

# Catalytic approaches to the Tree Evaluation Problem

*James Cook, Ian Mertz*

STOC 2020

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

Catalytic approaches to the Tree Evaluation Problem

James Cook, Ian Mertz

STOC 2020

Hello from Toronto, Canada!

This video is an overview of a paper by Ian Mertz and myself, about a new space-efficient algorithm for the Tree Evaluation Problem.

The video is divided into two parts.

# Outline

The Tree Evaluation Problem

New algorithm

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition

Outline

The Tree Evaluation Problem

New algorithm

In the first part, I'll talk about the Tree Evaluation problem. It was introduced in an attempt to separate complexity classes: the problem can easily be solved in polynomial time, but it seems impossible to solve in low-memory classes like log space.

In the second part, I'll show you a new algorithm for solving this problem with limited memory. This algorithm gives the first space improvement since the problem was originally introduced ten years ago, and it makes use of some techniques for re-using memory using reversible computations.

## 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]

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition

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 first part is older work mostly done by other people.  
It's based on a couple of papers from 2010 that introduced the problem.

## 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]

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition

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]

I'll start by describing the problem and its motivation. Then I'll talk about a couple of abstractions we use to analyse it, called branching programs and pebbling games. And finally, before I move on the new algorithm, I'll talk about some lower bounds that the our algorithm had to work around.



## 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]

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition

## 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]

So, let's start with the motivation.

# The Tree Evaluation Problem (TEP)

## Motivation

Fact

$\text{TEP} \in \text{P}$

Conjecture

$\text{TEP} \notin \text{L}$

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)

## The Tree Evaluation Problem (TEP)

### Motivation

#### Fact

TEP  $\in$  P

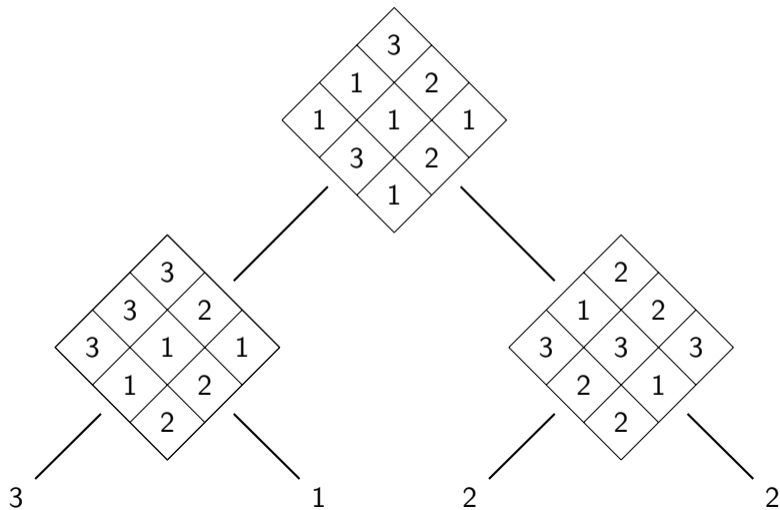
#### Conjecture

TEP  $\notin$  L

The Tree Evaluation Problem, or TEP for short is easy to solve in polynomial time, but it's conjectured that you can't solve it in log space. The goal is to prove this conjecture, implying that L is not equal to P. In fact, the gap it aims to close is a little narrower than that: it's in log CFL but conjectured not to be in NL. But we'll just focus on P and L in this video.

So, that's the motivation. Now let's talk about what this problem actually is.

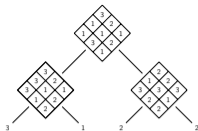
# The Tree Evaluation Problem (TEP)



## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)

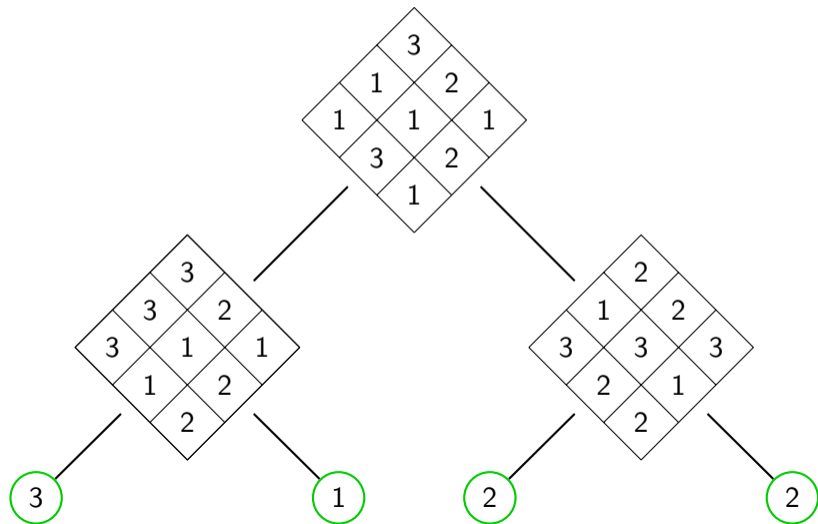
The Tree Evaluation Problem (TEP)



The input to the Tree Evaluation Problem is a complete binary tree with some information attached to each node. Each leaf has a number attached to it — in this case, 3, 1, 2 and 2 — and each internal node has a table of numbers.

Given that input, we're going to recursively define a single number at each node, called the value of the node.

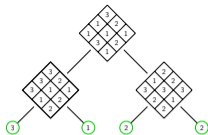
# The Tree Evaluation Problem (TEP)



## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)

The Tree Evaluation Problem (TEP)

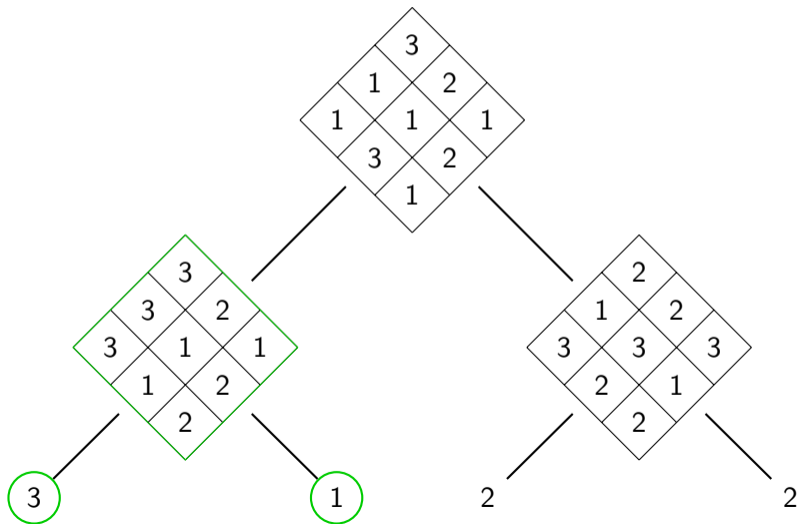


The values of the leaves are already part of the input.

To compute the value of an internal node, we need to first know the values of its children.



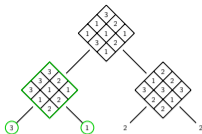
# The Tree Evaluation Problem (TEP)



## Catalytic approaches to the Tree Evaluation Problem

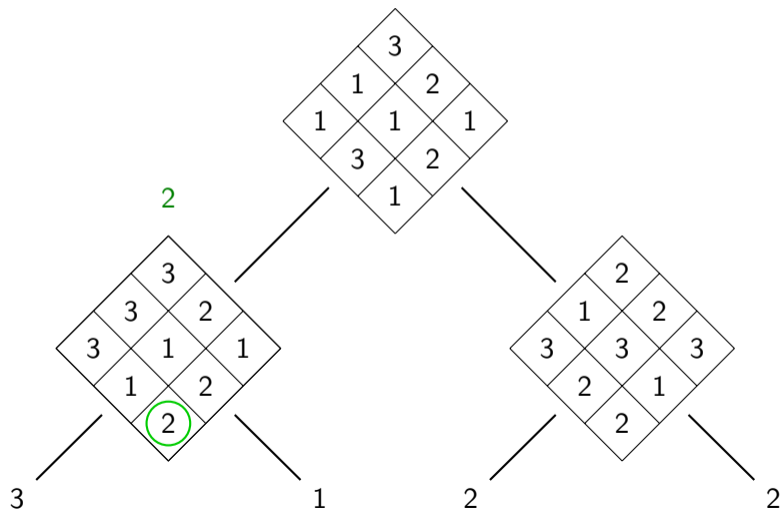
- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)

The Tree Evaluation Problem (TEP)



For example, let's look at the left child of the root. The values of its two children tell us where to look in its table. In this case, we look at row three, column one, and we find the number two.

