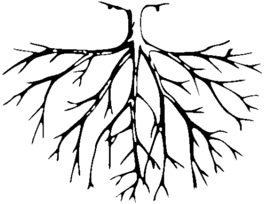
Recent Surprises on Tree Evaluation Problem (TEP)



Catalytic approaches to the tree evaluation problem - James Cook, Ian Mertz, 2020.
Encodings and the Tree Evaluation Problem - James Cook, Ian Mertz, 2021
Tree Evaluation is in Space $O(\log n . \log\log n)$ - James Cook, Ian Mertz, 2024

Speaker: Jayalal Sarma (IITM)

Pic courtsey: Internet

Talk inspired by Ian Mertz's talk on the topic

Space vs Time - a fundamental frontier

$\mathsf{SPACE}(\log t) \subseteq \mathsf{TIME}(t) \subseteq \mathsf{SPACE}(t)$

Space vs Time - a fundamental frontier

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

- Is either containment strict?

Space vs Time - a fundamental frontier

$$\mathrm{SPACE}(\log t) \subseteq \mathrm{TIME}(t) \subseteq \mathrm{SPACE}\left(\frac{t}{\log t}\right)$$

- Is either containment strict?
- L vs P problem.

Space vs Time - a fundamental frontier

$$\mathrm{SPACE}(\log t) \subseteq \mathrm{TIME}(t) \subseteq \mathrm{SPACE}\left(\frac{t}{\log t}\right)$$

- Is either containment strict?
- L vs P problem.
- Natural candidate : Circuit Value Problem (CVP).

Space vs Time - a fundamental frontier

$$\mathrm{SPACE}(\log t) \subseteq \mathrm{TIME}(t) \subseteq \mathrm{SPACE}\left(\frac{t}{\log t}\right)$$

- Is either containment strict?
- L vs P problem.
- Natural candidate : Circuit Value Problem (CVP).
- Conjecture is that CVP is not in L.

PSPACE
|
NP
|
P
|
NC^2
|
NL
|
$\mathrm{L} = \mathrm{SL}$
|
$\mathrm{NC}^1 = W5BP$
|
$\mathrm{ACC}^0[6]$

Space vs Time - a fundamental frontier

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

- Is either containment strict?
- L vs P problem.
- Natural candidate : Circuit Value Problem (CVP).
- Conjecture is that CVP is not in L.

PSPACE
|
NP
|
P
|
NC^2
|
NL
|
$\mathsf{L} = \mathsf{SL}$
|
$\mathsf{NC}^1 = W5BP$
|
$\mathsf{ACC}^0[6]$

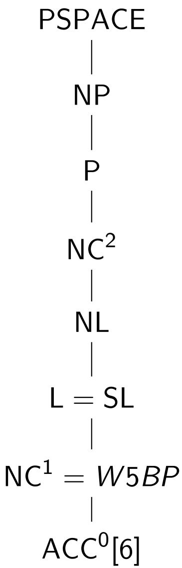
Space vs Time - a fundamental frontier

I BELIEVE P=NP

The only things that matter in a theoretical study are those that you can prove, but it's always fun to speculate. After worrying about P vs. NP for half my life, and having carefully reviewed the available "evidence" I have decided I believe that P = NP.

A main justification for my belief is history:

1. In the 1950's Kolmogorov conjectured that multiplication of n-bit integers requires time $\Omega(n^2)$. That's the time it takes to multiply using the method that mankind has used for at least six millennia. Presumably, if a better method existed it would have been found already. Kolmogorov subsequently started a seminar where he presented again this conjecture. Within one week of the start of the seminar, Karatsuba discovered his famous algorithm running in time $O(n^{\log_2 3}) \approx n^{1.58}$. He told Kolmogorov about it, who became agitated and terminated the seminar. Karatsuba's algorithm unleashed a new age of fast algorithms, including the next one. I recommend Karatsuba's own account [9] of this compelling story.
2. In 1968 Strassen started working on proving that the standard $O(n^3)$ algorithm for multiplying two $n \times n$ matrices is optimal. Next year his landmark $O(n^{\log_2 7}) \approx n^{2.81}$ algorithm appeared in his paper "Gaussian elimination is not optimal" [12].
3. In the 1970s Valiant showed that the graphs of circuits computing certain linear transformations must be a *super-concentrator*, a graph which certain strong connectivity properties. He conjectured that super-concentrators must have a super-linear number of wires, from which super-linear circuit lower bounds follow [13]. However, he later disproved the conjectured [14]: building on a result of Pinsker he constructed super-concentrators using a linear number of edges.
4. At the same time Valiant also defined *rigid* matrices and showed that an explicit construction of such matrices yields new circuit lower bounds. A specific matrix that was conjectured to be sufficiently rigid is the Hadamard matrix. Alman and Williams recently showed that, in fact, the Hadamard matrix is not rigid [1].
5. After finite automata, a natural step in lower bounds was to study sightly more general programs with constant memory. Consider a program that only maintains $O(1)$ bits of memory, and reads the input bits in a fixed order, where bits may be read several times. It seems quite obvious that such a program could not compute the majority function in polynomial time. This was explicitly conjectured by several people, including [5]. Barrington [4] famously disproved the conjecture by showing that in fact those seemingly very restricted constant-memory programs are in fact equivalent to log-depth circuits, which can compute majority (and many other things).
6. [Added 2/18] Mansour, Nisan, and Tiwari conjectured [10] in 1990 that computing hash functions on n bits requires circuit size $\Omega(n \log n)$. Their conjecture was disproved in 2008 [8] where a circuit of size $O(n)$ was given.



Space vs Time - a fundamental frontier

5. After finite automata, a natural step in lower bounds was to study sightly more general programs with constant memory. Consider a program that only maintains $O(1)$ bits of memory, and reads the input bits in a fixed order, where bits may be read several times. It seems quite obvious that such a program could not compute the majority function in polynomial time. This was explicitly conjectured by several people, including [5]. Barrington [4] famously disproved the conjecture by showing that in fact those seemingly very restricted constant-memory programs are in fact equivalent to log-depth circuits, which can compute majority (and many other things).

Surprise: Barrington's Theorem

continues to inspire further surprises

PSPACE

|

NP

|

P

|

NC^2

|

NL

|

$\mathsf{L} = \mathsf{SL}$

|

$\mathsf{NC}^1 = W5BP$

|

$\mathsf{ACC}^0[6]$

The Tree Evaluation Problem - $\text{TEP}_{d,k,h}$ - **[CMWBS 12]**

Fix: the complete d-ary tree, of height h, alphabet $[k]$.

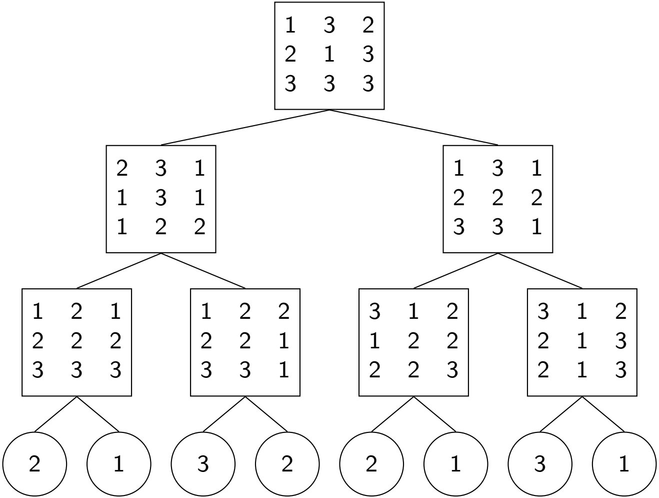
Input:

- Values from $[k]$ at the leaves.
- Tables of size $[k]^d$ with entries from $[k]$ at each internal node.

Task: Compute the value at the root.

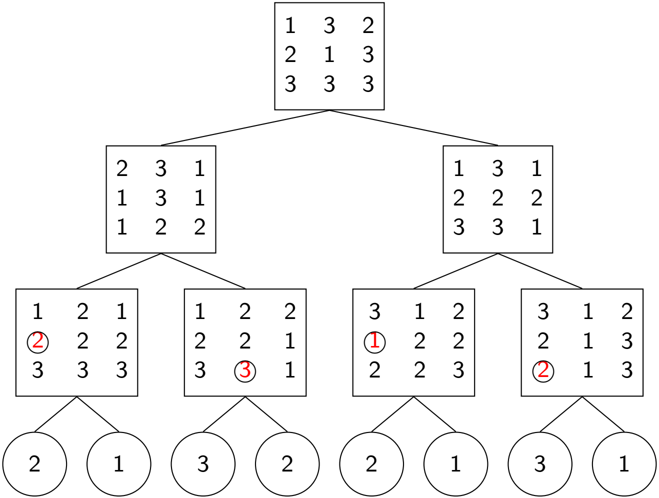
The Tree Evaluation Problem - $\mathrm{TEP}_{2,k,h}$ - **[CMWBS 12]**

$d = 2,\ k = 3,\ h = 4.$



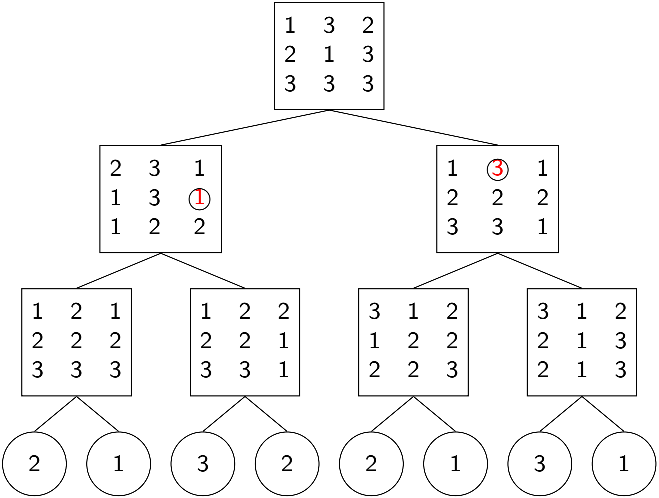
The Tree Evaluation Problem - $\mathrm{TEP}_{2,k,h}$ - **[CMWBS 12]**

