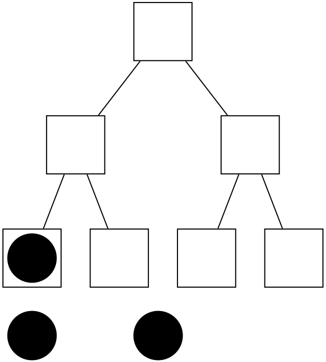


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

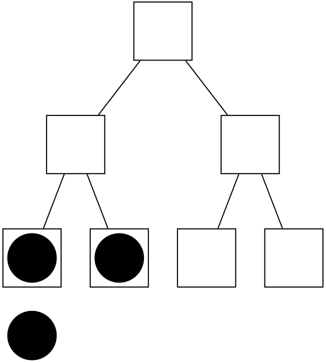


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

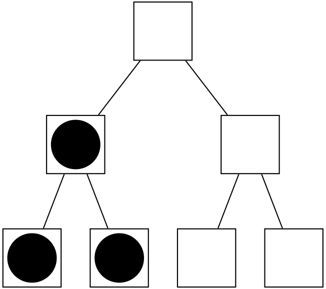


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

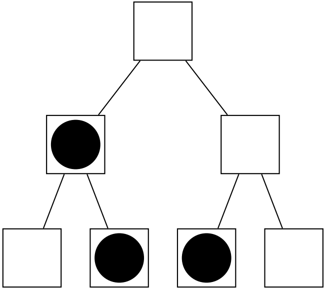


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

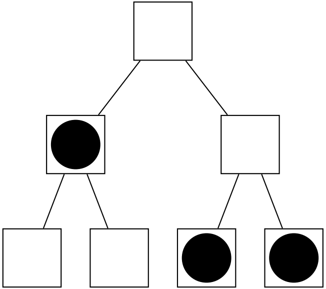


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

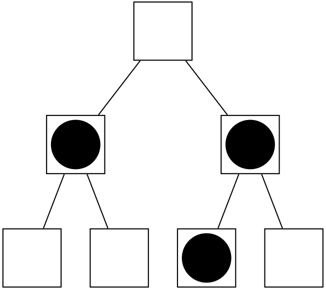


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

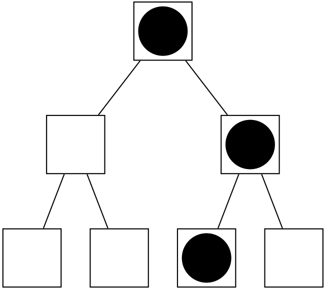


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]

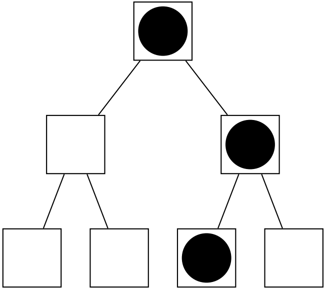


Limited supply of pebbles (say, 3).
Two kinds of move:

- 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.

Pebbling game [Paterson Hewitt 1970]



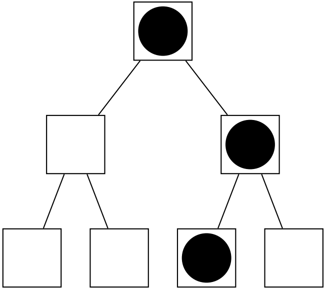
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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]



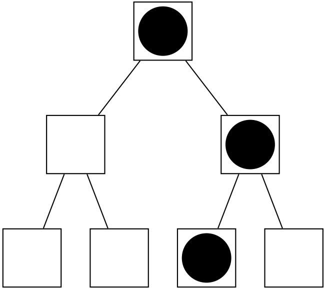
Limited supply of pebbles (say, 3).
Two kinds of move:

- 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:

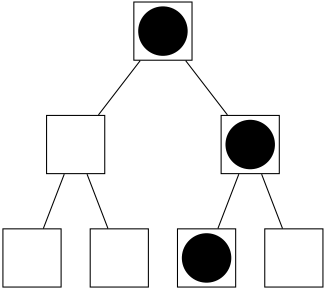
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

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.

Conjecture (TEP $\notin$ L)

TEP can’t be solved by a uniform family of branching programs with $2^{O(h)}k^{O(1)}$ states.

Algorithm (pebbling)

The *pebbling algorithm* uses $\Theta((k+1)^h)$ states.

Conjecture (false)

A branching program for TEP requires $\Omega(k^h)$ states.

Conjecture (TEP $\notin$ L)

TEP can't be solved by a uniform family of branching programs with $2^{O(h)}k^{O(1)}$ states.