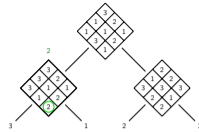
# The Tree Evaluation Problem (TEP)



## Catalytic approaches to the Tree Evaluation Problem

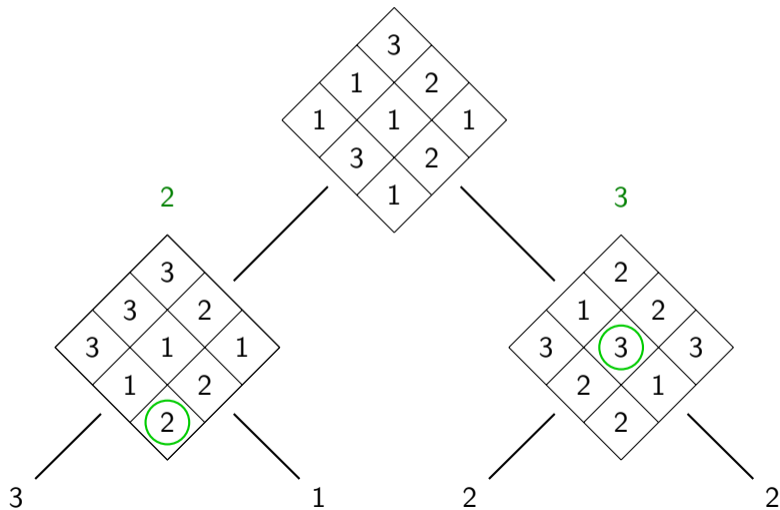
- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)

The Tree Evaluation Problem (TEP)



Similarly, we look up row two column two of the node on the right, and find the number three.

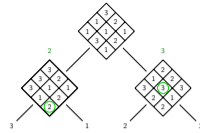
# The Tree Evaluation Problem (TEP)



## Catalytic approaches to the Tree Evaluation Problem

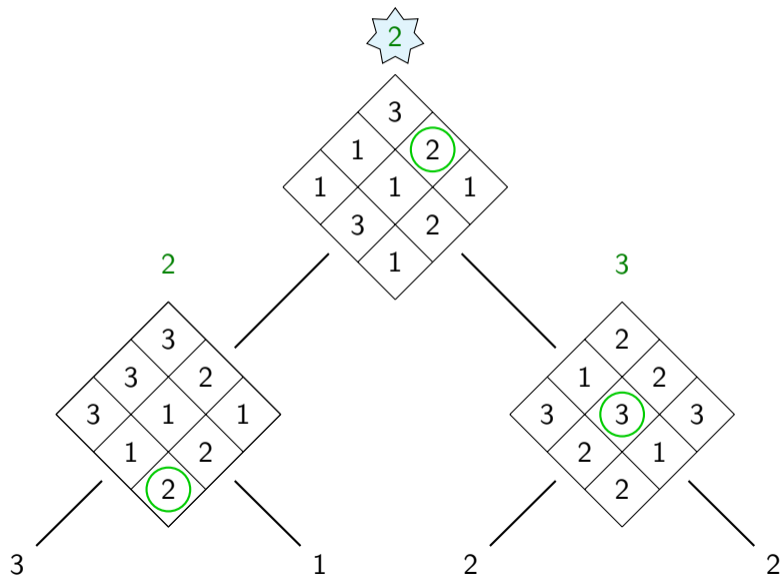
- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)

The Tree Evaluation Problem (TEP)



Finally, the numbers two and three tell us where to look in the root node, and we find the number two.

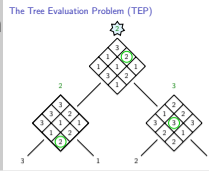
# The Tree Evaluation Problem (TEP)



2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

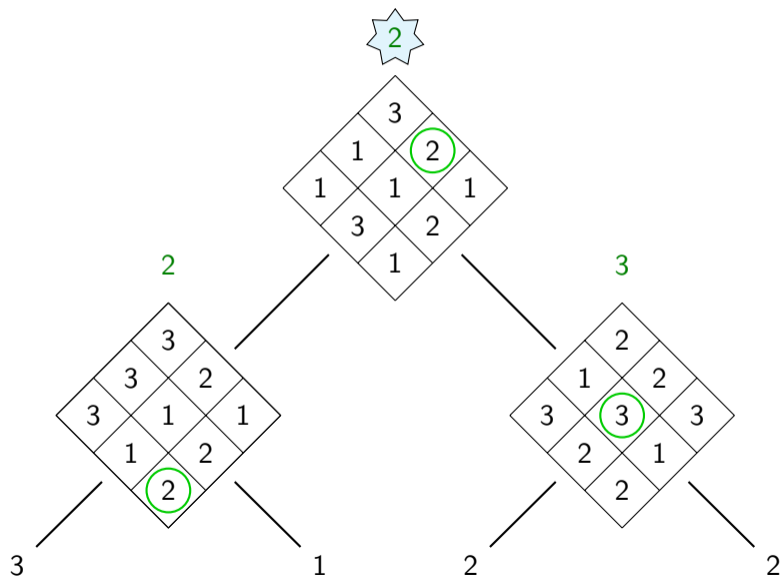
- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)



The output of the Tree Evaluation Problem is the value at the root.



# The Tree Evaluation Problem (TEP)

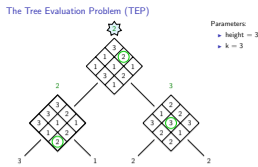


Parameters:

- ▶ height = 3
- ▶  $k = 3$

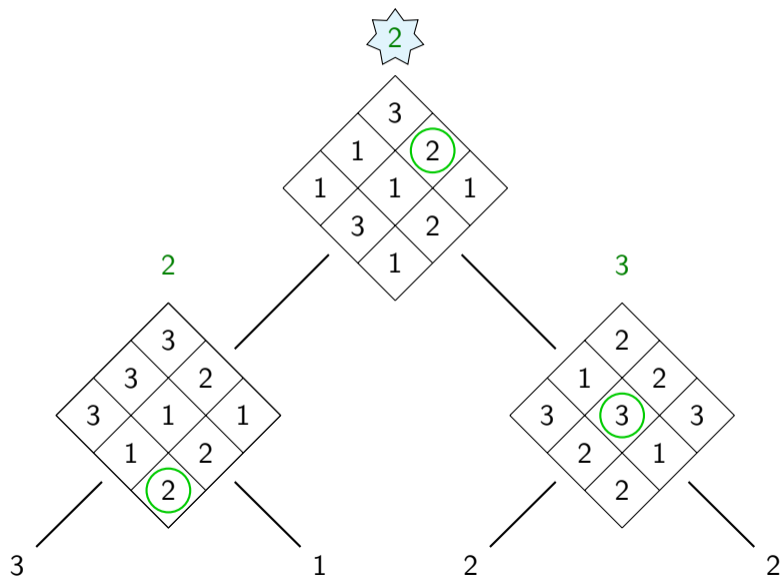
## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition
    - └ The Tree Evaluation Problem (TEP)



There are two parameters to this problem. The first is the height of the tree. Three in this case. The second parameter is  $k$ , which is the range of the numbers at the nodes. In this case it's also three, meaning every number is between one and three, and the tables are all three by three.

# The Tree Evaluation Problem (TEP)



Parameters:

- ▶ height = 3
- ▶  $k = 3$

Input size:

$$n = \Theta(2^h k^2 \log k) \text{ bits.}$$

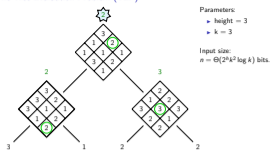
## Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

## └└ Motivation and definition

## └└└ The Tree Evaluation Problem (TEP)

The Tree Evaluation Problem (TEP)



The size of the input to TEP is on the order of two to the  $h$  internal nodes, times  $k$  squared numbers stored in each node, times  $\log k$  bits to store each number.

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.

## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Motivation and definition

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

Conjecture

TEP  $\notin L$

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

Using this formula for the input size, we can rephrase the conjecture I showed you earlier.

Saying TEP is not in L is the same as saying it can't be solved in big oh of  $h$  plus  $\log k$  space.

## The Tree Evaluation Problem

Motivation and definition

Branching programs and pebbling games

Lower bounds

New algorithm

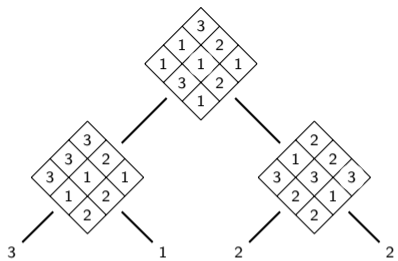
# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games

The Tree Evaluation Problem  
Minimax and Defenders  
Branching programs and pebbling games  
Lower bounds  
New algorithms

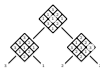
Now that I've defined the Tree Evaluation Problem, I want to talk about algorithms for solving it. I'll start by describing branching programs, which are the computational model we're using. Then I'll talk about an abstraction called a *pebbling game* which can be useful for both upper and lower bounds.