$d = 2,\ k = 3,\ h = 4.$



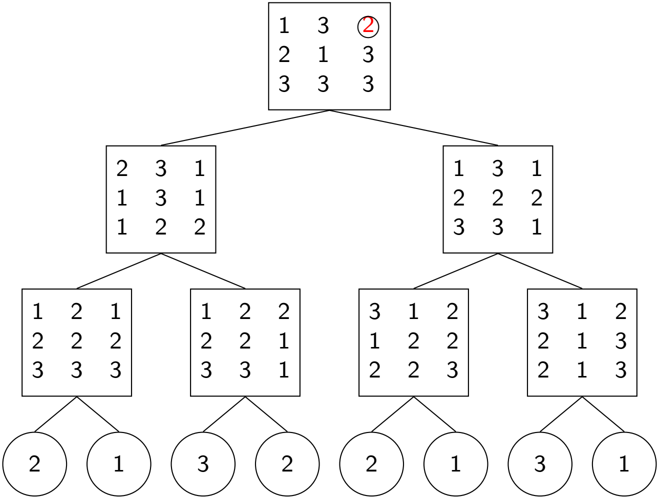
The Tree Evaluation Problem - $\mathrm{TEP}_{2,k,h}$ - **[CMWBS 12]**

$d = 2,\ k = 3,\ h = 4.$



The Tree Evaluation Problem - $\mathrm{TEP}_{2,k,h}$ - **[CMWBS 12]**

$d = 2,\ k = 3,\ h = 4.$



The Tree Evaluation Problem - $\text{TEP}_{d,k,h}$ - **[CMWBS 12]**

Fix: the complete d-ary tree, of height h (number of nodes), alphabet $[k]$.
Input:

- Values from $[k]$ at the leaves.
- Tables of size $[k]^d$ with entries from $[k]$ at each internal node.

Task: Compute the value at the root.

The Tree Evaluation Problem - $\mathrm{TEP}_{d,k,h}$ - **[CMWBS 12]**

Fix: the complete d-ary tree, of height h (number of nodes), alphabet $[k]$.
Input:

- Values from $[k]$ at the leaves.
- Tables of size $[k]^d$ with entries from $[k]$ at each internal node.

Task: Compute the value at the root.

Input Size (in bits) $d^{h-1} \log k + \left(\frac{d^{h-1}-1}{d-1}\right) k^d \log k$

The Tree Evaluation Problem - $\mathrm{TEP}_{d,k,h}$ - **[CMWBS 12]**

Fix: the complete d-ary tree, of height h (number of nodes), alphabet $[k]$.
Input:

- Values from $[k]$ at the leaves.
- Tables of size $[k]^d$ with entries from $[k]$ at each internal node.

Task: Compute the value at the root.

Input Size (in bits) $\quad d^{h-1}\log k + \left(\frac{d^{h-1}-1}{d-1}\right) k^d \log k$

With $d = 2$,

$$2^{h-1}\log k + (2^{h-1}-1)k^2\log k = 2^h \mathrm{poly}(k)$$

The Tree Evaluation Problem - $\text{TEP}_{d,k,h}$ - **[CMWBS 12]**

Fix: the complete d-ary tree, of height h (number of nodes), alphabet $[k]$.
Input:

- Values from $[k]$ at the leaves.
- Tables of size $[k]^d$ with entries from $[k]$ at each internal node.

Task: Compute the value at the root.

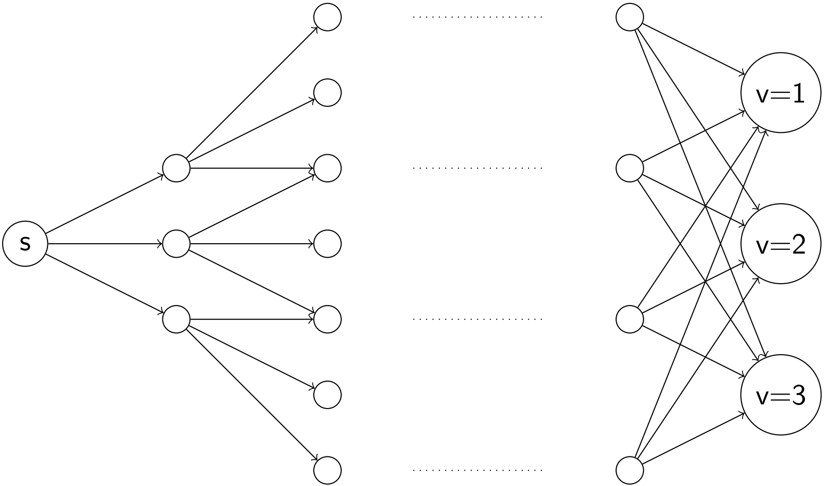
Input Size (in bits) $\quad d^{h-1} \log k + \left(\frac{d^{h-1}-1}{d-1}\right) k^d \log k$

With $d = 2$,

$$2^{h-1} \log k + (2^{h-1} - 1)k^2 \log k = 2^h \text{poly}(k)$$

Natural Algorithms: Bottom-up evaluation, Recursive evaluation.

Model : k-ary Deterministic Branching Programs



Trivial upper bound : 2^{h-1} length and k^h width

Complexity Bounds

- By the trivial evaluation algorithm :
  - $\mathsf{TEP}_{2,k,h} \in \mathsf{P}$ (even in NC^2).
  - $\mathsf{TEP}_{2,k,h}$ can be solved in space 2^h.

Complexity Bounds

- By the trivial evaluation algorithm :
  - $\mathsf{TEP}_{2,k,h} \in \mathsf{P}$ (even in NC^2).
  - $\mathsf{TEP}_{2,k,h}$ can be solved in space 2^h.
- Recursive (with reuse of space) : $\mathsf{TEP}_{2,k,h}$ is in space $O(h \log k)$.

Complexity Bounds

- By the trivial evaluation algorithm :
  - $\mathrm{TEP}_{2,k,h} \in \mathrm{P}$ (even in NC^2).
  - $\mathrm{TEP}_{2,k,h}$ can be solved in space 2^h.
- Recursive (with reuse of space) : $\mathrm{TEP}_{2,k,h}$ is in space $O(h \log k)$.

Conjecture [CMWBS **2012]** : $\mathrm{TEP}_{2,k,h} \notin \mathrm{L}$.

That is, $\mathrm{TEP}_{2,k,h}$ cannot be solved in $O(h + \log k)$ space.

Complexity Bounds

- By the trivial evaluation algorithm :
  - $\mathrm{TEP}_{2,k,h} \in \mathrm{P}$ (even in NC^2).
  - $\mathrm{TEP}_{2,k,h}$ can be solved in space 2^h.
- Recursive (with reuse of space) : $\mathrm{TEP}_{2,k,h}$ is in space $O(h \log k)$.

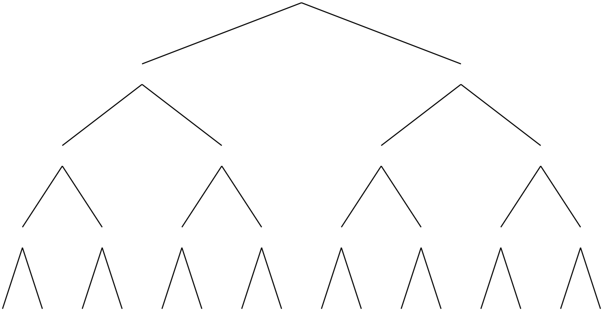
Conjecture [CMWBS **2012]** : $\mathrm{TEP}_{2,k,h} \notin \mathrm{L}$.

That is, $\mathrm{TEP}_{2,k,h}$ cannot be solved in $O(h + \log k)$ space.

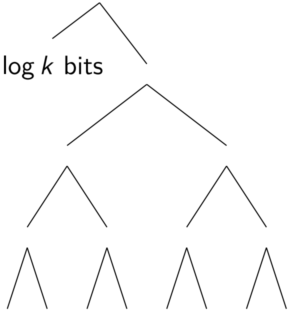
Conjecture [KRW **1995]** : $\mathrm{TEP}_{d,2,h} \notin \mathrm{NC}^1$.

That is, depth($\mathrm{TEP}_{d,2,h}$) is at least $\Omega(h \log d)$.

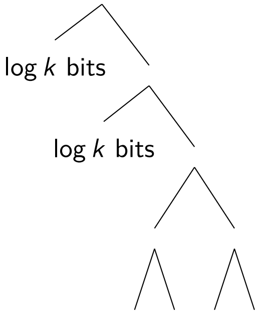
Why do/did we believe $\mathrm{TEP}_{2,k,h} \notin \mathrm{L}$?



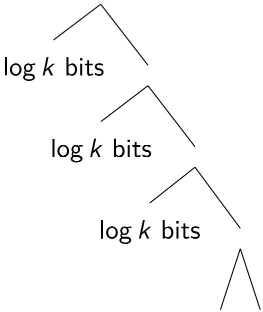
Why do/did we believe $\mathsf{TEP}_{2,k,h} \notin \mathsf{L}$?



Why do/did we believe $\mathsf{TEP}_{2,k,h} \notin \mathsf{L}$?



Why do/did we believe $\mathrm{TEP}_{2,k,h} \notin \mathrm{L}$?

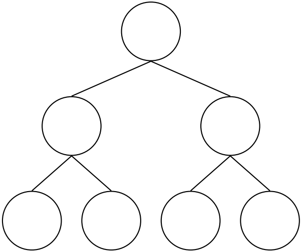


- Memory adds up to $\Omega(h \log k)$ space.
- Recall : input size $2^h \mathrm{poly}(k)$.
- For $\mathrm{TEP}_{2,k,h} \in \mathrm{L}$, this should be doable in $O(h + \log k)$ space.

Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

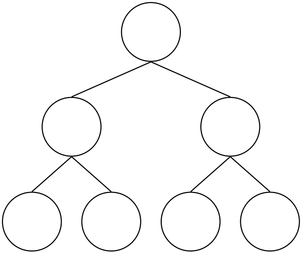
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

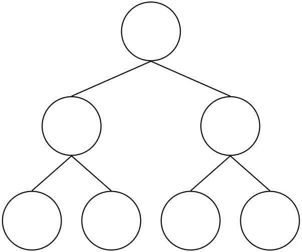
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

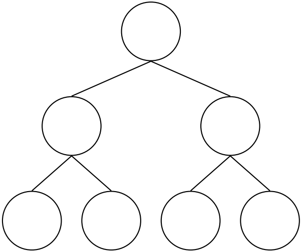
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

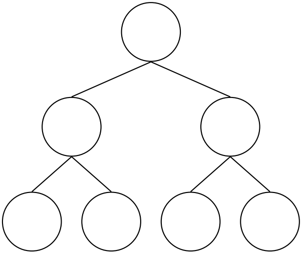
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

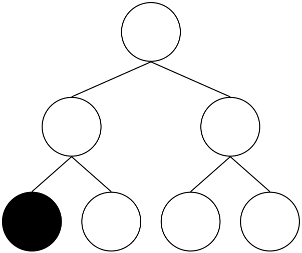
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

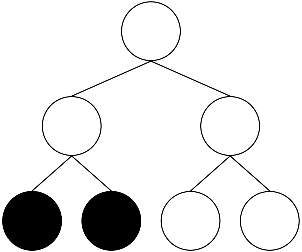
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

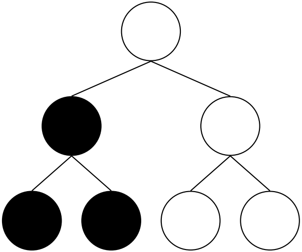
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

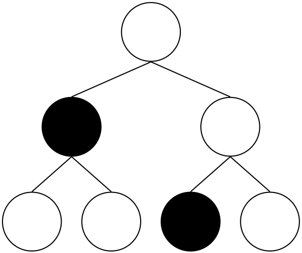
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

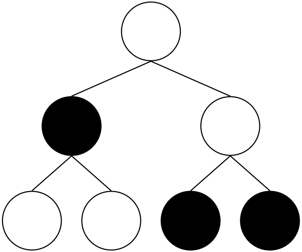
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

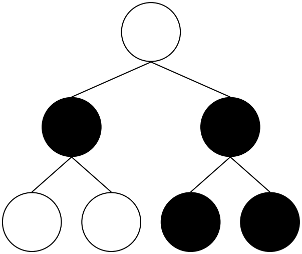
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

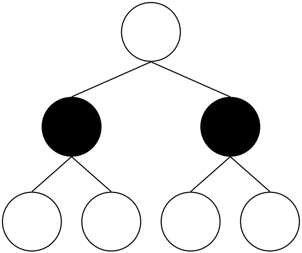
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

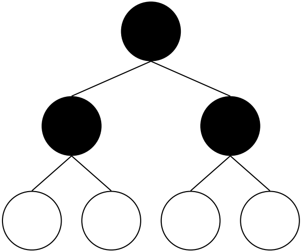
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

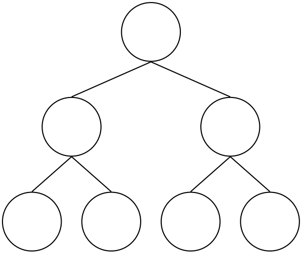
- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.

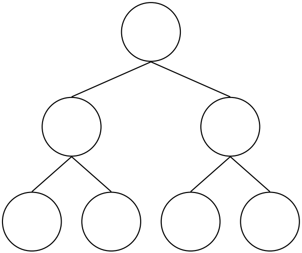


[**Pebbling Bound**] Pebbling of T_2^h requires $\Omega(h)$ pebbles.

Pebbling Algorithms and Tree Evaluation

Graph Pebbling: [Paterson and Hewit 1970]

- Pebble can be placed on a leaf any time.
- Pebble can be removed from any node at any time.
- To pebble a node, all its children should be pebbled.
- Minimise the number of pebbles used at any point of time.



[**Pebbling Bound**] Pebbling of T_2^h requires $\Omega(h)$ pebbles.

[**Lower Bound Strategy**] From the algorithm that uses space s, extract a pebbling strategy of T_2^h with number of pebbles function of s.

$\mathsf{TEP}_{2,k,h}$ cannot be in $o(h \log k)$ space if we assume ...

$\mathsf{TEP}_{2,k,h}$ cannot be in $o(h \log k)$ space if we assume ...

Read-once Branching Programs (ROBP): In any computation path, the branching program queries values at a node only once.

$\mathsf{TEP}_{2,k,h}$ cannot be in $o(h \log k)$ space if we assume ...

Read-once Branching Programs (ROBP): In any computation path, the branching program queries values at a node only once.

Thrifty Branching Programs: For each internal node, the algorithm must query the table only on the correct pair of values.

$\mathsf{TEP}_{2,k,h}$ cannot be in $o(h \log k)$ space if we assume ...

Read-once Branching Programs (ROBP): In any computation path, the branching program queries values at a node only once.

Thrifty Branching Programs: For each internal node, the algorithm must query the table only on the correct pair of values.

[**CMWBS 2009**]
$\mathsf{TEP}_{2,k,h}$ requires $\Omega(k^h)$ size for Thrifty or RO BPs.

$\mathsf{TEP}_{2,k,h}$ cannot be in $o(h \log k)$ space if we assume ...

Read-once Branching Programs (ROBP): In any computation path, the branching program queries values at a node only once.

Thrifty Branching Programs: For each internal node, the algorithm must query the table only on the correct pair of values.

[**CMWBS 2009**]
$\mathsf{TEP}_{2,k,h}$ requires $\Omega(k^h)$ size for Thrifty or RO BPs.

[**Thrifty Hypothesis**]
For $\mathsf{TEP}_{2,k,h}$, any Branching Program can be made thrifty without increasing the size beyond poly factors.

$\mathsf{TEP}_{2,k,h}$ cannot be in $o(h \log k)$ space if we assume ...

Read-once Branching Programs (ROBP): In any computation path, the branching program queries values at a node only once.

Thrifty Branching Programs: For each internal node, the algorithm must query the table only on the correct pair of values.

[**CMWBS 2009**]
$\mathsf{TEP}_{2,k,h}$ requires $\Omega(k^h)$ size for Thrifty or RO BPs.

[**Thrifty Hypothesis**]
For $\mathsf{TEP}_{2,k,h}$, any Branching Program can be made thrifty without increasing the size beyond poly factors.

Many of these results extend to non-deterministic setting as well.

Conjecture : $\mathsf{TEP}_{2,n,k} \notin \mathsf{L}$

Conjecture : $\mathsf{TEP}_{2,n,k} \notin \mathsf{L}$

[**Stephen Cook's 100 USD TEP Challenge**]

Design an algorithm for $\mathsf{TEP}_{2,n,k}$ that uses $o(h \log k)$ space.

Design a branching program for $\mathsf{TEP}_{2,n,k}$ of size $k^{h-\epsilon}$ for a const ϵ.

Conjecture : $\mathsf{TEP}_{2,n,k} \notin \mathsf{L}$

[**Stephen Cook's 100 USD TEP Challenge**]

Design an algorithm for $\mathsf{TEP}_{2,n,k}$ that uses $o(h \log k)$ space.

Design a branching program for $\mathsf{TEP}_{2,n,k}$ of size $k^{h-\epsilon}$ for a const ϵ.

Rest of this talk: how James Cook & Ian Mertz won that 100 USD.

An Earlier Surprise in Complexity Theory

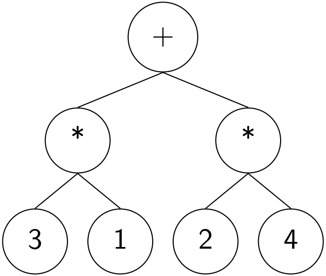
Barrington's Theorem (1989):
Any $f \in \mathrm{NC}^1$ can be computed by width 5 branching programs of polynomial length.

$$\mathrm{P}$$
$$|$$
$$\mathrm{L}$$
$$|$$
$$\mathrm{NC}^1 = \mathrm{W5}BP$$
$$|$$
$$\mathrm{ACC}^0[6]$$

An Earlier Surprise in Complexity Theory

Barrington's Theorem (1989):
Any $f \in \mathsf{NC}^1$ can be computed by width 5 branching programs of polynomial length.

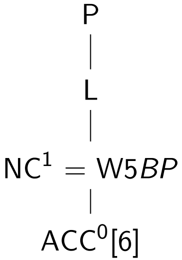
P
|
L
|
$\mathsf{NC}^1 = \mathsf{W5}BP$
|
$\mathsf{ACC}^0[6]$

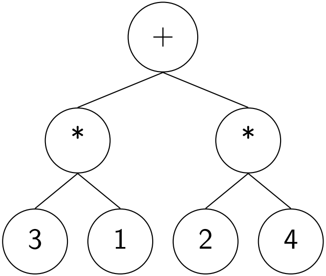


$$\#\mathsf{NC}^1 = \left\{ f : \begin{array}{l} f \text{ computed by log-depth} \\ \text{fan-in 2, poly size} \\ \text{circuits with } +/* \text{ gates} \\ \text{over } \mathbb{N} \end{array} \right\}$$

An Earlier Surprise in Complexity Theory

Barrington's Theorem (1989):
Any $f \in \mathsf{NC}^1$ can be computed by width 5 branching programs of polynomial length.