Algorithm (pebbling)

The *pebbling algorithm* uses $\Theta((k+1)^h)$ states.

Conjecture (false)

A branching program for TEP requires $\Omega(k^h)$ states.

Algorithm (new)

Our new algorithm uses $(O(\frac{k}{h}))^{2h+\epsilon}k^{\Theta(1)}$ states.

New algorithm defeats $\Omega(k^h)$ conjecture when $h \geq k^{1/2+\epsilon'}$, but is still not log space.

The Tree Evaluation Problem

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

New algorithm

Lower bounds

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

Lower bounds

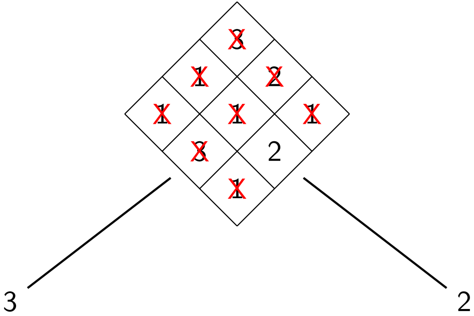
Solving TEP requires $\Omega(k^h)$ states (like the pebbling algorithm) if you assume. . .

- 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.



The Tree Evaluation Problem

New algorithm

- Reversible computation
- Solving TEP

The Tree Evaluation Problem

New algorithm

- Reversible computation
- Solving TEP

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, . . .

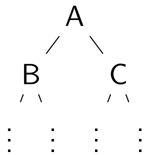
Catalytic space

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 $-f$.)

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 $-f$.)

ℓ instuctions $\Rightarrow$ branching program with $(\ell + 1)|R|^m$ states.

Example

Cleanly compute $x_1 + x_2$ into r_1:

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

Example

Cleanly compute $x_1 + x_2$ into r_1:

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

Example

Cleanly compute $x_1 + x_2$ into r_1:

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

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:

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

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:

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

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:

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

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:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$			
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:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
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}			
$r_3 \leftarrow r_3 + r_1 \times r_2$			

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:

	r_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$

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_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$

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_1	r_2	r_3
P_1	$\tau_1 + f_1$	τ_2	τ_3
$r_3 \leftarrow r_3 - r_1 \times r_2$	$\tau_1 + f_1$	τ_2	$\tau_3 - \tau_1 \times \tau_2 - f_1 \times \tau_2$
P_2			
$r_3 \leftarrow r_3 + r_1 \times r_2$	$\tau_1 + f_1$	$\tau_2 + f_2$	$\tau_3 + \tau_1 \times f_2 + f_1 \times f_2$
P_1^{-1}			
$r_3 \leftarrow r_3 - r_1 \times r_2$	τ_1	$\tau_2 + f_2$	$\tau_3 - \tau_1 \times \tau_2 + f_1 \times f_2$
P_2^{-1}			
$r_3 \leftarrow r_3 + r_1 \times r_2$	τ_1	τ_2	$\tau_3 + f_1 \times f_2$

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

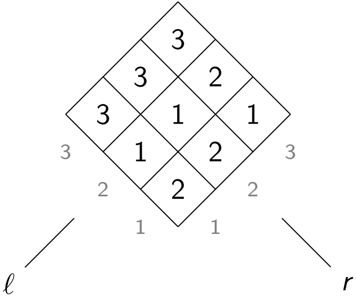
A formula for TEP

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:

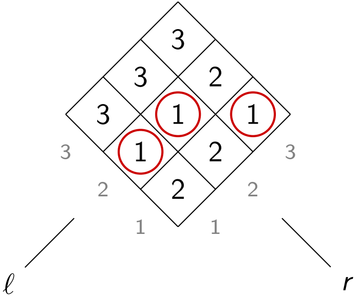


$[v = 1] =$

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:

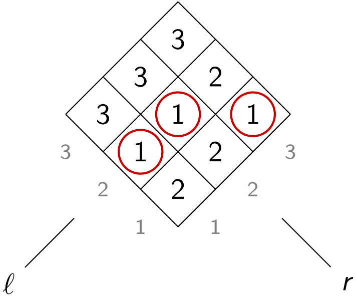


$$[v = 1] =$$

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:

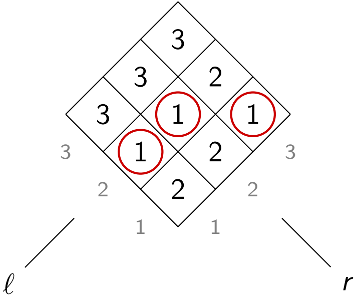


$$[v = 1] =$$
$$[\ell = 2] \times [r = 1] + [\ell = 2] \times [r = 2] + [\ell = 1] \times [r = 3]$$

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 algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

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 algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

Needs three registers total.