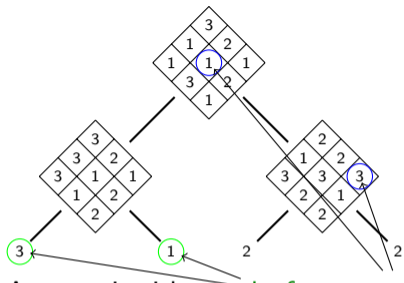


## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games



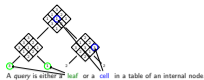
So, here's our TEP input again. I'll define a *query* to be any piece of that input we might want to read.



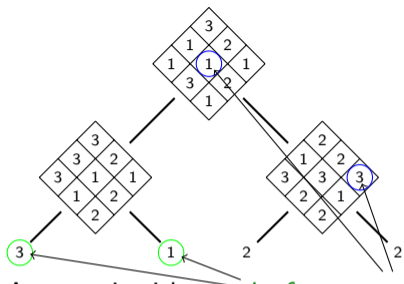
A query is either a **leaf** or a **cell** in a table of an internal node.

## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games



Specifically, a query is either a leaf, meaning we want to read the input at that leaf, or it's a particular cell in one of the tables in 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:

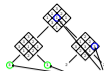
- ▶ *query state*: labelled with a query and has  $k$  outgoing edges labelled with the possible answers.
- ▶ *final state*: labelled with a number  $1..k$ .

One state is the starting state.

## Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

## └└ Branching programs and pebbling games



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.

A branching program is a directed graph, where the nodes are called states. There are two kinds of state.

A query state is labelled with a query, and has  $k$  outgoing edges: the edge you follow depends on the answer to the query.

The other kind is a final state. When you get to one of those, the computation stops, and you output whatever the state is labelled with.

One of the states is marked as the starting state, where computation begins.

## Conjecture

TEP  $\notin$  L

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

## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games

## Conjecture

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

Let's return to our lower bound conjecture. We've written it as: TEP can't be solved in big oh of  $h + \log k$  space.

Any Turing machine can be transformed into a uniform family of branching programs, with one state for each possible configuration.



## Conjecture

TEP  $\notin$  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.

## Catalytic approaches to the Tree Evaluation Problem

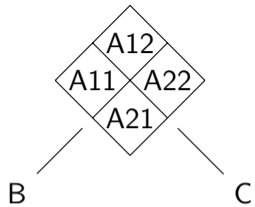
- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games

## Conjecture

TEP  $\notin$  LIn 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.

So, we can rephrase our conjecture one more time: TEP can't be solved by a uniform family of branching programs with only two to the order  $h$  times a polynomial in  $k$  states. We could also state the conjecture without the uniformity condition.

Now, let's look at an example of a branching program for solving TEP. To keep it small, we'll set both the height and the alphabet size to 2.

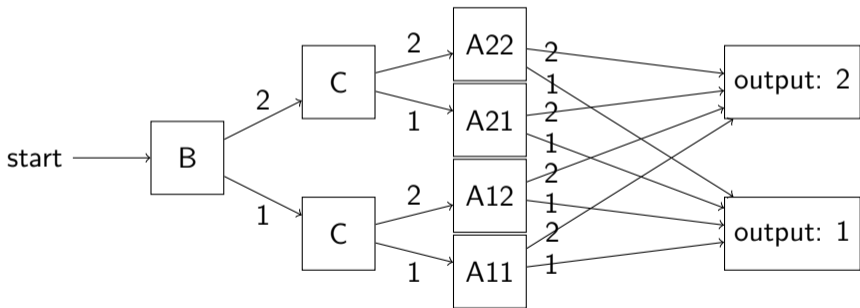
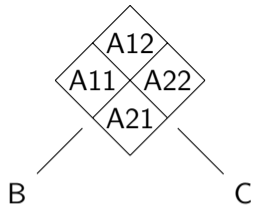


## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games



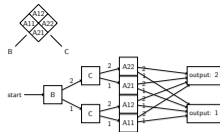
When both  $h$  and  $k$  are two, an input to TEP is structured like this. There are six things we can query: the four cells in the root node  $A$ 's table, and the two leaves  $B$  and  $C$ .



## Catalytic approaches to the Tree Evaluation Problem

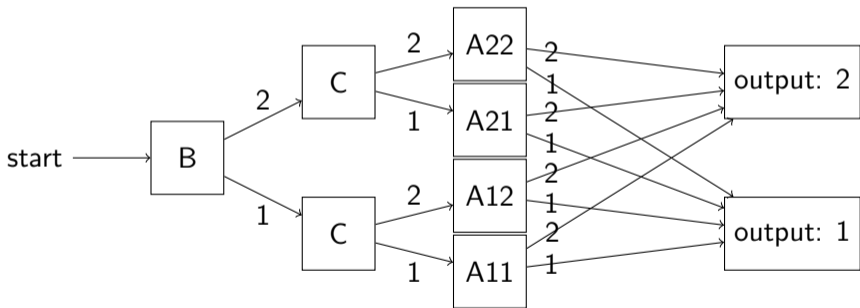
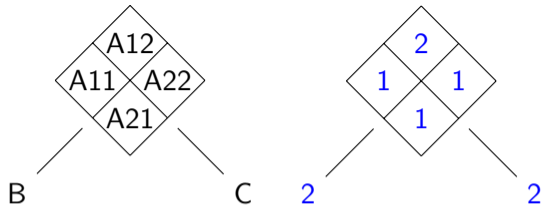
## └ The Tree Evaluation Problem

## └ Branching programs and pebbling games



Here's a branching program that solves it. It's organized into *layers* going from left to right.

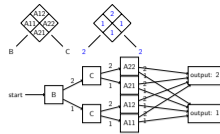
The starting state queries the first leaf, B. Depending on the answer, we end up in one of the two states in the next layer. Those states query the other leaf C, and depending on the answer, we end up in one of four possible states in the third layer. Each node in the third layer queries a different cell in the root node's table, and depending on the answer, we output 1 or 2.



## Catalytic approaches to the Tree Evaluation Problem

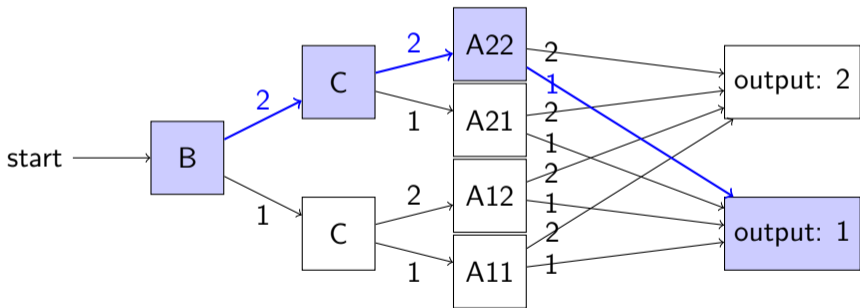
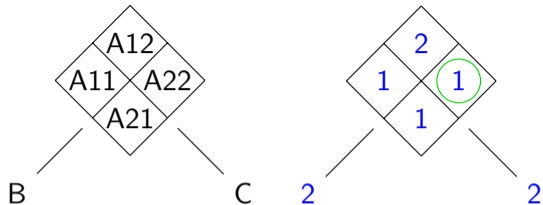
## └ The Tree Evaluation Problem

## └└ Branching programs and pebbling games



Here's an example input. Let's see what the computation looks like.

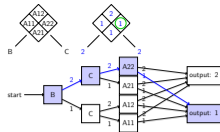




## Catalytic approaches to the Tree Evaluation Problem

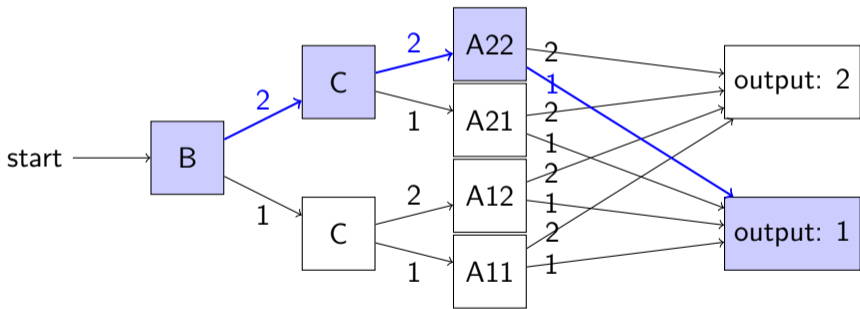
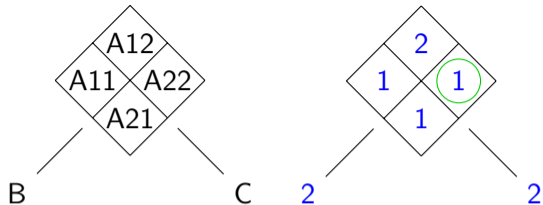
## └ The Tree Evaluation Problem

## └└ Branching programs and pebbling games



Both the leaves are 2, so we end up at the node that queries A22. Then the value is 1, so we output 1.