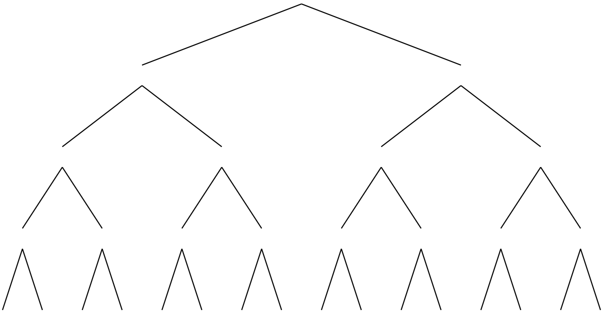




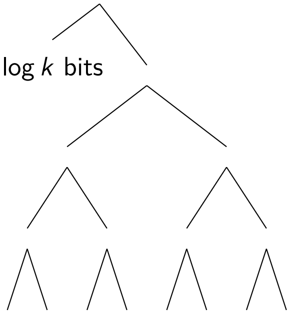
$$\#\mathsf{NC}^1 = \left\{ f : \begin{array}{l} f \text{ computed by log-depth} \\ \text{fan-in 2, poly size} \\ \text{circuits with } +/* \text{ gates} \\ \text{over } \mathbb{N} \end{array} \right\}$$

Ben-Or and Cleve (1992):
Any $f \in \#\mathsf{NC}^1$ can be computed in $O(\log n)$ space.

Ben-Or and Cleve solves $\mathrm{TEP}_{2,k,h}$ special case !!

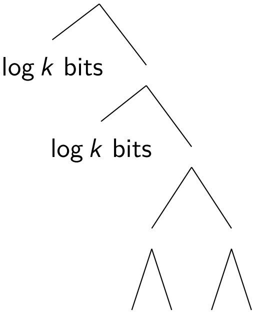


Ben-Or and Cleve solves $\text{TEP}_{2,k,h}$ special case !!



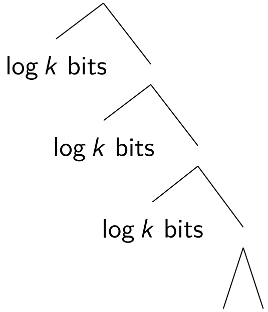
Why did this logic not apply?

Ben-Or and Cleve solves $\text{TEP}_{2,k,h}$ special case !!



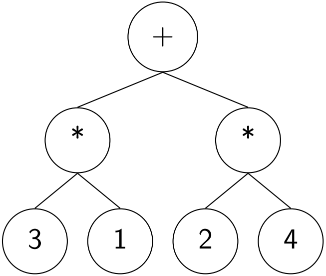
Why did this logic not apply?

Ben-Or and Cleve solves $TEP_{2,k,h}$ special case !!

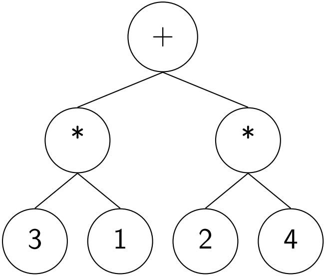


Why did this logic not apply?

Let us learn from Ben-Or and Cleve

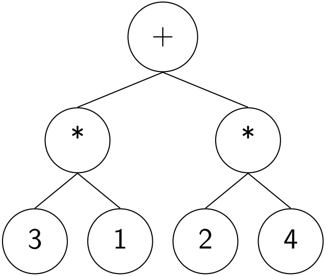


Let us learn from Ben-Or and Cleve



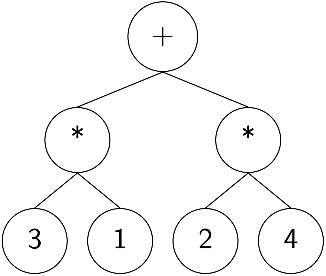
- Use register programs (1975): Three registers - R_0, R_1, and R_2 each holding a value in $[k]$. Total $3 \log k$ space.

Let us learn from Ben-Or and Cleve



- Use register programs (1975): Three registers - R_0, R_1, and R_2 each holding a value in $[k]$. Total $3 \log k$ space.
- Registers are updated by "invertible" instructions of the form $R_0 \leftarrow R_0 + R_1 R_2$.

Let us learn from Ben-Or and Cleve

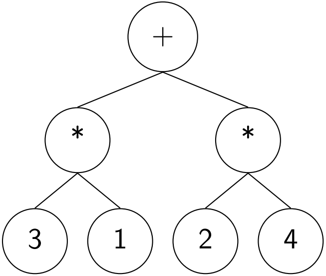


- Use register programs (1975): Three registers - R_0, R_1, and R_2 each holding a value in $[k]$. Total $3 \log k$ space.
- Registers are updated by "invertible" instructions of the form $R_0 \leftarrow R_0 + R_1 R_2$.
- Program computing $f(x)$ must transform:

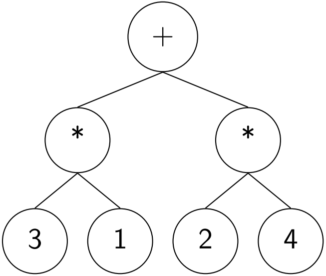
$$\left\{ \begin{array}{c} R_0 = \tau_1 \\ R_1 = \tau_2 \\ R_2 = \tau_3 \end{array} \right\} \rightarrow \left\{ \begin{array}{c} R_0 = \tau_1 + f(x) \\ R_1 = \tau_2 \\ R_2 = \tau_3 \end{array} \right\}$$

We will call this as "clean computation"

A Quick Proof of Ben-Or and Cleve

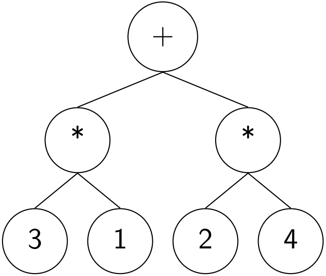


A Quick Proof of Ben-Or and Cleve



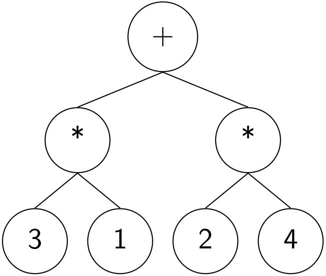
- Given a algebraic formula, we construct the program inductively.

A Quick Proof of Ben-Or and Cleve



- Given a algebraic formula, we construct the program inductively.
- Base case for a node x_i : $R_0 \leftarrow R_0 + x_i$.

A Quick Proof of Ben-Or and Cleve



- Given a algebraic formula, we construct the program inductively.
- Base case for a node x_i : $R_0 \leftarrow R_0 + x_i$.
- If $g = h_1 + h_2$, and inductively, $h_1(x)$ and $h_2(x)$ can be computed into R_1 and R_2 respectively. Run those programs and then the instructions $R_0 \leftarrow R_0 + R_1$ followed by $R_0 \leftarrow R_0 + R_2$ is enough.

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $R_0 = \tau_0$, $R_1 = \tau_1$, $R_2 = \tau_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- $P_1,\ P_2$ $R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- $P_1,\ P_2$ $\quad R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\quad R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\qquad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- $P_1,\ P_2$ $\qquad R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\qquad R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$
- P_1^{-1} $\qquad R_1 = \tau_1$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- $P_1,\ P_2$ $\quad R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\quad R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$
- P_1^{-1} $\quad R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 + v_1\tau_2 + v_1 v_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization — $R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- $P_1,\ P_2$ — $R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ — $R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$
- P_1^{-1} — $R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ — $R_0 = \tau_0 + v_1\tau_2 + v_1 v_2$
- P_1 — $R_1 = \tau_1 + v_1$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization — $R_0 = \tau_0$, $R_1 = \tau_1$, $R_2 = \tau_2$
- P_1, P_2 — $R_1 = \tau_1 + v_1$, $R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ — $R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$
- P_1^{-1} — $R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ — $R_0 = \tau_0 + v_1\tau_2 + v_1 v_2$
- P_1 — $R_1 = \tau_1 + v_1$
- P_2^{-1} — $R_2 = \tau_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1, P_2 $\quad R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1R_2$ $\quad R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$
- P_1^{-1} $\quad R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1R_2$ $\quad R_0 = \tau_0 + v_1\tau_2 + v_1 v_2$
- P_1 $\quad R_1 = \tau_1 + v_1$
- P_2^{-1} $\quad R_2 = \tau_2$
- $R_0 \leftarrow R_0 - R_1R_2$ $\quad R_0 = \tau_0 - \tau_1\tau_2 + v_1 v_2.$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- $P_1,\ P_2$ $\quad R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\quad R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$
- P_1^{-1} $\quad R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 + v_1\tau_2 + v_1 v_2$
- P_1 $\quad R_1 = \tau_1 + v_1$
- P_2^{-1} $\quad R_2 = \tau_2$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 - \tau_1\tau_2 + v_1 v_2.$
- P_1^{-1} $\quad R_1 = \tau_1$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- $P_1,\ P_2$ $\quad R_1 = \tau_1 + v_1,\ R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\quad R_0 = \tau_0 + \tau_1\tau_2 + \tau_1 v_2 + v_1\tau_2 + v_1 v_2$
- P_1^{-1} $\quad R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 + v_1\tau_2 + v_1 v_2$
- P_1 $\quad R_1 = \tau_1 + v_1$
- P_2^{-1} $\quad R_2 = \tau_2$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 - \tau_1\tau_2 + v_1 v_2.$
- P_1^{-1} $\quad R_1 = \tau_1$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\quad R_0 = \tau_0 + v_1 v_2.$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1 $R_1 = \tau_1 + v_1$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\qquad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1 $\qquad R_1 = \tau_1 + v_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\qquad R_0 = \tau_0 - \tau_1\tau_2 - v_1\tau_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1 $\quad R_1 = \tau_1 + v_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 - \tau_1\tau_2 - v_1\tau_2$
- P_2 $\quad R_2 = \tau_2 + v_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization — $R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1 — $R_1 = \tau_1 + v_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ — $R_0 = \tau_0 - \tau_1\tau_2 - v_1\tau_2$
- P_2 — $R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ — $R_0 = \tau_0 + \tau_1 v_2 - v_1 v_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1 $\quad R_1 = \tau_1 + v_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 - \tau_1\tau_2 - v_1\tau_2$
- P_2 $\quad R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\quad R_0 = \tau_0 + \tau_1 v_2 - v_1 v_2$
- P_1^{-1} $\quad R_1 = \tau_1$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\qquad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1 $\qquad R_1 = \tau_1 + v_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\qquad R_0 = \tau_0 - \tau_1\tau_2 - v_1\tau_2$
- P_2 $\qquad R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\qquad R_0 = \tau_0 + \tau_1 v_2 - v_1 v_2$
- P_1^{-1} $\qquad R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\qquad R_0 = \tau_0 - \tau_1\tau_2 + v_1 v_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

- Initialization $\quad R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
- P_1 $\quad R_1 = \tau_1 + v_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 - \tau_1\tau_2 - v_1\tau_2$
- P_2 $\quad R_2 = \tau_2 + v_2$
- $R_0 \leftarrow R_0 + R_1 R_2$ $\quad R_0 = \tau_0 + \tau_1 v_2 - v_1 v_2$
- P_1^{-1} $\quad R_1 = \tau_1$
- $R_0 \leftarrow R_0 - R_1 R_2$ $\quad R_0 = \tau_0 - \tau_1\tau_2 + v_1 v_2$
- P_2^{-1} $\quad R_2 = \tau_2$

Ben-Or and Cleve Construction for Multiplication Gate

$g = h_1 \times h_2$ and inductively, $v_1 = h_1(x)$ and $v_2 = h_2(x)$ can be computed into R_1 and R_2 by programs P_1 and P_2 resp.

