Beating Neciporuk on the Tree Evaluation Problem

July 26, 2017

1 Easy Problem

Really guys! Here is a version of the problem that I cant get a lower bound for. And it is SO SO obvious.

Let z and x be in $[k]$. (k is a prime)
Let $pi(z, x) = x + iz \ mod \ k$.
Player Q_i, knows $p_i = x + iz$, but actually has zero information about x or about z.
Note that each x' is equally likely to be the actual x.
He has to reveal something about f, but does not know which x' for which to give $f(x')$.
It is like he has no information at all. And hence can give next to no information about $f(x)$.

But here is the catch.
If two players Q_i and $Q_{i'}$ got together, they would know $x + iz$ and $x + i'z$ and hence could solve for both z and x. Then either could send $f(x)$.
But they can't talk to each other.
We have to use the fact they cant talk.

If the different players could be given different z, then the lower bound is easy. (See proof in PS)
But then two players could not solve for x and z.
Is there a link between these?

Prove that this cant work.

PS If $z_i(x)$ was the same for each player i or the players could be given different z, then lower bound goes through as follows.
We have that $|\{p_i(z(x), x) \mid x\}| \leq \sharp p_i/\sharp z$. (ignore log factors)
Alice wants to communicate f to Bob.
For each i and each p in $\{p_i(z(x), x) \mid x\}$, Alice communicates $Q_i(p, f)$.
This takes only $\sum_i \sharp p_i/\sharp z$ field elements.
Note for each x, Bob can now determine $f(x)$.
But f contains $\sharp x$ field elements.
Hence $\sum_i \sharp p_i/\sharp z >= \sharp x$.

2 Separate Bits

The intuition is that the i^{th} player from $P_i(x,z)$ knows only enough about x to narrow it down to some set of possible x' values and hence does not know for which x' it should send bits about $f(x')$. We consider the case where each player is restricted so that he can send the referee any number of bit about any one such $f(x')$ value, but is not allowed to send mixed bits. More specifically, let $q_i(p_i, x') \in \mathcal{R}$ denote the fraction of a field element that $Q_i(p_i, f)$ sends about $f(x')$ value. Because the player can send at most one fields element in total, we have that $\forall i, \forall p, \sum_{x'} q_i(p, x') \leq 1$. Because from all the players, the referee must obtain the entire field element $f(x)$ for the actual input x, we have that $\forall x, z,\ \sum_i q_i(P_i(x,z), x) \geq 1$.

$$\begin{aligned}
\sum_i |\{p_i\}| &= \sum_i \sum_{p_i} [1] && (1)\\
&\geq \sum_i \sum_{p_i} \left[\sum_x q_i(p_i, x) \right] && (2)\\
&\geq \sum_i \left[\sum_{p_i} \sum_{x, \exists z, p_i = P_i(x,z)} q_i(p_i, x) \right] && (3)\\
&= \sum_i \left[\sum_{x,z} q_i(P_i(x,z), x) \right] && (4)\\
&= \sum_{x,z} \left[\sum_i q_i(P_i(x,z), x) \right] && (5)\\
&\geq \sum_{x,z} [1] && (6)\\
&= |\{x\}||\{z\}| && (7)
\end{aligned}$$

The brackets in (3) is a sum over all the edges in the bipartite graph from values of x to the values of p_i. Because this graph has no double edges, the sum in the brackets in (4) is the same.

Russell's Predictability, says that if $Q_i(p_i, f)$ sends one field element about $\langle f(x_1), \ldots, f(x_{k^d}) \rangle$ then for any parameter t, it can be converted so that it sends everything about t of the $f(x')$ and sends at most $\frac{1}{t}$ of a field element about all the others. With this in mind, lets redo the above ideas except that $\forall i, \forall p$, there are t values x' such that $q_i(p, x') = 1$ and for the others $q_i(p, x') \leq \frac{1}{t}$. Then there is an upper bound and lower bound such that $\sum_i |\{p_i\}| = \frac{1}{t}|\{x\}||\{z\}|$.