One thing to notice here is that every layer remembers a different set of information.



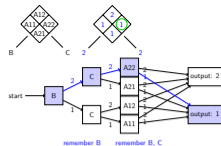
remember B

remember B, C

## Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

## └ Branching programs and pebbling games



In the second layer, we remember node  $B$ , and in the third layer, we remember both  $B$  and  $C$ . All the lower bounds we have so far for TEP involve arguments about how many things the branching program needs to remember at once.

One way to model this idea of remembering things is pebbling games.

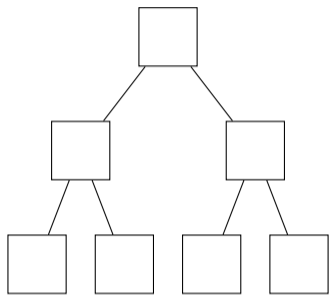
# Pebbling game [Paterson Hewitt 1970]

## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games
    - └ Pebbling game [Paterson Hewitt 1970]

Pebbling games were first defined by Paterson and Hewitt in 1970. In the context of the Tree Evaluation Problem, they work like this. Suppose we have a complete binary tree of height  $h$ .

## Pebbling game [Paterson Hewitt 1970]



2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games
    - └ Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]

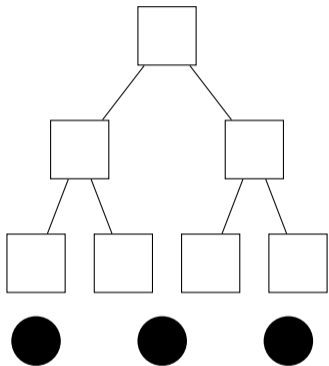


Three in this case.



## Pebbling game [Paterson Hewitt 1970]

Limited supply of pebbles (say, 3).



# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games
    - └ Pebbling game [Paterson Hewitt 1970]

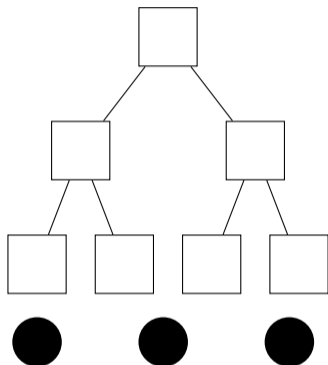
Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

You have some limited number of pebbles. Let's say it's also three. They all start in your hand. You're allowed two kinds of move.

## 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.

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Branching programs and pebbling games
    - └ Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]



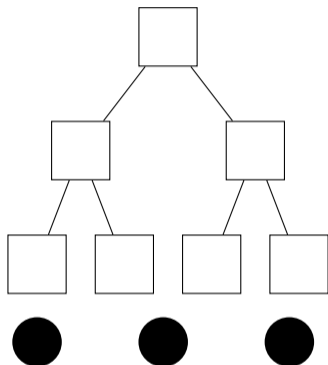
Limited supply of pebbles (say, 3).

Two kinds of move:

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

First, you can move one of your pebbles to a leaf of the tree. And second, if a node's two children both have pebbles on them, you can move one of your pebbles to that node. The goal is to place a pebble on the root 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.

# Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

### └└ Branching programs and pebbling games

#### └└└ Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

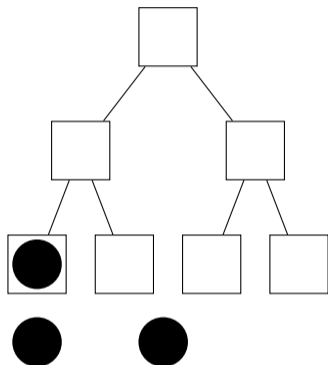
Two kinds of move:

- 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.

Here's a sequence of moves that does this. We start with the leaves and work our way up.

## Pebbling game [Paterson Hewitt 1970]



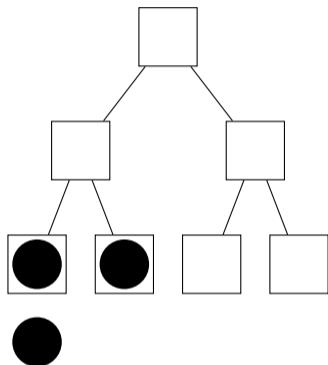
Limited supply of pebbles (say, 3).

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).

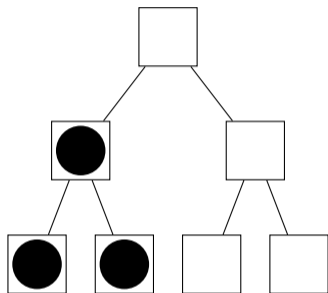
Two kinds of move:

- ▶ 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]



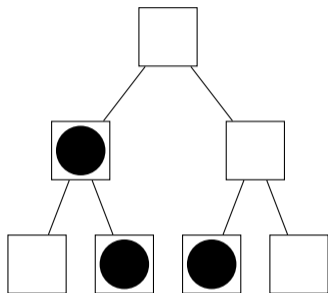
Limited supply of pebbles (say, 3).

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]



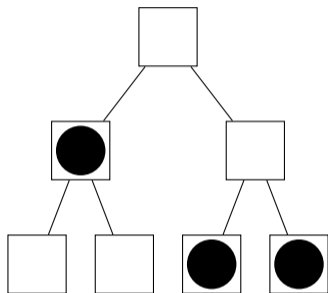
Limited supply of pebbles (say, 3).

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]



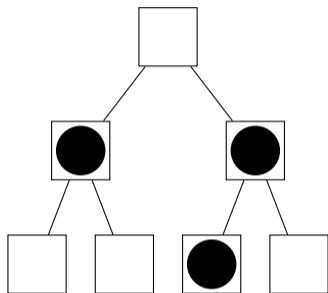
Limited supply of pebbles (say, 3).

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]



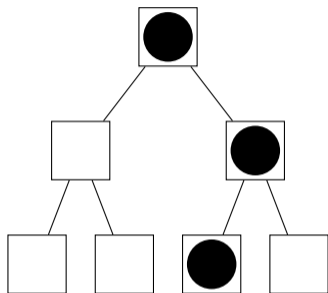
Limited supply of pebbles (say, 3).

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.

# Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

### └└ Branching programs and pebbling games

#### └└└ Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

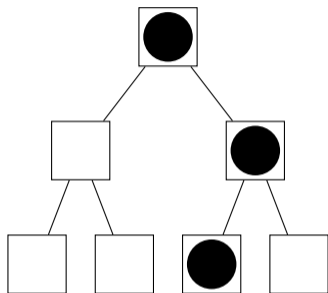
Two kinds of move:

- 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.

We've succeeded, because there's now a pebble on the root node. The important question is: how many pebbles do we need? In this case we had three pebbles, and it was 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.


# Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

### └└ Branching programs and pebbling games

#### └└└ Pebbling game [Paterson Hewitt 1970]

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.

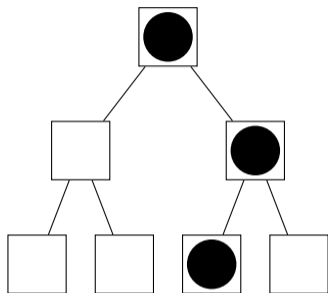
Goal: put a pebble on the root.

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

In general, you can solve this game with  $h$  pebbles, where  $h$  is the height of the tree, using a simple recursive algorithm. The algorithm visits each node once, so that's two to the  $h$  minus one steps.



## 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.

## Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

## └└ Branching programs and pebbling games

## └└└ Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]



Limited supply of pebbles (say, 3).

Two kinds of move:

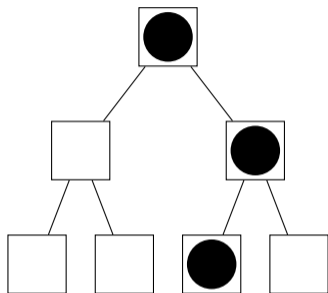
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

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

A corollary of that is that we can build a branching program that solves the tree evaluation problem using two to the  $h$  times  $k$  to the  $h$  states. Each step of the game translates into a layer of the branching program, and the placement of the pebbles determines which values the program is remembering. Since our strategy uses at most  $h$  pebbles at a time, the program will only need to remember at most  $h$  values at once, which requires  $k$  to the power  $h$  states in a single layer. Now, the pebbling strategy is tight: if you only have  $h-1$  pebbles, no sequence of legal moves can put one 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.

Corollary: A branching program with  $2^h k^h$  states can solve TEP.

Theorem:  $h$  pebbles are needed.


## Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

## └└ Branching programs and pebbling games

## └└└ Pebbling game [Paterson Hewitt 1970]

Pebbling game [Paterson Hewitt 1970]



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

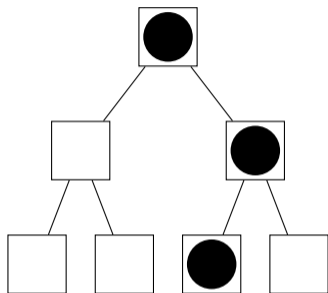
- Move a pebble to a leaf.
- If a node's two children have pebbles, move a pebble to that node.

Goal: put a pebble on the root.

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

The proof of that is not as obvious. I'll leave it as an exercise. Now, it would be nice if we could make a corresponding corollary.

## 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.

## Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

## └└ Branching programs and pebbling games

## └└└ Pebbling game [Paterson Hewitt 1970]

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.

Since we need at least  $h$  pebbles, maybe we can prove that the tree evaluation problem needs at least on the order of  $k$  to the  $h$  states. We'll see in a moment that this would imply that log space is not equal to polytime.

For a long time, nobody could come up with any algorithm that did better, so this conjecture seemed quite plausible.

The algorithm I'll present later is the first counterexample.

Let's take a look at where we are.

## Conjecture ( $\text{TEP} \notin \text{L}$ )

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.

# Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

### └ Branching programs and pebbling games

#### Conjecture (TEP $\notin L$ )

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

#### Algorithm (pebbling)

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

#### Conjecture (false)

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

We started with this conjecture that TEP is not in L, meaning two to the order  $h$  times poly  $k$  states isn't enough.

We saw our first algorithm. If you analyse it carefully, it turns out the pebbling algorithm uses on the order of  $k$  plus one to the power  $h$  states. And the pebbling framework led to a conjectured lower bound of  $k$  to the  $h$ .



## 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.

# Catalytic approaches to the Tree Evaluation Problem

## └ The Tree Evaluation Problem

### └ Branching programs and pebbling games

#### 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  $O(\frac{1}{\epsilon})^{2^{h+\epsilon}}k^{O(1)}$  states.

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

The new algorithm I'm going to show you has on the order of  $k$  over  $h$  to the power two  $h$  plus an arbitrarily small constant times a polynomial in  $k$  states.

This defeats the conjectured lower bound of  $k$  to the  $h$  whenever  $h$  is not too small compared to  $k$ . Specifically, if  $h$  is  $k$  to a power bigger than one half, this algorithm is an asymptotic improvement.

But, it's still not a log space algorithm, so the door is still open to using TEP as a way to separate  $L$  from  $P$ .

## The Tree Evaluation Problem

Motivation and definition

Branching programs and pebbling games

Lower bounds

New algorithm

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Lower bounds

## The Tree Evaluation Problem

Motivation and definition

Branching programs and pebbling games

**Lower bounds**

New algorithm

Now, I want to briefly mention some existing lower bounds for TEP, to give you an idea of why we found this new algorithm surprising.

## Lower bounds

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

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Lower bounds

## Lower bounds

Solving TEP requires  $\Omega(k^4)$  states (like the pebbling algorithm) if you assume...

It turns out that under some pretty reasonable-sounding assumptions, you can prove that the pebbling algorithm is essentially the best possible.

## Lower bounds

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

- ▶ the algorithm is *read-once*

# Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Lower bounds

## Lower bounds

Solving TEP requires  $\Omega(k^4)$  states (like the pebbling algorithm) if you assume...

- the algorithm is *read-once*

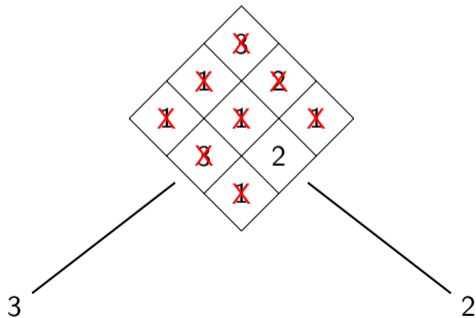
You can prove it if you assume the algorithm is *read-once*. That means that once the algorithm reads a certain piece of the input, it is not allowed to read it again.



## 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 irrelevant piece of the input.



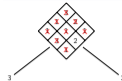
## Catalytic approaches to the Tree Evaluation Problem

- └ The Tree Evaluation Problem
  - └ Lower bounds

## 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 irrelevant piece of the input.



Another assumption we can make instead is that the algorithm is *thrifty*. This means that the algorithm never reads an irrelevant piece of the input. For example, if an internal node's left child has value three and its right child has value 2, then it's only allowed to read the entry at position three two in that node's table, since none of the other entries matter. Our new algorithm beats this lower bound of  $k$  to the  $h$ , so, as you may have already inferred, it's not read-once or thrifty. Our algorithm is actually going to read every piece of the input several times. I've said a lot of mysterious things about the algorithm, so maybe it's time I told you how it works.

## The Tree Evaluation Problem

### New algorithm

Reversible computation

Solving TEP

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation

The Tree Evaluation Problem

New algorithm  
Reversible computation  
Solving TEP

This part of the video has two pieces.

I'll begin with some techniques we use related to reversible computation, and then I'll tell you how we apply them to solve TEP.

## The Tree Evaluation Problem

### New algorithm

Reversible computation

Solving TEP

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation

The Tree Evaluation Problem

New algorithm  
Reversible computation  
Solving TEP

The first thing I want to tell you about is a paper that caught our attention, and showed us that reversible computation is something we should be looking at.

# 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 approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Catalytic space

## 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

The paper is from 2014, and it's called *Computing with a full memory: catalytic space*.

The idea is that you're given a small amount of ordinary memory to work with, and a much larger amount of extra memory. The catch with the extra memory is that it starts out filled with data, possibly incompressible, and once you're done with your computation, you need to return it back the way it was.

Surprisingly, the authors found that the extra memory seems to help.



## 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 approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Catalytic space

## 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, ...

With only a logarithmic amount of ordinary memory but a polynomial amount of borrowed memory, you can compute many things not known to be computable in log space, such as the determinant of a matrix, or anything in nondeterministic log space.

We stumbled on this result when we were trying to prove a lower bound for the Tree Evaluation Problem.

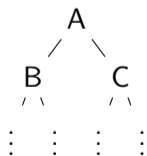
# Catalytic space

*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.

## Catalytic approaches to the Tree Evaluation Problem

- └─ New algorithm
  - └─ Reversible computation
    - └─ Catalytic space

## 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.

We had the following idea for a proof. First, at some point you need to compute the left child of the root: node B in this diagram. Then you need to keep that in memory while you compute the right child, C. That uses up  $\log k$  bits of memory in addition to the subroutine that's computing C. Therefore, the argument goes, every level you add to the tree adds  $\log k$  bits that your algorithm needs to remember.

The catalytic space result effectively shows that this approach will never work. Even if we could argue that you need to remember B while you're computing C, this result says that the subroutine computing C can borrow the memory being used to store B.

The history of the techniques we use goes back pretty far.

*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]

# Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Reversible computation

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

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

A 1989 paper by Barrington showed that if you restrict branching programs to have just five nodes in every layer, you can still do a lot with them. A later 1992 paper by Ben-Or and Cleve showed how you can do a lot with register programs that only use three registers.

Both of these papers show how you can trade time for space in order to make algorithms that use an extremely limited amount of memory.

Another thing they have in common is that they use reversible operations. The basic ingredient in our algorithm is reversible operations on registers.

Ring  $R$

Inputs  $x_1, \dots, x_n \in R$

Work registers  $r_1, \dots, r_m \in R$

Reversible instructions:

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Reversible computation

Ring  $R$   
Inputs  $x_1, \dots, x_n \in R$   
Work registers  $r_1, \dots, r_m \in R$   
Reversible instructions:  
▶ Example:  $r_5 \leftarrow r_5 + r_1 \times x_1$ .  
▶ Inverse is  $r_5 \leftarrow r_5 - r_1 \times x_1$ .

The model is that we have  $n$  inputs,  $x$  one through  $x$   $n$ , and  $m$  work registers  $r$  one through  $r$   $m$ , and their values are all in some ring  $R$ . We're interested in reversible instructions. For example, the first instruction here adds register four times input 1 to register five. We can reverse that instruction by subtracting instead of adding. When you run these two instructions in sequence, it's the same as doing nothing.



Ring  $R$

Inputs  $x_1, \dots, x_n \in R$

Work registers  $r_1, \dots, r_m \in R$

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:  $\tau_j$  denotes the starting value of register  $r_j$ .

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Reversible computation

Ring  $R$   
Inputs  $x_1, \dots, x_n \in R$   
Work registers  $r_1, \dots, r_m \in R$   
Reversible instructions:  
▶ Example:  $r_1 \leftarrow r_2 + r_1 \times x_1$ .  
▶ Inverse is  $r_2 \leftarrow r_2 - r_1 \times x_1$ .  
Notation:  $\tau_j$  denotes the starting value of register  $r_j$ .

For any register  $r_j$ , let's define  $\tau_j$  to be its initial value before our computation begins.