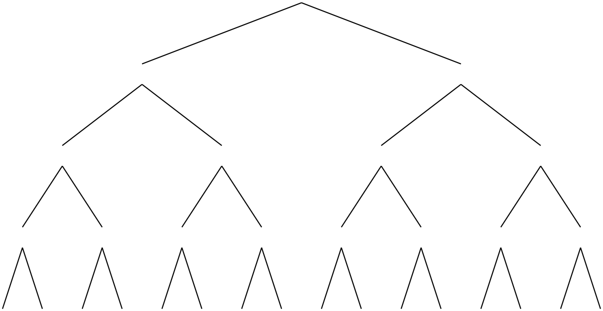
Initialization	$R_0 = \tau_0,\ R_1 = \tau_1,\ R_2 = \tau_2$
P_1	$R_1 = \tau_1 + v_1$
$R_0 \leftarrow R_0 - R_1 R_2$	$R_0 = \tau_0 - \tau_1\tau_2 - v_1\tau_2$
P_2	$R_2 = \tau_2 + v_2$
$R_0 \leftarrow R_0 + R_1 R_2$	$R_0 = \tau_0 + \tau_1 v_2 - v_1 v_2$
P_1^{-1}	$R_1 = \tau_1$
$R_0 \leftarrow R_0 - R_1 R_2$	$R_0 = \tau_0 - \tau_1\tau_2 + v_1 v_2$
P_2^{-1}	$R_2 = \tau_2$
$R_0 \leftarrow R_0 - R_1 R_2$	$R_0 = \tau_0 + v_1 v_2$

[**Register Programs for Multiplication Gates**]
For all nodes g, there is a program P_g which results in

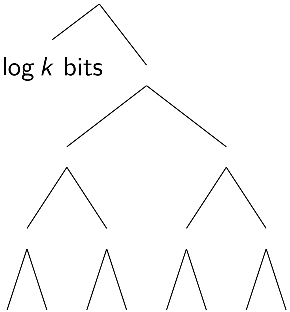
$$R_0 \leftarrow R_0 + v_g$$
$$R_i \leftarrow R_i \quad \forall i \neq 0$$

where $v_g \in \mathbb{N}$ is the value at the node $g \in T$ computed by the multiplication gate.
using 3 registers holding values from $\mathbb{N}$ and 6 recursive calls.

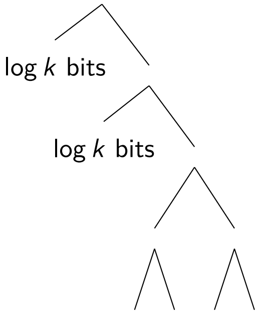
What did we learn from Ben-Or and Cleve?



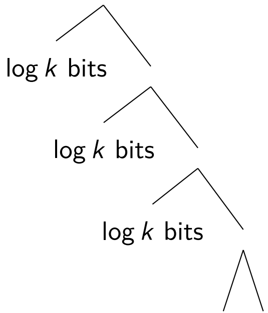
What did we learn from Ben-Or and Cleve?



What did we learn from Ben-Or and Cleve?



What did we learn from Ben-Or and Cleve?



Why did this logic not apply?

Storage + Computation

Storage+Computation : Catalytic Computation [BCKLS14]

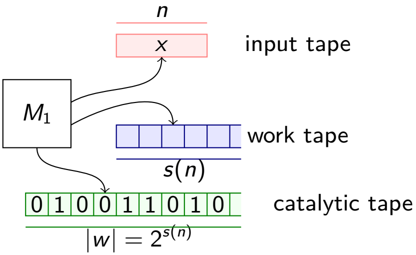
- The catalytic tape contains a string w.
- Restore the string w at the end of the computation.

Storage+Computation : Catalytic Computation [BCKLS14]

- The catalytic tape contains a string w.
- Restore the string w at the end of the computation.
- CL is the class where $s(n) = O(\log n)$

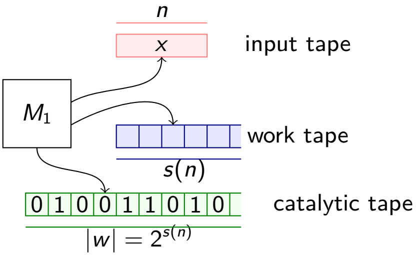
Storage+Computation : Catalytic Computation [BCKLS14]

- The catalytic tape contains a string w.
- Restore the string w at the end of the computation.
- CL is the class where $s(n) = O(\log n)$
- Question : Can anything more than L be accepted?



Storage+Computation : Catalytic Computation [BCKLS14]

- The catalytic tape contains a string w.
- Restore the string w at the end of the computation.
- CL is the class where $s(n) = O(\log n)$
- Question : Can anything more than L be accepted?



[BCKLS 2014]

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{TC}^1 \subseteq \mathsf{CL} \subseteq \mathsf{ZPP}$$

Picture credit: Bhabya

[**Register Programs for Majority Gates**]

For all nodes g, there is a program P_g which results in

$$R_0 \leftarrow R_0 + v_g$$
$$R_i \leftarrow R_i \quad \forall i \neq 0$$

where $v_g \in \{0, 1\}$ is the value at the node $g \in T$ computed by the MAJORITY gate.

using poly(n) registers holding values from $\{0, 1\}$ and $O(1)$ recursive calls.

[**Register Programs for Majority Gates**]

For all nodes g, there is a program P_g which results in

$$R_0 \leftarrow R_0 + v_g$$
$$R_i \leftarrow R_i \quad \forall i \neq 0$$

where $v_g \in \{0,1\}$ is the value at the node $g \in T$ computed by the MAJORITY gate.

using poly(n) registers holding values from $\{0,1\}$ and $O(1)$ recursive calls.

$$MAJ(x_1, x_2, \ldots x_n) = \sum_{k=\frac{n}{2}}^{n} \left[1 - \left(\sum_i x_i - k \right)^{p-1} \right] \mod p$$

[BCKLS 2014] : Designed programs for unbounded sum and powering.

How do we do TEP? (Cook and Mertz 2020)

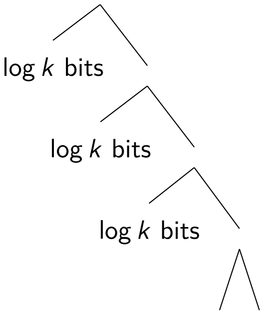
- Idea: storage + computation.

How do we do TEP? (Cook and Mertz 2020)

- Idea: storage + computation.
- Encode : the value in vector from.

A vector $\vec{v_p} \in \mathbb{F}_2^k$ stores $x \in [k]$ if

$$\vec{v}_{p,i} = \begin{cases} 1 & \text{if } i = x \\ 0 & \text{otherwise} \end{cases}$$



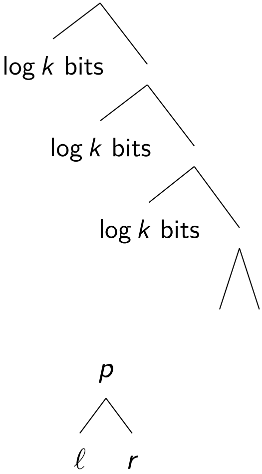
How do we do TEP? (Cook and Mertz 2020)

- Idea: storage + computation.
- Encode : the value in vector from.

A vector $\vec{v_p} \in \mathbb{F}_2^k$ stores $x \in [k]$ if

$$\vec{v}_{p,i} = \begin{cases} 1 & \text{if } i = x \\ 0 & \text{otherwise} \end{cases}$$

$$\vec{v}_{p,x} = \sum_{(y,z) \in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$



How do we do TEP? (Cook and Mertz 2020)

$$\vec{v}_{p,x} = \sum_{(y,z) \in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

How do we do TEP? (Cook and Mertz 2020)

$$\vec{v}_{p,x} = \sum_{(y,z) \in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

Similar to the instruction $R_0 \leftarrow R_0 - R_1 R_2$:

for x, y, z such that $f_p(y, z) = x$ do the following:

$$R_{p,x} \leftarrow R_{p,x} - R_{\ell,y} R_{r,z}$$

This will result in

$$R_{p,x} = \tau_{p,x} - \sum_{(y,z) \in f_p^{-1}(x)} \left(\tau_{\ell,y} \tau_{r,z} + v_{\ell,y} v_{r,z} \right)$$

How do we do TEP? (Cook and Mertz 2020)

$$\vec{v}_{p,x} = \sum_{(y,z)\in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

lifted from CM20 paper

P_ℓ

for $x; (y, z)$ such that $f_p(y, z) = x$ **do**

  $R_{p,x} \leftarrow R_{p,x} - R_{\ell,y} R_{r,z}$

    $\triangleright\ R_{p,x} = \tau_{p,x} - \sum_{(y,z)\in f_p^{-1}(x)} (\tau_{\ell,y}\tau_{r,z} + v_{\ell,y}\tau_{r,z})$

end for

P_r

for $x; (y, z)$ such that $f_p(y, z) = x$ **do**

  $R_{p,x} \leftarrow R_{p,x} + R_{\ell,y} R_{r,z}$

    $\triangleright\ R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)} (\tau_{\ell,y} v_{r,z} + v_{\ell,y} v_{r,z})$

end for

P_ℓ^{-1}

for $x; (y, z)$ such that $f_p(y, z) = x$ **do**

  $R_{p,x} \leftarrow R_{p,x} - R_{\ell,y} R_{r,z}$

    $\triangleright\ R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)} (-\tau_{\ell,y}\tau_{r,z} + v_{\ell,y} v_{r,z})$

end for

P_r^{-1}

for $x; (y, z)$ such that $f_p(y, z) = x$ **do**

  $R_{p,x} \leftarrow R_{p,x} + R_{\ell,y} R_{r,z}$

    $\triangleright\ R_{p,x} = \tau_{p,x} + \sum_{(y,z)\in f_p^{-1}(x)} v_{\ell,y} v_{r,z}$

end for

How do we do TEP? (Cook and Mertz 2020)

$$\vec{v}_{p,x} = \sum_{(y,z)\in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

- $3k$ registers, $4k^2$ instructions
- Two recursive calls to P_ℓ and P_r each.
- Register program : $4^h k^2$ length, $3k$ binary registers.
- Branching program with $4^h \text{poly}(k)$ length and 2^{3k} width.