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 algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

Needs three registers total. Gives branching program with width 8 and length $(4k^2)^{h-1}$.

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 algorithm: 4 recursive calls each to CheckNode to compute $[\ell = y]$ and $[r = z]$, using two extra registers j and j'.

Needs three registers total. Gives branching program with width 8 and length $(4k^2)^{h-1}$.

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

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

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

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

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

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

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

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

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

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

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

$r_{j'} \leftarrow r_{j'} + [r = 2]$

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

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

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

$r_{j'} \leftarrow r_{j'} - [r = 2]$

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

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

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

$r_{j'} \leftarrow r_{j'} + [r = 3]$

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

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

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

$r_{j'} \leftarrow r_{j'} - [r = 3]$

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

$\cdots$

$\cdots$

$\cdots$

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

$$
\begin{array}{llll}
r_j \leftarrow r_j + [\ell = 1] & r_j \leftarrow r_j + [\ell = 1] & r_j \leftarrow r_j + [\ell = 1] & \\
r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & \\
r_{j'} \leftarrow r_{j'} + [r = 1] & r_{j'} \leftarrow r_{j'} + [r = 2] & r_{j'} \leftarrow r_{j'} + [r = 3] & \cdots \\
r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} & \cdots \\
r_j \leftarrow r_j - [\ell = 1] & r_j \leftarrow r_j - [\ell = 1] & r_j \leftarrow r_j - [\ell = 1] & \cdots \\
r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & r_i \leftarrow r_i - r_j \times r_{j'} & \\
r_{j'} \leftarrow r_{j'} - [r = 1] & r_{j'} \leftarrow r_{j'} - [r = 2] & r_{j'} \leftarrow r_{j'} - [r = 3] & \\
r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} & r_i \leftarrow r_i + r_j \times r_{j'} &
\end{array}
$$

Running in parallel reduces to 4 recursive calls instead of $4k^2$. The catch: need $3k$ registers instead of 3.

- Pebbling algorithm: $\Theta((k+1)^h)$ states.

- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^hk^2)$ states.

- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^h k^2)$ states.
  - Beats pebbling when $h \gg \dfrac{k}{\log k}$.

- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
    - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
    - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
    - Total $\Theta(2^{3k}4^hk^2)$ states.
    - Beats pebbling when $h \gg \dfrac{k}{\log k}$.
- "Binary encoding" algorithm:
    - Recursively compute $\log k$ bit vector representing node value.
    - $3\log k$ registers.

- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
  - Recursively computes k-bit vector $([v=1],[v=2],\ldots,[v=k])$.
  - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
  - Total $\Theta(2^{3k}4^h k^2)$ states.
  - Beats pebbling when $h \gg \dfrac{k}{\log k}$.
- "Binary encoding" algorithm:
  - Recursively compute $\log k$ bit vector representing node value.
  - $3\log k$ registers.
  - Degree $2\log k$ multiplication requires k^2 recursive calls instead of 4.
  - Total $k^{2h+\Theta(1)}$ states. (Always worse than pebbling.)

- Pebbling algorithm: $\Theta((k+1)^h)$ states.
- "One-hot encoding" algorithm:
    - Recursively computes k-bit vector $([v=1],[v=2],\dots,[v=k])$.
    - $3k$ registers. 4 recursive calls $\Rightarrow \Theta(4^h)k^2$ total steps.
    - Total $\Theta(2^{3k}4^h k^2)$ states.
    - Beats pebbling when $h \gg \dfrac{k}{\log k}$.
- "Binary encoding" algorithm:
    - Recursively compute $\log k$ bit vector representing node value.
    - $3\log k$ registers.
    - Degree $2\log k$ multiplication requires k^2 recursive calls instead of 4.
    - Total $k^{2h+\Theta(1)}$ states. (Always worse than pebbling.)
- "Hybrid encoding algorithm" interpolates between the two, and uses $(O(\frac{k}{h}))^{2h+\epsilon}k^{\Theta(1)}$ states.
    - Beats pebbling when $h \geq k^{1/2+\epsilon'}$.

Conclusion

- We present a new algorithm for TEP: first improvement over classic "pebbling" algorithm since the problem was introduced in 2010.
- Still might be possible to prove TEP $\notin$ L, implying P $\neq$ L.

Conclusion

- We present a new algorithm for TEP: first improvement over classic "pebbling" algorithm since the problem was introduced in 2010.
- Still might be possible to prove TEP $\notin$ L, implying P $\neq$ L.

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.)

Thanks!