Now, suppose we have some function  $f$  we're interested in computing. I'm going to define something called *cleanly computing*  $f$ .

Ring  $R$

Inputs  $x_1, \dots, x_n \in R$

Work registers  $r_1, \dots, r_m \in R$

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:  $\tau_j$  denotes the starting value of register  $r_j$ .

## Definition

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

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Reversible computation

Ring  $R$   
 Inputs  $x_1, \dots, x_n \in R$   
 Work registers  $r_1, \dots, r_m \in R$   
 Reversible instructions:  
 ▶ Example:  $r_3 \leftarrow r_2 + r_1 \times x_1$ .  
 ▶ Inverse is  $r_3 \leftarrow r_2 - r_1 \times x_1$ .  
 Notation:  $\tau_j$  denotes the starting value of register  $r_j$ .

## Definition

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

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

A sequence of reversible instructions *cleanly computes* a function  $f$  into register  $i$  if, once the computation finishes, the new value of register  $i$  is its old value  $\tau_i$  plus  $f$ , and every other register is unchanged:  $r_j$  equals  $\tau_j$  for  $j$  not equal to  $i$ . Note that we're allowed to use these other registers, as long we make sure to undo all our changes.

Ring  $R$

Inputs  $x_1, \dots, x_n \in R$

Work registers  $r_1, \dots, r_m \in R$

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:  $\tau_j$  denotes the starting value of register  $r_j$ .

## Definition

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

- ▶  $r_i = \tau_i + f(x_1, \dots, x_n)$
- ▶ 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  $-f$ .)

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Reversible computation

Ring  $R$   
 Inputs  $x_1, \dots, x_n \in R$   
 Work registers  $r_1, \dots, r_m \in R$   
 Reversible instructions:

- ▶ Example:  $r_3 \leftarrow r_2 + r_1 \times x_1$ .
- ▶ Inverse is  $r_3 \leftarrow r_2 - r_1 \times x_1$ .

Notation:  $r_j$  denotes the starting value of register  $r_j$ .

## Definition

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

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

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

Since each instruction is reversible, we can reverse the entire sequence by running the inverses of the original instructions in reverse order. If we do that, the result is a clean computation of negative  $f$ .

There are two reasons we like this definition. The first is that it's designed to help us re-use memory, as we'll see later. The second reason is we can translate register programs into branching programs.

Ring  $R$

Inputs  $x_1, \dots, x_n \in R$

Work registers  $r_1, \dots, r_m \in R$

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:  $\tau_j$  denotes the starting value of register  $r_j$ .

## Definition

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

- ▶  $r_i = \tau_i + f(x_1, \dots, x_n)$
- ▶ 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  $-f$ .)

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Reversible computation

Ring  $R$   
 Inputs  $x_1, \dots, x_n \in R$   
 Work registers  $r_1, \dots, r_m \in R$   
 Reversible instructions:

- ▶ Example:  $r_3 \leftarrow r_2 + r_1 \times x_1$ .
- ▶ Inverse is  $r_3 \leftarrow r_2 - r_1 \times x_1$ .

Notation:  $r_j$  denotes the starting value of register  $r_j$ .

**Definition**

A sequence of reversible instructions cleanly computes  $f$  into  $r_j$  if, once it finishes:

- ▶  $r_j = r_j + f(x_1, \dots, x_n)$
- ▶ all other registers are unchanged ( $r_j = r_j$  for  $j \neq i$ )

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

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

If we can cleanly compute  $f$  with  $m$  registers and  $\ell$  instructions, then we can turn that into a branching program with  $\ell$  plus one layers, each containing  $R$  to the  $m$  states in order to remember all the register values. So, we can design our algorithm using register instructions and then convert it to a branching program. Now, let's try some examples of clean computation.



## Example

Cleanly compute  $x_1 + x_2$  into  $r_1$ :

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Reversible computation

## Example

Cleanly compute  $x_1 + x_2$  into  $r_1$ :

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

For our first example, suppose we want to cleanly compute  $x$  one plus  $x$  two into register one. We can do this with two instructions: first add  $x$  one, then add  $x$  two.

## Example

Cleanly compute  $x_1 + x_2$  into  $r_1$ :

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation

## Example

Cleanly compute  $x_1 + x_2$  into  $r_1$ :

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

After we add  $x$  one, the value of the register is tau one plus  $x$  one.

## Example

Cleanly compute  $x_1 + x_2$  into  $r_1$ :

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation

## Example

Cleanly compute  $x_1 + x_2$  into  $r_1$ :

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

And after we add  $x$  two, the value of the register is tau one plus  $x$  one plus  $x$  two. By definition, we've cleanly computed  $x_1$  plus  $x_2$  into  $r_1$ .

## 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:

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

For our next example, let's say we've got a subroutine  $P_1$  that cleanly computes a function  $f_1$ , and a subroutine  $P_2$  that cleanly computes a function  $f_2$ , and our goal is to compute the product  $f_1$  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:

$P_1$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$P_2$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$

$P_1^{-1}$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$P_2^{-1}$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

```
P1
r1 ← r1 - r1 × r2
P2
r2 ← r2 + r1 × r2
P1-1
r1 ← r1 - r1 × r2
P2-1
r3 ← r2 + r1 × r2
```

The program looks like this. We can think of it as being made out of two interlocking pieces.

## 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:

$P_1$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$P_2$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$

$P_1^{-1}$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$P_2^{-1}$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

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

The first piece is calling the subroutines P one and P two. We first call P one, then P two. Since everything's made out of reversible instructions, we're then able to run P one backward and P two backward.

## 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:

$P_1$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$P_2$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$

$P_1^{-1}$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$P_2^{-1}$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

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

The other piece is adding and subtracting  $r$  one times  $r$  two. Since the subroutines are modifying the contents of  $r$  one and  $r$  two, this has a different effect each time. So, let's see what happens when we run the program.

## 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:

$$\begin{array}{r} P_1 \\ r_1 \quad r_2 \quad r_3 \\ \tau_1 + f_1 \quad \tau_2 \quad \tau_3 \end{array}$$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$$P_2$$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$

$$P_1^{-1}$$

$$r_3 \leftarrow r_3 - r_1 \times r_2$$

$$P_2^{-1}$$

$$r_3 \leftarrow r_3 + r_1 \times r_2$$





## 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:

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

$P_1$	$r_1$	$r_2$	$r_3$
$r_1 \leftarrow r_1 - r_1 \times r_2$	$r_1 + f_1$	$r_2$	$r_3$
$P_2$	$r_1 + f_1$	$r_2$	$r_3 - r_1 \times r_2 - f_1 \times r_2$
$r_2 \leftarrow r_2 + r_1 \times r_2$			
$P_1^{-1}$			
$r_3 \leftarrow r_3 - r_1 \times r_2$			
$P_2^{-1}$			
$r_1 \leftarrow r_1 + r_1 \times r_2$			

The next instruction subtracts two terms from register three, leaving a value of tau 3 minus tau 1 times tau 2 minus f one times 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:

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

$P_1$	$r_1$	$r_2$	$r_3$
$r_1 \leftarrow r_1 - r_1 \times r_2$	$r_1 + f_1$	$r_2$	$r_3$
$P_2$	$r_1 + f_1$	$r_2$	$r_3 - r_1 \times r_2 - f_1 \times r_2$
$r_2 \leftarrow r_2 + r_2 \times r_1$	$r_1 + f_1$	$r_2 + f_2$	$r_3 + r_1 \times f_2 + f_1 \times f_2$
$P_1^{-1}$	$r_1 + f_1$	$r_2 + f_2$	$r_3 - r_1 \times r_2 + f_1 \times f_2$
$r_1 \leftarrow r_1 - r_1 \times r_2$	$r_1$	$r_2 + f_2$	$r_3 - r_1 \times r_2 + f_1 \times f_2$
$P_2^{-1}$	$r_1$	$r_2$	$r_3 + f_1 \times f_2$
$r_2 \leftarrow r_2 + r_2 \times r_1$	$r_1$	$r_2$	$r_3 + f_1 \times f_2$

As the program continues, different terms are added and subtracted from register 3.

## 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:

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

$P_1$	$r_1$	$r_2$	$r_3$
$r_1 \leftarrow r_1 - r_1 \times r_2$	$r_1 + f_1$	$r_2$	$r_3$
$P_2$	$r_1 + f_1$	$r_2 + f_2$	$r_3 - r_1 \times r_2 - f_1 \times r_2$
$r_2 \leftarrow r_2 + r_2 \times r_1$	$r_1 + f_1$	$r_2 + f_2$	$r_3 + r_1 \times f_2 + f_1 \times f_2$
$P_1^{-1}$	$r_1 + f_1$	$r_2 + f_2$	$r_3 + r_1 \times r_2 + f_1 \times f_2$
$r_1 \leftarrow r_1 - r_1 \times r_2$	$r_1$	$r_2 + f_2$	$r_3 - r_1 \times r_2 + f_1 \times f_2$
$P_2^{-1}$	$r_1$	$r_2$	$r_3 + f_1 \times f_2$
$r_2 \leftarrow r_2 + r_2 \times r_1$	$r_1$	$r_2$	$r_3 + f_1 \times f_2$

At the end, register three holds its original value plus  $f_1$  times  $f_2$ , and the other registers have been restored. That means we've succeeded.