How do we do TEP? (Cook and Mertz 2020)

$$\vec{v}_{p,x} = \sum_{(y,z)\in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

- $3k$ registers, $4k^2$ instructions
- Two recursive calls to P_ℓ and P_r each.
- Register program : $4^h k^2$ length, $3k$ binary registers.
- Branching program with $4^h \text{poly}(k)$ length and 2^{3k} width.
- This gives an algorithm that uses $O(h + k)$ space.

How do we do TEP? (Cook and Mertz 2020)

$$\vec{v}_{p,x} = \sum_{(y,z)\in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

- $3k$ registers, $4k^2$ instructions
- Two recursive calls to P_ℓ and P_r each.
- Register program : $4^h k^2$ length, $3k$ binary registers.
- Branching program with $4^h\text{poly}(k)$ length and 2^{3k} width.
- This gives an algorithm that uses $O(h+k)$ space.
- Better than $O(h \log k)$ when $h >> \frac{k}{\log k}$.

How do we do TEP? (Cook and Mertz 2020)

$$\vec{v}_{p,x} = \sum_{(y,z) \in f_p^{-1}(x)} [v_{\ell,y} = 1][v_{r,z} = 1]$$

- $3k$ registers, $4k^2$ instructions
- Two recursive calls to P_ℓ and P_r each.
- Register program : $4^h k^2$ length, $3k$ binary registers.
- Branching program with $4^h \text{poly}(k)$ length and 2^{3k} width.
- This gives an algorithm that uses $O(h + k)$ space.
- Better than $O(h \log k)$ when $h >> \frac{k}{\log k}$.
- Sanity Check: not thrifty, not read-once.

TEP (attempt to improve) (Cook and Mertz 2020)

TEP (attempt to improve) (Cook and Mertz 2020)

Idea : use binary encoding - v_p, v_ℓ, v_r are binary encodings of the value in $[k]$.

TEP (attempt to improve) (Cook and Mertz 2020)

Idea : use binary encoding - v_p, v_ℓ, v_r are binary encodings of the value in $[k]$.

$$\vec{v}_{p,b} = \sum_{(x,y,z)\in[k]^3} [x_b = 1][f(y,z) = x] \prod_{b'\in[\log k]} [v_{\ell,b'} = y'][v_{r,b'} = z_{b'}]$$

Instead of binary products, now we have $(2\log k)$-ary products !

Evaluating t-ary products

Given $\{P_i\}_{i \in [t]}$ programs to compute $\{v_i\}_{i \in [t]}$

We need to compute $R_0 = R_0 + \prod_{i=1}^{t} v_i$ "cleanly".

Evaluating t-ary products

Given $\{P_i\}_{i\in[t]}$ programs to compute $\{v_i\}_{i\in[t]}$

We need to compute $R_0 = R_0 + \prod_{i=1}^{t} v_i$ "cleanly".

- $t+1$ registers - $R_0, R_1 \ldots R_t$
- P_S is program that gets
  $\mathrm{R}_i = \tau_i + v_i$ for $i \notin S$
  $\mathrm{R}_i = \tau_i$ for $i \in S$

for each $S \subseteq [t]$

- Run P_S
- $R_0 \leftarrow \tau_0 + c_S \prod_{i=1}^{d} R_i$

$$R_0 \leftarrow \tau_0 + \sum_{S\subseteq[t]} c_S \left(\prod_{i\in S} \tau_i\right)\left(\prod_{i\notin S}(\tau_i + v_i)\right)$$

Choose c_S's such that this is $R_0 \leftarrow \tau_0 + \prod_{i=1}^{t} v_i$

Evaluating t-ary products

Given $\{P_i\}_{i\in[t]}$ programs to compute $\{v_i\}_{i\in[t]}$

We need to compute $R_0 = R_0 + \prod_{i=1}^{t} v_i$ "cleanly".

- $t+1$ registers - $R_0, R_1 \dots R_t$
- P_S is program that gets
  $\mathrm{R}_i = \tau_i + v_i$ for $i \notin S$
  $\mathrm{R}_i = \tau_i$ for $i \in S$

for each $S \subseteq [t]$

- Run P_S
- $R_0 \leftarrow \tau_0 + c_S \prod_{i=1}^{d} R_i$

$$R_0 \leftarrow \tau_0 + \sum_{S\subseteq[t]} c_S \left(\prod_{i\in S} \tau_i\right)\left(\prod_{i\notin S}(\tau_i + v_i)\right)$$

Choose c_S's such that this is $R_0 \leftarrow \tau_0 + \prod_{i=1}^{t} v_i$

2^t recursive calls $+$ 2^t additional instructions

Plugging in to $\mathrm{TEP}_{2,k,h}$

$$\vec{v}_{p,b} = \sum_{(x,y,z)\in[k]^3} [x_b = 1][f(y,z) = x] \prod_{b'\in[\log k]} [v_{\ell,b'} = y'][v_{r,b'} = z_{b'}]$$

- We need to evaluate $2\log k$-ary products.
- $3\log k$ registers, $O(k^3 \log k)$ additional instructions.
- $2k^2$ recursive calls.
- Register program : $(2k)^{2h}$ length, $3\log k$ binary registers.
- Branching program with $(2k)^{2h}\mathrm{poly}(k)$ length and $\mathrm{poly}(k)$ width.
- This gives an algorithm that uses $O(h\log k)$ space.
- NOT Better than $O(h\log k)$.

TEP (hybrid and improved) (Cook and Mertz 2020)

- $a \in \log k$
- Break $[k]$ into blocks of length $2^a - 1$.
- Encode each value as a pair - (A, B) - block number, and non-zero index into block.
- $\vec{v} \in \{0, 1\}^t$ where $t = a \times \frac{k}{2^a - 1}$.
- Program needs to compute $2a$-ary products.
- Choose $a = \log\left(\frac{ck}{h} + 1\right)$ for a constant c.

[**Cook and Mertz 2020**] For $h \geq k^{\frac{1}{2} + \frac{\epsilon}{4}}$, then TEP can be solved by branching programs of size much less than $k^{h-\epsilon}$

TEP (100 USD prize !) (Cook and Mertz 2021)

- Fix b, d such that $b^d \geq k$.
- Write v as d digits in base b, and then encode each digit using a characteristic vector encoding of length b.

[**Cook and Mertz 2021**] $\mathrm{TEP}_{2,k,h}$ can be computed by branching programs of size

$$k^{O\left(\frac{h}{\log h}\right)} + 2^{O(h)}$$

TEP in $O(\log n . \log\log n)$ space (Cook and Mertz 2024)

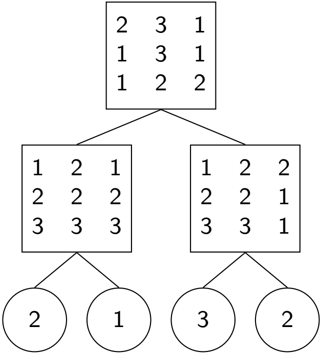
[**Cook and Mertz 2024**]
$TEP_{2,k,h}$ can be solved in space

$$O((h + \log k) \log\log k)$$

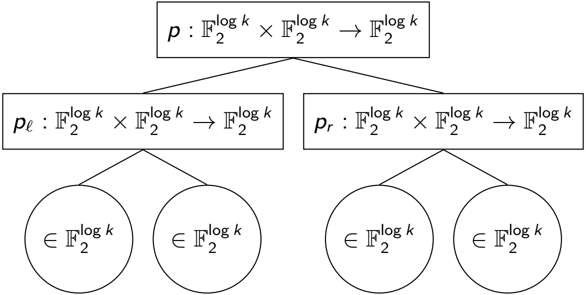
[**Cook and Mertz 2024**]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d \log k) \log(d \log k))$$

Arithmetize !



Arithmetize !



From multiplication to polynomial gates

[**Register Programs for Polynomial Gates**]
For all nodes g, there is a program P_g which results in

$$R_0 \leftarrow R_0 + v_g$$
$$R_i \leftarrow R_i \quad \forall i \neq 0$$

where $v_g \in \mathbb{F}_2^{\log k}$ is the value at the node $g \in T$ computed by the polynomial $p_g(\vec{y}, \vec{z}) : \mathbb{F}_2^{\log k} \times \mathbb{F}_2^{\log k} \to \mathbb{F}_2^{\log k}$
using $3 \log k$ registers holding values from $\mathbb{F}_2$ and $deg(p)$ recursive calls.

Building Blocks

- $m = |\mathbb{F}| - 1$.

Building Blocks

- $m = |\mathbb{F}| - 1$.
- Roots of unity of order m : $\omega \in \mathbb{F}$ such that $\omega^m = 1$.

Building Blocks

- $m = |\mathbb{F}| - 1$.
- Roots of unity of order m : $\omega \in \mathbb{F}$ such that $\omega^m = 1$.
- Primitive if $\forall m' < m$, $\omega^{m'} \neq 1$.