For the lower bound, note that unless one of the players tells the referee about all of $f(x)$, then learning $\frac{1}{t}$ from each of the t players is not quite enough to learn the one field element $f(x)$. Hence, we might as well assume that the $\frac{1}{t}$ are rounded down to zero. This gives that $\forall i, \forall p, \sum_{x'} q_i(p, x') \leq t$. The same lower bound as above then works except for a lost factor of t.

The upper bound is as follows. For each i, let $P_i(x,z)$ give all of z and the first $\log(|\{P_i\}|) - \log(|\{z\}|)$ bits of x. This leaves $\frac{|\{x\}||\{z\}|}{|\{p_i\}|}$ different values of x' that are possible. Each of the t players send about t of these $f(x')$ for a total of t^2. Then set $t = \sqrt{\frac{|\{x\}||\{z\}|}{|\{p_i\}|}}$ so that all of the remaining $f(x')$ are learned. Then $\sum_i |\{p_i\}| = t|\{p\}| = \frac{1}{t}|\{x\}||\{z\}|$.

3 $\langle z, f(x)\rangle$ with $z_{\langle f,t\rangle}(x)$ not depending on t.

The goal is to prove that a branching program computing $g(f_1(x_{1,1},\ldots,x_{1,d}),\ldots,f_d(x_{d,1},\ldots,x_{d,d}))$ requires k^{2d-2} leaf reading states because it must use k^{d-1} to remember the first $d-1$ answers and another k^{d-1} to compute $f_d(x_{d,1},\ldots,x_{d,d})$ and these numbers need to be multiplied in order to remember the first while doing the second.

A necessary easier problem is getting a lower bound for $\langle z, f(x)\rangle$. Here $x = \langle x_1,\ldots,x_d\rangle$ are our usual leaf variables and $z = \langle z_1,\ldots,z_{d-1}\rangle = \langle f_1(x_{1,1},\ldots,x_{1,d}),\ldots,f_{d-1}(x_{d-1,1},\ldots,x_{d-1,d})\rangle$ denote the first $d-1$ answers. The restriction on the branching program is that z is known at the beginning and must be known at the end. This should take $|\{z\}|k^{d-1}$ leaf reading states because it must use $|\{z\}|$ to remember the z and another k^{d-1} to compute $f(x)$.

The only states we are going to talk about are leaf reading ones such that at least one of their k out going edges is an f read. Hence, we are just going to say "state" to talk about them. Let ℓ denote the maximum number of such states in any computation path. This is the number of alternations between leaf and f reading states.

Theorem 1 *Assume for each f, the "entanglement" between z and x in the branching program is the same at each level so that our detangling function $z_{\langle f,t\rangle}(x)$ depends on f and x but does not depend on t. Any such branching program solving $\langle z, f(x)\rangle$ requires at least $s \geq \frac{1}{d\log s}\cdot|\{z\}|\cdot k^{d-1}$ leaf reading states. (Assuming $|\{z\}| \leq k^d$, gives $s \geq \frac{1}{2d^2\log k}\cdot|\{z\}|\cdot k^{d-1}$.)*

Partition these states S into levels $S_1,\ldots,S_\ell$ so that a state α is in S_t if any computation path from the start state to it goes through at most t states. In this way, each computation path visits at most one state at each level. Let $P_{\langle f,t\rangle}(z,x) \in [s_t]$ denote the at most one level t state that the computation goes through on input $\langle z,x,f\rangle$. Let $s_t = |S_t|$ with $\sum_t s_t = s$.

A key property of this function $p = P_{\langle f,t\rangle}(z,x)$ is that from it and rereading x (for fixed f), z needs to be recoverable. Specifically there needs to be a function $P^{-1}_{\langle f,t\rangle}(p,x) = z$. Informally, we would like to say that "# bits in $p = P_{\langle f,t\rangle}(z,x)$ about z" must be at least $\log|\{z\}|$ because z needs to be recovered in the end. Exponentiating this gives that the number of values of P when varying only z is at least $|\{z\}|$. We formalize this as follows.