## 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:

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

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Reversible computation
    - └ Lemma: Multiplication

## 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:

$P_1$	$r_1$	$r_2$	$r_3$
$r_1 \leftarrow r_1 - r_1 \times r_2$	$r_1 + f_1$	$r_2$	$r_3$
$P_2$	$r_1 + f_1$	$r_2 + f_2$	$r_3 - r_1 \times r_2 - f_1 \times f_2$
$r_2 \leftarrow r_2 + r_2 \times r_1$	$r_1 + f_1$	$r_2 + f_2$	$r_3 + r_1 \times f_2 + f_1 \times f_2$
$P_1^{-1}$	$r_1 + f_1$	$r_2 + f_2$	$r_3 + r_1 \times f_2 + f_1 \times f_2$
$r_1 \leftarrow r_1 - r_1 \times r_2$	$r_1 + f_1$	$r_2 + f_2$	$r_3 - r_1 \times r_2 + f_1 \times f_2$
$P_2^{-1}$	$r_1 + f_1$	$r_2 + f_2$	$r_3 - r_1 \times r_2 + f_1 \times f_2$
$r_2 \leftarrow r_2 + r_2 \times r_1$	$r_1 + f_1$	$r_2$	$r_3 + f_1 \times f_2$

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

Now, we've computed  $f_1$  times  $f_2$ , but what did it cost us? Well, we had to make four subroutine calls:  $P_1$  and  $P_2$  forward and backward. But, this algorithm is extremely efficient with memory. Notice that  $P_1$  and  $P_2$  are allowed to use all of our memory, as long as they restore it when they're done. There is no memory that needs to be set aside for the parent routine's exclusive use. I like to think of these programs as "borrowing" the memory they use.

Now let's talk about how to apply these techniques to solving the Tree Evaluation Problem.



## The Tree Evaluation Problem

### New algorithm

Reversible computation

Solving TEP

2021-10-26

# Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

The Tree Evaluation Problem

New algorithm

Reversible computation

Solving TEP

This is the last part of the video.

We want to build a reversible computation to compute the value at the root node of the tree. In order to do that, it will be helpful to have an algebraic formula for that root value.

## A formula for TEP

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

2021-10-26

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ A formula for TEP

A formula for TEP

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

From here on, our ring will be the integers mod two, meaning registers will store bits. I'll introduce some notation: brackets  $x$  equals  $y$  is an *indicator* which equals one if they are equal and otherwise zero.



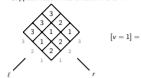
## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ A formula for TEP

## 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  $r$ :



Now, suppose we have some node  $v$  with two children,  $l$  and  $r$ , and this is the table at that node. Let's try to build a formula for the indicator  $v$  equals one.



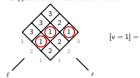
## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ A formula for TEP

## 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  $r$ :



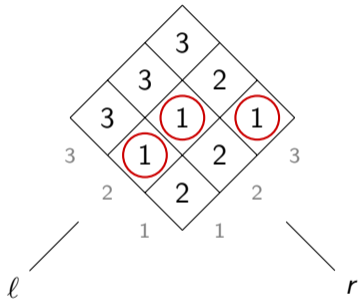
Well, there are three ways that node  $v$  can be equal to one, corresponding to the three times one appears in the table at node  $v$ . We can turn this into a formula with three terms.



## 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  $\ell$  and  $r$ :



$$[v = 1] =$$

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ A formula for TEP

## A formula for TEP

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

Suppose node  $v$  has children  $l$  and  $r$ :



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

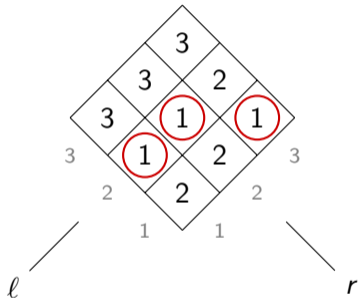
The terms say: either  $l$  equals 2 and  $r$  equals 1, or  $l$  equals 2 and  $r$  equals 2, or  $l$  equals 1 and  $r$  equals 3.

Now let's write the general formula.

## 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  $\ell$  and  $r$ :



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

Let  $f_v$  denote  $v$ 's table. In general,

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

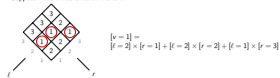
## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ A formula for TEP

## A formula for TEP

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

Suppose node  $v$  has children  $\ell$  and  $r$ :



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

Let  $t_v$  denote  $v$ 's table. In general,

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

Let's say  $t_v$  is the table of values at node  $v$ .

In general, we take the sum over all possible values  $y$  and  $z$  for the two children. Inside the sum, we check node  $v$ 's table to see whether each term should be included. We multiply that indicator by the indicators  $\ell$  equals  $y$  and  $r$  equals  $z$ .

With that formula in hand, let's try to build a recursive algorithm.

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

---

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ First attempt

First attempt

$$[v = x] = \sum_{(y,z) \in \{0,1\}^2} [(v,y,z) = x] \times [t = y] \times [r = z]$$

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

I've left our formula at the top of the slide for reference. Our algorithm's goal is to compute the formula, which determines whether node  $v$  has value  $x$ .

The algorithm is parameterized by the node  $v$ , the value  $x$ , and some target register  $i$ . If node  $v$  has value  $x$ , it will flip the bit in register  $i$ . In other words, it assigns  $r_i$  plus the indicator  $v$  equals  $x$  to  $r_i$ .

## 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.

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ First attempt

First attempt

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

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.

If  $v$  is a leaf node, then the value of  $v$  is directly available as part of the input. So, we can do this in just 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  $j'$ .

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ First attempt

## First attempt

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

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  $(y, z) \in [k]^2$ :
  - $r_i \leftarrow r_i + [l(y, z) = x] \times [l = y] \times [r = z]$   
using multiplication algorithm: 4 recursive calls each to CheckNode to compute  $[l = y]$  and  $[r = z]$ , using two extra registers  $j$  and  $j'$ .

If  $v$  is an internal node, then we compute this formula by looping over all  $k$  squared possible values for  $y$  and  $z$  and adding each term to  $r_i$  one at a time.

Each term includes a product of the indicators  $l$  equals  $y$  times  $r$  equals  $z$ , which we compute using the multiplication algorithm. This requires four recursive calls to CheckNode and two auxiliary 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.

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ First attempt

First attempt

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

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  $(y, z) \in [k]^2$ :
  - $r_i \leftarrow r_i + [c(y, z) = x] \times [t = y] \times [r = z]$   
using multiplication algorithm: 4 recursive calls each to CheckNode to compute  $[t = y]$  and  $[r = z]$ , using two extra registers  $j$  and  $j'$ .

Needs three registers total.

We use a total of three registers: register  $i$  holds our output, and two more registers  $j$  and  $j'$  are required by the multiplication algorithm. Since we're using clean computations, the calls to the subroutine are free to use those same three registers, so we really don't need any more than three registers, including all the recursive calls.

## 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  $(4k^2)^{h-1}$ .

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ First attempt

## First attempt

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

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

- If  $v$  is a leaf:
  - $r_i \leftarrow r_i \pm [v = x]$  is one instruction.
- else: for  $(y, z) \in [k]^2$ :
  - $r_i \leftarrow r_i + [(l(y,z) = x) \wedge [l = y] \wedge [r = z]]$  using multiplication algorithm: 4 recursive calls each to CheckNode to compute  $[l = y]$  and  $[r = z]$ , using two extra registers  $j$  and  $j'$ .

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

If we convert this to a branching program, those three one-bit registers translate to eight states in each layer. The length of the program is four  $k$  squared to the power  $h$  minus one, since at every level, we make four  $k$  squared recursive calls.

This isn't very good.

## 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.

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ First attempt

## First attempt

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

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

- If  $v$  is a leaf:
  - $r_i \leftarrow r_i \pm [v = x]$  is one instruction.
- else: for  $(y, z) \in [k]^2$ :
  - $r_i \leftarrow r_i \pm [l(y,z) = x] \times [l = y] \times [r = z]$   
using multiplication algorithm: 4 recursive calls each to CheckNode to compute  $[l = y]$  and  $[r = z]$ , using two extra registers  $j$  and  $j'$ .

Needs these registers total. Gives branching program with width 8 and length  $(4k^2)^{h-1}$ .  
Worse than pebbling, which uses  $\Theta((k+1)^h)$  states.