Building Blocks

- $m = |\mathbb{F}| - 1$.
- Roots of unity of order $m : \omega \in \mathbb{F}$ such that $\omega^m = 1$.
- Primitive if $\forall m' < m$, $\omega^{m'} \neq 1$.
- **Fact:** $\sum_{j=1}^{m} \omega_m^j = 0$.

Building Blocks

- $m = |\mathbb{F}| - 1$.
- Roots of unity of order m : $\omega \in \mathbb{F}$ such that $\omega^m = 1$.
- Primitive if $\forall m' < m$, $\omega^{m'} \neq 1$.
- **Fact:** $\sum_{j=1}^{m} \omega_m^j = 0$.
- **Fact:** $\sum_{j=1}^{m} \omega_m^{jb} = 0$ for all $0 < b < m$.

Building Blocks

- $m = |\mathbb{F}| - 1$.
- Roots of unity of order m : $\omega \in \mathbb{F}$ such that $\omega^m = 1$.
- Primitive if $\forall m' < m$, $\omega^{m'} \neq 1$.
- **Fact:** $\sum_{j=1}^{m} \omega_m^j = 0$.
- **Fact:** $\sum_{j=1}^{m} \omega_m^{jb} = 0$ for all $0 < b < m$.
- We can build indicators for $[b = 0]$

Building Blocks

- $m = |\mathbb{F}| - 1$.
- Roots of unity of order m : $\omega \in \mathbb{F}$ such that $\omega^m = 1$.
- Primitive if $\forall m' < m$, $\omega^{m'} \neq 1$.
- **Fact:** $\sum_{j=1}^{m} \omega_m^j = 0$.
- **Fact:** $\sum_{j=1}^{m} \omega_m^{jb} = 0$ for all $0 < b < m$.
- We can build indicators for $[b = 0]$
  There is ω_m and m^{-1} such that for all $0 \leq b < m$,

$$m^{-1} \sum_{j=1}^{m} \omega_m^{jb} = \begin{cases} 1 & \text{if } b = 0 \\ 0 & \text{otherwise} \end{cases}$$

The Root-of-unity Trick

$$\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega_m^j \tau_i + v_i)$$

The Root-of-unity Trick

$$\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega_m^j \tau_i + v_i) \quad = \quad \sum_{j=1}^{m} \sum_{S \subseteq [d]} \left(\prod_{i \in S} \omega_m^j \tau_i \right) \left(\prod_{i \notin S} v_i \right)$$

The Root-of-unity Trick

$$\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega_m^j \tau_i + v_i) = \sum_{j=1}^{m} \sum_{S \subseteq [d]} \left(\prod_{i \in S} \omega_m^j \tau_i \right) \left(\prod_{i \notin S} v_i \right)$$

$$= \sum_{j=1}^{m} \sum_{S \subseteq [d]} \omega_m^{j|S|} \left(\prod_{i \in S} \tau_i \right) \left(\prod_{i \notin S} v_i \right)$$

The Root-of-unity Trick

$$\begin{aligned}
\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega_m^j \tau_i + v_i) &= \sum_{j=1}^{m} \sum_{S \subseteq [d]} \left(\prod_{i \in S} \omega_m^j \tau_i \right) \left(\prod_{i \notin S} v_i \right) \\
&= \sum_{j=1}^{m} \sum_{S \subseteq [d]} \omega_m^{j|S|} \left(\prod_{i \in S} \tau_i \right) \left(\prod_{i \notin S} v_i \right) \\
&= \sum_{S \subseteq [d]} \left[\sum_{j=1}^{m} \omega_m^{j|S|} \right] \left(\prod_{i \in S} \tau_i \right) \left(\prod_{i \notin S} v_i \right)
\end{aligned}$$

The Root-of-unity Trick

$$\begin{aligned}
\sum_{j=1}^{m}\prod_{i=1}^{d}(\omega_m^j\tau_i+v_i) &= \sum_{j=1}^{m}\sum_{S\subseteq[d]}\left(\prod_{i\in S}\omega_m^j\tau_i\right)\left(\prod_{i\notin S}v_i\right)\\
&= \sum_{j=1}^{m}\sum_{S\subseteq[d]}\omega_m^{j|S|}\left(\prod_{i\in S}\tau_i\right)\left(\prod_{i\notin S}v_i\right)\\
&= \sum_{S\subseteq[d]}\left[\sum_{j=1}^{m}\omega_m^{j|S|}\right]\left(\prod_{i\in S}\tau_i\right)\left(\prod_{i\notin S}v_i\right) = m\prod_{i=1}^{d}v_i
\end{aligned}$$

$$\sum_{j=1}^{m} m^{-1}\prod_{i=1}^{d}(\omega_m^j\tau_i+v_i) = \prod_{i=1}^{d}v_i$$

The Root-of-unity Trick

$$\begin{aligned}
\sum_{j=1}^{m} \prod_{i=1}^{d} (\omega_m^j \tau_i + v_i) &= \sum_{j=1}^{m} \sum_{S \subseteq [d]} \left(\prod_{i \in S} \omega_m^j \tau_i \right) \left(\prod_{i \notin S} v_i \right) \\
&= \sum_{j=1}^{m} \sum_{S \subseteq [d]} \omega_m^{j|S|} \left(\prod_{i \in S} \tau_i \right) \left(\prod_{i \notin S} v_i \right) \\
&= \sum_{S \subseteq [d]} \left[\sum_{j=1}^{m} \omega_m^{j|S|} \right] \left(\prod_{i \in S} \tau_i \right) \left(\prod_{i \notin S} v_i \right) = m \prod_{i=1}^{d} v_i
\end{aligned}$$

$$\sum_{j=1}^{m} m^{-1} \prod_{i=1}^{d} (\omega_m^j \tau_i + v_i) = \prod_{i=1}^{d} v_i$$

Generalizing:

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

Register Program for Polynomial Evaluation

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

Register Program for Polynomial Evaluation

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

- Choose field to be $\mathbb{F}_{2^r}$ for $r > \log \deg(p) + 1$.
- Choose ω_m to be any generator of the multiplicative group.

for each j :

Prepare: $\forall i \in [n] \ : \ R_i \leftarrow R_i \omega_m^j$ $\qquad R_i = \tau_i \omega_m^j$

Load: $\forall i \in [n] \ :$ `Run` P_i $\qquad R_i = \tau_i \omega_m^j + v_i$

Unload: $\forall i \in [n] \ :$ `Run` P_i^{-1} $\qquad R_i = \tau_i \omega_m^j$

Unprepare: $\forall i \in [n] \ : \ R_i \leftarrow R_i \omega_m^{-j}$ $\qquad R_i = \tau_i$

Register Program for Polynomial Evaluation

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

- Choose field to be $\mathbb{F}_{2^r}$ for $r > \log \deg(p) + 1$.
- Choose ω_m to be any generator of the multiplicative group.

for each j :

Prepare: $\forall i \in [n] \ : \ R_i \leftarrow R_i \omega_m^j$ $\qquad R_i = \tau_i \omega_m^j$

Load: $\forall i \in [n] \ : \ \texttt{Run} \ \ P_i$ $\qquad R_i = \tau_i \omega_m^j + v_i$

Evaluate: $R_0 \leftarrow R_0 + m^{-1} p(R_1, R_2, \ldots R_n)$

Unload: $\forall i \in [n] \ : \ \texttt{Run} \ \ P_i^{-1}$ $\qquad R_i = \tau_i \omega_m^j$

Unprepare: $\forall i \in [n] \ : \ R_i \leftarrow R_i \omega_m^{-j}$ $\qquad R_i = \tau_i$

Implementing for $\mathrm{TEP}_{2,k,h}$

- In our case the polynomial is $p : \mathbb{F}_2^{\log k} \times \mathbb{F}_2^{\log k} \to \mathbb{F}_2^{\log k}$ at the node u with ℓ and r as the children.
- Let P_ℓ and P_r be the programs computing values v_ℓ and v_r.
- Recall i-th bit of the function can be written as:

$$f_u(y,z)_i = \sum_{(\alpha,\beta,\gamma)\in[k]^3} [\alpha_i = 1][f_u(\beta,\gamma) = 1][y = \beta][z = \gamma]$$

- Turn this into a $2\log k$-variate polynomial with degree $\leq 2\log k$.
- $[y = \beta]$ is same as $\prod_{i=1}^{\log k}(1 - y_i + (2y_i - 1)\beta_i)$ for $y_i \in \{0,1\}$.
- Number of registers is $3\log k$ each holding an element in $\mathbb{F}$.
- Number of instructions is $(4|\mathbb{F}|)^h \log k$.

Space used : $O((h + \log k)\log\log k)$

Demystifying the trick - Goldreich 2024

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

[Interpolation View]

(Improved) space used : $O(h \log\log k + \log k)$

Demystifying the trick - Goldreich 2024

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

[Interpolation View]

- Let $f_{u,i}(y, z)$ by the i-th bit of the function at node u.

(Improved) space used : $O(h \log\log k + \log k)$

Demystifying the trick - Goldreich 2024

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

[**Interpolation View**]

- Let $f_{u,i}(y, z)$ by the i-th bit of the function at node u.
- Define the multilinear extension of the function $\widehat{f} : \mathbb{F}^{\log k} \times \mathbb{F}^{\log k} \rightarrow \mathbb{F}$.

(Improved) space used : $O(h \log\log k + \log k)$

Demystifying the trick - Goldreich 2024

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

[Interpolation View]

- Let $f_{u,i}(y, z)$ by the i-th bit of the function at node u.
- Define the multilinear extension of the function $\widehat{f} : \mathbb{F}^{\log k} \times \mathbb{F}^{\log k} \to \mathbb{F}$.
- If we are given values of $\widehat{f}(\alpha \hat{x} + v_1, \alpha \hat{y} + v_2)$ for every $\alpha \in \{1, 2, \ldots 2 \log k + 1\}$, then we interpolate and find out the value of $\widehat{f}(0\hat{x} + v_1, 0\hat{y} + v_2) = \widehat{f}(v_1, v_2)$.