Claim 2 *For two different values z' and z'', the states $P_{\langle f,t\rangle}(z',x)$ and $P_{\langle f,t\rangle}(z'',x)$ are distinct.*

Proof: If this state is in the computational paths of both $\langle z',x,f\rangle$ and $\langle z'',x,f\rangle$, then the value of z cannot be recovered at the output. ∎

Informally, we would like to say that "# bits in $p = P_{\langle f,t\rangle}(z,x)$ about x" is at most "the # bits in p" minus "# bits in p about z". The number of bits in $P(z,x)$ is clearly $\log|\{P\}| = \log s$. Then because z must be recovered in the end, the number about z must be the $\log|\{z\}|$. Exponentiating this gives that the number of values of P when varying only x is at most $\frac{s}{|\{z\}|}$.

A troublesome example occurs when the function $P_{\langle f,t\rangle}(z,x)$ computes $z+x$. Entropy is not good at telling what one knows about x or z when one knows $z+x$. Russell Impagliazzo's idea of *Predictability*, instead fixes the value of z as a function of the value of x. In this example, let $z(x) = -x$. Then knowing $z(x)+x$ always gives you the same value zero and all values of x are still possible. These ideas inspired the following proof. For each $x \in [k]^d$, we map one value $z_{\langle f,t\rangle}(x)$ for z such that the number of states for these restricted inputs is as small as possible.

Lemma 3 *There exists a choice $z_{\langle f,t\rangle}(x)$ for z for each x, such that $\left|\left\{P_{\langle f,t\rangle}(z_{\langle f,t\rangle}(x),x) \mid x\right\}\right| \leq \frac{s_t}{|\{z\}|}d\log k$.*

To understand this lemma better consider the following two examples. If $P(z,x) = \langle z, \ell \text{ bits of } x\rangle$, then $|\{P\}| = s = |\{z\}| \times 2^\ell$. Set $z(x) = 0$. Then as needed $|\{P(0,x) \mid x\}| = 2^\ell = \frac{s_t}{|\{z\}|}$. On the other hand, if $P(z,x) = z + x$, then $|\{P\}| = s = |\{z\}| = |\{x\}|$. Set $z(x) = -x$. Then as needed $|\{P(-x,x) \mid x\}| = 1 = \frac{s_t}{|\{z\}|}$.

Proof: We use a greedy algorithm. Suppose by induction on r, we have mapped a value $z(x)$ for all x but those in X_r, the mapped x lead to at most r values of $P(z(x),x)$, and P_r denotes the unmapped values of $P(z,x)$. Make a bipartite graph between X_r and P_r with an edge between x and p if there exists a z such that $P(z,x) = p$. By the previous claim, for each x, each value of z contributes a distinct edge. Hence, the number of edges in this graph is $|X_r| \cdot |\{z\}|$. Hence, there exists state p whose degree is at least the average $|X_r| \cdot \frac{|\{z\}|}{s_t}$. For each of the at least this number of values x with an edge to this p, map x to the $z(x)$ that put this edge in the graph. This gives $|X_{r+1}| \leq |X_r| - |X_r| \cdot \frac{|\{z\}|}{s_t}$, increases the number of $P(z(x),x)$ mapped to from r to $r+1$, and removes p from P_r. We started with $|X_0| = k^d$ unmapped values of $x \in [k]^d$. We are done when every value of x is mapped, i.e. $|X_r| < 1$. This occurs at least by $|X_r| \leq k^d \left(1 - \frac{|\{z\}|}{s_t}\right)^r \leq k^d e^{-r|\{z\}|/s_t} < 1$, giving $r \leq \frac{s_t}{|\{z\}|} d \log k$ as required. ∎

We now prove the lower bound for computing $\langle z, f(x)\rangle$ using a compression/communication argument.