Our original pebbling algorithm just uses  $k$  plus one to the  $h$  states. So, we'll need another trick if we're going to beat it. Let's take a closer look at what's going on in this for loop.



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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

Each iteration of the for loop is using the multiplication lemma to combine the indicators  $\ell$  equals  $y$  and  $r$  equals  $z$ .

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

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

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

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

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

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

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

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

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

```

r ← r + [l = 1]
r ← r - r × rl
r ← r + [r = 1]
r ← r + r × rl
r ← r - [l = 1]
r ← r - r × rl
r ← r - [r = 1]
r ← r + r × rl

```

If you remember the multiplication lemma, it looks kind of like this. We make four calls to our subroutines for checking  $l$  and  $r$ , and in between those four calls, we update our final output register  $r$ . I've coloured the recursive calls in blue.

► for  $(y, z) \in [k]^2$ :

►  $r_i \leftarrow r_i + [f_v(y, z) = x] \times [\ell = y] \times [r = z]$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

...

...

...

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

$r \leftarrow r + [l = 1]$	$r \leftarrow r + [l = 1]$	$r \leftarrow r + [l = 1]$	...
$r \leftarrow r - r \times r$	$r \leftarrow r - r \times r$	$r \leftarrow r - r \times r$	...
$r \leftarrow r + [r = 1]$	$r \leftarrow r + [r = 2]$	$r \leftarrow r + [r = 3]$	...
$r \leftarrow r + r \times r$	$r \leftarrow r + r \times r$	$r \leftarrow r + r \times r$	...
$r \leftarrow r - [l = 1]$	$r \leftarrow r - [l = 1]$	$r \leftarrow r - [l = 1]$	...
$r \leftarrow r - r \times r$	$r \leftarrow r - r \times r$	$r \leftarrow r - r \times r$	...
$r \leftarrow r - [r = 1]$	$r \leftarrow r - [r = 2]$	$r \leftarrow r - [r = 3]$	...
$r \leftarrow r + r \times r$	$r \leftarrow r + r \times r$	$r \leftarrow r + r \times r$	...

The for loop just means we do this whole thing over and over again,  $k$  squared times.

It turns out we can completely parallelize this. All of the instructions on the first row can be run at the same time, with one recursive call that checks all of the possible values for the left child. We can do similar things for the other lines.

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

$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	$r_j \leftarrow r_j + [\ell = 1]$	
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	
$r_{j'} \leftarrow r_{j'} + [r = 1]$	$r_{j'} \leftarrow r_{j'} + [r = 2]$	$r_{j'} \leftarrow r_{j'} + [r = 3]$	...
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	...
$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	$r_j \leftarrow r_j - [\ell = 1]$	...
$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	$r_i \leftarrow r_i - r_j \times r_{j'}$	
$r_{j'} \leftarrow r_{j'} - [r = 1]$	$r_{j'} \leftarrow r_{j'} - [r = 2]$	$r_{j'} \leftarrow r_{j'} - [r = 3]$	
$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	$r_i \leftarrow r_i + r_j \times r_{j'}$	

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

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

• for  $(y, z) \in [k]^2$ :

- $r_1 \leftarrow r_1 + [k, (y, z) = x] \times [l = y] \times [r = z]$

$r_1 \leftarrow r_1 + [l = 1]$	$r_1 \leftarrow r_1 + [l = 1]$	$r_1 \leftarrow r_1 + [l = 1]$	...
$r_1 \leftarrow r_1 - r_2 \times r_3$	$r_1 \leftarrow r_1 - r_2 \times r_3$	$r_1 \leftarrow r_1 - r_2 \times r_3$	...
$r_2 \leftarrow r_2 + [r = 1]$	$r_2 \leftarrow r_2 + [r = 2]$	$r_2 \leftarrow r_2 + [r = 3]$	...
$r_1 \leftarrow r_1 + r_2 \times r_3$	$r_1 \leftarrow r_1 + r_2 \times r_3$	$r_1 \leftarrow r_1 + r_2 \times r_3$	...
$r_2 \leftarrow r_2 - [l = 1]$	$r_2 \leftarrow r_2 - [l = 1]$	$r_2 \leftarrow r_2 - [l = 1]$	...
$r_1 \leftarrow r_1 - r_2 \times r_3$	$r_1 \leftarrow r_1 - r_2 \times r_3$	$r_1 \leftarrow r_1 - r_2 \times r_3$	...
$r_2 \leftarrow r_2 - [r = 1]$	$r_2 \leftarrow r_2 - [r = 2]$	$r_2 \leftarrow r_2 - [r = 3]$	...
$r_1 \leftarrow r_1 + r_2 \times r_3$	$r_1 \leftarrow r_1 + r_2 \times r_3$	$r_1 \leftarrow r_1 + r_2 \times r_3$	...

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

This means that instead of four  $k$  squared recursive calls, we only need to make four! The catch is that instead of three registers, we need three  $k$ , since each recursive call needs to return  $k$  different indicator values.

We can think of the output of the subroutine as a  $k$ -bit string, where exactly one of the bits is one and the others are zero. We call this a *one-hot encoding*.

So, how efficient is this strategy?



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

2021-10-26

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

Remember that the Pebbling algorithm uses on the order of  $k$  plus one to the  $h$  states.

- ▶ 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  $\Theta(2^{3k}4^hk^2)$  states.

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

This new parallel algorithm uses three  $k$  registers, so each layer of the branching program will have two to the three  $k$  states. It calls itself recursively four times, which means the number of layers is on the order of four to the  $h$  times  $k$  squared extra work that needs to be done. In total, we have on the order of two to the three  $k$  times four to the  $h$  times  $k$  squared states.

- ▶ 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  $\Theta(2^{3k}4^hk^2)$  states.
  - ▶ Beats pebbling when  $h \gg \frac{k}{\log k}$ .

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

When  $k$  is large compared to  $h$ , this is much worse than the pebbling algorithm. But when  $h$  is asymptotically larger than around  $k$  over  $\log k$ , this algorithm is an improvement.

The three times  $k$  registers are really hurting us, so the next thing we tried was a binary encoding: instead of  $k$  bits, use  $\log k$  bits to represent the value at the node in binary.

- ▶ 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  $\Theta(2^{3k}4^hk^2)$  states.
  - ▶ Beats pebbling when  $h \gg \frac{k}{\log k}$ .
- ▶ “Binary encoding” algorithm:
  - ▶ Recursively compute  $\log k$  bit vector representing node value.
  - ▶  $3 \log k$  registers.

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

The benefit here is that we only need three times  $\log k$  registers, instead of three  $k$ .



- ▶ 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  $\Theta(2^{3k}4^hk^2)$  states.
  - ▶ Beats pebbling when  $h \gg \frac{k}{\log k}$ .
- ▶ “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.)

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

The trouble is that we end up needing to multiply more than two values at once — our formula has degree two  $\log k$ . We found that we needed to make  $k$  squared recursive calls in order to multiply two  $\log k$  values, resulting in a total of  $k$  to the two h plus order one states.

This is strictly worse than the pebbling algorithm, but it's still a useful stepping stone.

- ▶ 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  $\Theta(2^{3k}4^hk^2)$  states.
  - ▶ Beats pebbling when  $h \gg \frac{k}{\log k}$ .
- ▶ “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  $h \geq k^{1/2+\epsilon'}$ .

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

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

By interpolating between the two encodings, you get an algorithm that does asymptotically better than pebbling as long as the height of the tree is at least  $k$  to the power one half plus any small constant.

## 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  $P \neq L$ .

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

## 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  $TEP \not\leq L$ , implying  $P \neq L$ .

In conclusion, we presented a new algorithm for the tree evaluation problem, which is the first improvement since TEP was introduced ten years ago.

It is not a log space algorithm, so TEP remains a possible approach for separating P from 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  $\text{TEP} \notin \text{L}$ , implying  $\text{P} \neq \text{L}$ .

## 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.)

# Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
- └ Solving TEP

## 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.)

There are two basic directions for future work.

The first is to improve the algorithm. The main limiting factor seems to be computing products. The “binary encoding” algorithm didn’t work because the number of recursive calls we have to make at each level is exponential in the degree of the polynomial we’re computing. It would be nice to be able to improve that. We’re not the first to point out this direction.

The other direction is to go back to proving lower bounds for the tree evaluation problem. If you remember, I briefly mentioned that we have lower bounds for two restricted classes of algorithm. The first is read-once algorithms, which are never allowed to read the same part of the input twice. The second is thrifty algorithms, which never read an irrelevant piece of the input. Our new algorithms violate both of those restrictions: we read every single part of the input, whether it’s relevant or not, and we do it over and over again, using repeated computation to save memory.



Thanks!

2021-10-26

## Catalytic approaches to the Tree Evaluation Problem

- └ New algorithm
  - └ Solving TEP
    - └ Thanks!

Thanks!

Thanks for watching, and I hope to see you at the first fully online STOC!