(Improved) space used : $O(h \log \log k + \log k)$

Demystifying the trick - Goldreich 2024

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

[**Interpolation View**]

- Let $f_{u,i}(y, z)$ by the i-th bit of the function at node u.
- Define the multilinear extension of the function $\widehat{f} : \mathbb{F}^{\log k} \times \mathbb{F}^{\log k} \to \mathbb{F}$.
- If we are given values of $\widehat{f}(\alpha\hat{x} + v_1, \alpha\hat{y} + v_2)$ for every $\alpha \in \{1, 2, \ldots 2\log k + 1\}$, then we interpolate and find out the value of $\widehat{f}(0\hat{x} + v_1, 0\hat{y} + v_2) = \widehat{f}(v_1, v_2)$.
- The above choices based on ω_m are specific points for evaluation and interpolation.

Demystifying the trick - Goldreich 2024

$$\sum_{j=1}^{m} m^{-1} p(\omega_m^j \tau_1 + v_1, \ldots, \omega_m^j \tau_n + v_n) = p(v_1, v_2, \ldots v_n)$$

[Interpolation View]

- Let $f_{u,i}(y,z)$ by the i-th bit of the function at node u.
- Define the multilinear extension of the function $\widehat{f} : \mathbb{F}^{\log k} \times \mathbb{F}^{\log k} \to \mathbb{F}$.
- If we are given values of $\widehat{f}(\alpha\hat{x} + v_1, \alpha\hat{y} + v_2)$ for every $\alpha \in \{1, 2, \ldots 2\log k + 1\}$, then we interpolate and find out the value of $\widehat{f}(0\hat{x} + v_1, 0\hat{y} + v_2) = \widehat{f}(v_1, v_2)$.
- The above choices based on ω_m are specific points for evaluation and interpolation.

(Improved) space used : $O(h \log\log k + \log k)$

Binary Alphabet Case and KRW Conjecture

$g_1 : \{0,1\}^{n_1} \to \{0,1\}$, $g_2 : \{0,1\}^{n_2} \to \{0,1\}$,

$$g_1 \circ g_2 \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n_2} \\ x_{21} & x_{22} & \dots & x_{2n_2} \\ \vdots & \dots & \dots & \vdots \\ x_{n_1 1} & x_{n_1 2} & \dots & x_{n_1 n_2} \end{pmatrix} = g_1(g_2(x_{11} \dots x_{1n_2}), \dots g_2(x_{n_1 1} \dots x_{n_1 n_2}))$$

Binary Alphabet Case and KRW Conjecture

$g_1 : \{0,1\}^{n_1} \to \{0,1\}$, $g_2 : \{0,1\}^{n_2} \to \{0,1\}$,

$$g_1 \circ g_2 \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n_2} \\ x_{21} & x_{22} & \dots & x_{2n_2} \\ \vdots & \dots & \dots & \vdots \\ x_{n_1 1} & x_{n_1 2} & \dots & x_{n_1 n_2} \end{pmatrix} = g_1(g_2(x_{11} \dots x_{1n_2}), \dots g_2(x_{n_1 1} \dots x_{n_1 n_2}))$$

[**KRW Conjecture (KRW 1995)**]

$$\text{depth}(g_1 \circ g_2) \geq \text{depth}(g_1) + \text{depth}(g_2) - O(1)$$

Binary Alphabet Case and KRW Conjecture

$g_1 : \{0,1\}^{n_1} \to \{0,1\}$, $g_2 : \{0,1\}^{n_2} \to \{0,1\}$,

$$g_1 \circ g_2 \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n_2} \\ x_{21} & x_{22} & \dots & x_{2n_2} \\ \vdots & \dots & \dots & \vdots \\ x_{n_1 1} & x_{n_1 2} & \dots & x_{n_1 n_2} \end{pmatrix} = g_1(g_2(x_{11} \dots x_{1n_2}), \dots g_2(x_{n_1 1} \dots x_{n_1 n_2}))$$

[**KRW Conjecture (KRW 1995)**]

$$\mathrm{depth}(g_1 \circ g_2) \geq \mathrm{depth}(g_1) + \mathrm{depth}(g_2) - O(1)$$

If true, $\mathrm{depth}(\mathrm{TEP}_{d,2,h})$ is $\Omega(dh)$

Binary Alphabet Case and KRW Conjecture

$g_1 : \{0,1\}^{n_1} \to \{0,1\}$, $g_2 : \{0,1\}^{n_2} \to \{0,1\}$,

$$g_1 \circ g_2 \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n_2} \\ x_{21} & x_{22} & \dots & x_{2n_2} \\ \vdots & \dots & \dots & \vdots \\ x_{n_1 1} & x_{n_1 2} & \dots & x_{n_1 n_2} \end{pmatrix} = g_1(g_2(x_{11} \dots x_{1n_2}), \dots g_2(x_{n_1 1} \dots x_{n_1 n_2}))$$

[**KRW Conjecture (KRW 1995)**]

$$\text{depth}(g_1 \circ g_2) \geq \text{depth}(g_1) + \text{depth}(g_2) - O(1)$$

If true, depth($\text{TEP}_{d,2,h}$) is $\Omega(dh)$

$\text{TEP}_{d,2,h} \notin \text{NC}^1$ when dh is $\omega(\log n)$

Binary Alphabet Case and KRW Conjecture

$g_1 : \{0,1\}^{n_1} \to \{0,1\}$, $g_2 : \{0,1\}^{n_2} \to \{0,1\}$,

$$g_1 \circ g_2 \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n_2} \\ x_{21} & x_{22} & \dots & x_{2n_2} \\ \vdots & \dots & \dots & \vdots \\ x_{n_1 1} & x_{n_1 2} & \dots & x_{n_1 n_2} \end{pmatrix} = g_1(g_2(x_{11} \dots x_{1n_2}), \dots g_2(x_{n_1 1} \dots x_{n_1 n_2}))$$

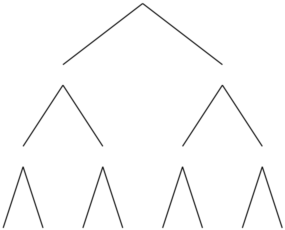
[**KRW Conjecture (KRW 1995)**]

$$\text{depth}(g_1 \circ g_2) \geq \text{depth}(g_1) + \text{depth}(g_2) - O(1)$$

If true, depth($\text{TEP}_{d,2,h}$) is $\Omega(dh)$

$\text{TEP}_{d,2,h} \notin \text{NC}^1$ when dh is $\omega(\log n)$

A random function has $g : \{0,1\}^d \to \{0,1\}$, w.h.p, requires $\Omega(d)$ depth.



Implications : KRW $\implies$ $\mathsf{NC}^1 \neq \mathsf{L}$

[**Cook and Mertz 2024**]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d\log k)\log(d\log k))$$

Implications : KRW $\implies$ $\mathsf{NC}^1 \neq \mathsf{L}$

[Cook and Mertz 2024]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d\log k)\log(d\log k))$$

- Pad $\mathrm{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros

Implications : KRW $\implies$ $NC^1 \neq L$

[Cook and Mertz 2024]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d \log k) \log(d \log k))$$

- Pad $\mathrm{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros
- We can solve the TEP instance in L.

Implications : KRW $\implies$ $\mathsf{NC}^1 \neq \mathsf{L}$

[Cook and Mertz 2024]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d \log k) \log(d \log k))$$

- Pad $\mathrm{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros
- We can solve the TEP instance in L.
- If $\mathsf{L} = \mathsf{NC}^1$, this contradicts the KRW conjecture since it results in $O((h + d)\log d)$ formula depth - which is $o(dh)$.

Implications : KRW $\implies$ Formulas $\neq$ Branching Programs

[**Cook and Mertz 2024**]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d \log k) \log(d \log k))$$

- Pad $\text{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros

Implications : KRW $\implies$ Formulas $\neq$ Branching Programs

[**Cook and Mertz 2024**]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d \log k) \log(d \log k))$$

- Pad $\mathrm{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros
- Fix $d = \log n$ and $h = \frac{\log n}{\log \log n}$.

Implications : KRW $\implies$ Formulas $\neq$ Branching Programs

[Cook and Mertz 2024]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d \log k) \log(d \log k))$$

- Pad $\text{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros
- Fix $d = \log n$ and $h = \frac{\log n}{\log \log n}$.
- Input size is $N = 2^{O(\log n \log \log n)}$

Implications : KRW $\implies$ Formulas $\neq$ Branching Programs

[**Cook and Mertz 2024**]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d \log k) \log(d \log k))$$

- Pad $\text{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros
- Fix $d = \log n$ and $h = \frac{\log n}{\log \log n}$.
- Input size is $N = 2^{O(\log n \log \log n)}$
- Function computable by a BP of size poly in N.

Implications : KRW $\implies$ Formulas $\neq$ Branching Programs

[Cook and Mertz 2024]
$TEP_{d,k,h}$ can be solved in space

$$O((h + d\log k)\log(d\log k))$$

- Pad $\mathrm{TEP}_{d,2,h}$ instance with $2^{(h+d)\log d}$ zeros
- Fix $d = \log n$ and $h = \frac{\log n}{\log\log n}$.
- Input size is $N = 2^{O(\log n \log\log n)}$
- Function computable by a BP of size poly in N.
- By KRW conjecture, the formula depth is $\Omega(dh) = \Omega\left(\frac{\log^2 N}{\log^3 \log N}\right)$

What next?

- Can TEP be solved in $O(\log n)$ space?

What next?

- Can TEP be solved in $O(\log n)$ space?
- Does non-determinism help? Can we show TEP $\in$ NL?

What next?

- Can TEP be solved in $O(\log n)$ space?
- Does non-determinism help? Can we show TEP $\in$ NL?
- Are there combinatorial counter parts to these algorithms?

What next?

- Can TEP be solved in $O(\log n)$ space?
- Does non-determinism help? Can we show TEP $\in$ NL?
- Are there combinatorial counter parts to these algorithms?
- Is there a direct combinatorial catalytic algorithm for reachability?

Thank You