Proof: (Theorem 1) By the statement of the theorem, our detangling function $z_{\langle f,t\rangle}(x) = z_f(x)$ depends on f and x but does not depend on t.

Alice wants to communicate f to Bob. Both know the branching program. For each level t and each value $x \in [k]^d$, Alice finds the state $P_{\langle f,t\rangle}(z_f(x),x)$. Being a leaf reading state, there are k edges out of it. From each of these, she follows the computation path with value f past all the f reading states until she gets to another leaf reading state. She uses $\log s$ bits to communicate to Bob which of the s leaf reading states was arrived at. The total number of bits communicated is $\sum_t \left|\left\{P_{\langle f,t\rangle}(z(x),x) \mid x\right\}\right| \cdot k \cdot \log s \leq \sum_t \frac{s_t}{|\{z\}|} d \log k \cdot k \cdot \log s = \frac{s}{|\{z\}|} dk \log k \log s$.

The claim is that from this Bob can recover all of f. Not knowing f, he does not know the mapping from x to $z_f(x)$. Hence, for each tuple $\langle z, x\rangle$, he runs the branching program on input $\langle z, x, f\rangle$. The only place he may have a challenge is when he gets to an f reading state. For some of the x to f transition states, Alice has told him the next leaf reading state that he will get to. If the computation path that he is following on $\langle z, x, f\rangle$ only passes through these revealed states then he can follow it and learn $f(x)$. If not he aborts and tries another z for this x. When he tries $z = z_f(x)$ he succeeds.

The entire function f is described by $k^d \log k$ bits. This gives $\frac{s}{|\{z\}|} dk \log k \log s \geq k^d \log k$ or $s \geq \frac{1}{d \log s} |\{z\}| \cdot k^{d-1}$. We are assuming that $|\{z\}| \leq k^d$ so that $s \leq k^{2d}$ and $\log s \leq 2d \log k$. This gives the result. ∎

Removing the restriction that $z_{\langle f,t\rangle}$ does not depend on t: The quick fix would to have Alice communicate for each $t \in [\ell]$. This would add an extra $\frac{1}{\ell}$ factor in the result. Yes, a $\ell = k^d$ factor would be a shame but still progress. But this does not work either. If we get x through level t by communicating about state $P_{\langle f,t\rangle}(z_f(x),x)$ and through level t' by communicating about state $P_{\langle f,t'\rangle}(z_{\langle f,t'\rangle}(x),x)$, we have not gotten one instance $\langle x, z_f(x)\rangle$ or $\left\langle x, z_{\langle f,t'\rangle}(x)\right\rangle$ through. Also the x and x' that come together in the same state in one level, can do completely different things at different levels.

4 An Easier Necessary Problem

With Ian

The goal is to prove that a branching program computing $g(f_1(x_{1,1},\ldots,x_{1,d}),\ldots,f_d(x_{d,1},\ldots,x_{d,d}))$ requires k^{2d-2} leaf reading states because it must use k^{d-1} to remember the first $d-1$ answers and another k^{d-1} to compute $f_d(x_{d,1},\ldots,x_{d,d})$ and these numbers need to be multiplied to remember both.

A necessary easier problem is getting a lower bound for $\langle z, f(x)\rangle$. Here $x = \langle x_1,\ldots,x_d\rangle$ are our usual leaf variables and $z = \langle z_1,\ldots,z_{d-1}\rangle = \langle f_1(x_{1,1},\ldots,x_{1,d}),\ldots,f_{d-1}(x_{d-1,1},\ldots,x_{d-1,d})\rangle$ denote the first $d-1$ answers. The restriction on the branching program is that z is known at the beginning and must be known at the end. This should take $|\{z\}|k^{d-1}$ leaf reading states because it must use $|\{z\}|$ to remember the z and another k^{d-1} to compute $f(x)$.

Some intuition about this $\langle z, f(x)\rangle$ problem is the following. Given z and reading x, the branching program could easy compute some function like $P(z,x) = z+x$ using the same number of states needed to store z alone. The branching program can recover z by subtracting x. In a counting sort of way, it also "knows" x. However with only $p = z + x$ it does not know where to query f. Learning $f(z + x)$ should not help. In fact, if f is "sensitive", then no non-thrifty query should help. The branching program needs to make the thrifty read $f(x)$. But to do that, it needs to know both z and x at the same time and this requires $|\{z\}|k^{d-1}$ leaf reading states.

Proving a lower bound for this problem becomes even easier when we restrict the branching program to have r pairs of stages followed by a final combining stage. (Ian and I proved the require lower bound with $r = 1$.) For $i \in [r]$, the first of the i^{th} pair of stages, knowing z and reading x, it ends up in the state specified by $p_i = P_i(z,x) \in [s]$. The outputs of each these phases must encode all of z, because there is a reverse function $P_i^{-1}(p_i,x) = z$ that can recover z. The second of the i^{th} pair of stages uses the information saved about x to query f, namely $q_i = Q_i(f,p_i) \in [k]$. The last stage combines these outputs to obtain the answer, namely $R(q_1,\ldots,q_d) = f(x)$.

The standard algorithm for $\langle z, f(x)\rangle$ fits into this form with $r = 1$. $p_1 = P_1(z,x) = \langle z,x\rangle$, $q_1 = Q_1(f,p_1) = f(x)$, and $R(q_1) = q_1$. Here $s = |\{z\}|k^d$. Note the $p_i = P_i(z,x) \in [s]$ stage has no access to f or else it could solve the problem with $P_i(z,x,f) = \langle z, f(x)\rangle$ with only $s = |\{z\}|k$ states.

It is easy to see that for first part $p_i = P_i(z,x) \in [s]$ of the i^{th} stage to end in s states and only contain x reads, it must contain at least $\frac{s}{k}$ leaf reading states, for a total of at least $\frac{rs}{k}$. Remembering $\langle q_1,\ldots,q_r\rangle$ may take k^r states, but we will give this and the computation of P_i, Q_i and R for free. The goal is to prove that $rs \geq |\{z\}|k^d$. An upper bound for this r stage problem translates into a branching program as follows. Suppose the first $i-1$ stages end in the branching program knowing z and $\langle q_1,\ldots q_{i-1}\rangle$ with $|\{z\}|k^{i-1}$ states. From here it reads x, computes $p_i = P_i(z,x) \in [s]$, forgets z and x, ending knowing p_i and $\langle q_1,\ldots q_{i-1}\rangle$ with $s \cdot k^{i-1}$ states. Then it reads f computes $q_i = Q_i(f,p_i) \in [k]$, forgets f, ending knowing p_i and $\langle q_1,\ldots q_i\rangle$ with $s \cdot k^i$ states. Then it reads x computes $P_i^{-1}(p_i,x) = z$, forgets p_i and x, ending knowing z and $\langle q_1,\ldots q_i\rangle$ as required for the end of the stage. The last stage combines these outputs to obtain the answer, namely $R(q_1,\ldots,q_d) = f(x)$.

The catalytic memory Ben-OR and Cleve algorithm computing $\times(x_1,x_2)$ fits into this form with $r = 4$. It starts with $\langle 0, z\rangle$ in its registers. The four first stages compute $p_1 = z$, $p_2 = x_1 - z$, $p_3 = x_2 - x_1 + z$, and $p_4 = x_2 + z$. Each second stage queries the function $f = \times$ by squaring its value, namely $q_i = Q_i(\times,p_i) = (p_i)^2$. The final stage subtracts these values and divides by two, namely $R(q_1,\ldots,q_d) = \frac{1}{2}(-q_1+q_2-q_3+q_4) = -(z)^2+(x_1-z)^2-(x_2-x_1+z)^2+(x_2+z)^2 = x_1 \times x_2$. Amazingly $s = |\{z\}| = |\{x\}| = k^d$ with no extra memory needed. The reason this works is because

$f = \times$ is never queried on the correct values $f(x_1, x_2)$, i.e. thriftily, but instead on four different values.[1]

Ian and I proved the required bound with $r = 1$. Consider the following set of f. Let $x = \langle x_1, \ldots, x_d \rangle$ and $a = \langle a_1, \ldots, a_d \rangle$. $f_0(x) = 0$ and $f_a(x) = 1$ if $a = x$ else is 0 Note $f_0(x)$ is a sensitive function. Hence, on every x, the branching program must make the thrifty read. We will use such a branching program to communicate/compress $\langle z, x \rangle$ using only $\log s$ bits of communication. Hence $s = |\{z\}|k^d$. Given $\langle z, x \rangle$, the first player specifies $p_1 = P_1(z, x) \in [s]$. The second player for each value of a computes $R(Q_1(f_a, p_1)) = f_a(x) \in \{0, 1\}$ and in so doing learns whether or not $a = x$. Now knowing x, he recovers $P_1^{-1}(p_1, x) = z$.

Given this, we figured that this set of f_a would be sufficiently hard for the general proof, but it is not. Here is a $ZLFLFLFLFLFLFLFL$ branching program with d leaf reads that works when restricted to these f_a.

For $i = 1..d$
    Reads x_i, then reads enough of f to know if $x_i = a_i$.
    if $x_i \neq a_i$, then quit and answer 0.
Quit and answer 1.

This easily fits into our model with $r = d$. For $i \in [d]$, $p_i = P_i(z, x) = \langle z, x_i \rangle$, $q_i = Q_i(f, p_i) =$ (if $x_i = a_i$), and $R(q_1, \ldots, q_r) =$ (if $x = a$) $= f_a(x)$. Here $s = |\{z\}|k$.

A yet even easier form of this problem becomes the number on a forehead problem. Let $r = d$. For $i \in [d]$, the i^{th} player receives $p_i = P_i(z, x) = \langle z, x_{j \neq i} \rangle$ where $x_{j \neq i}$ gives you all of x except for x_i because presumably x_i is on his forehead. Here $s = |\{z\}|k^{d-1}$, which we hope to show is not enough. The i^{th} player also receives f. Hence, he knows $f(x)$ is one of a column of k possible values. He, however, is only allowed to send a one way communication of one such value $q_i = Q_i(f, p_i) = Q_i(f, x_{j \neq i}) \in [k]$. The referee then must be able to combine these outputs to obtain the answer, namely $R(q_1, \ldots, q_d) = f(x)$. We have to show that this is not possible. Note that this number on forehead problem says nothing about z and hence our lower bound of k^{d-1} for the $f(x)$ problem should give a lower bound for this.

The intuition for the lower bound would go something like this. The player Q_i, knowing only $p_i = P_i(z, x) \in [s]$, knows only $\log s$ bits about $\langle z, x \rangle$. But he knows all of z. Hence, he knows only $\log s - \log |\{z\}|$ bits about x. Hence, the set $S_i \subseteq [k]^d$ of values x that he still thinks are possible has size on average $\frac{|\{z\}|k^d}{s}$. Player Q_i, however, is only allowed to send a one way communication of one value $q_i = Q_i(f, p_i) \in [k]$. The best he could do would be to send $\frac{s \log k}{|\{z\}|k^d}$ bits about each of the possible values $f(x')$ for $x' \in S_i$ consistent with p_i. The referee then must be able to combine these outputs to obtain the answer, namely $R(q_1, \ldots, q_d) = f(x)$. The referee knows a total of $r \cdot \frac{s \log k}{|\{z\}|k^d}$ bits about the required $\log k$ bits of $f(x)$. This gives $r \cdot \frac{s \log k}{|\{z\}|k^d} \geq \log k$ or $rs \geq k \cdot |\{z\}|k^d$ as we want for our lower bound.

What do you think?

[1] This Ben-OR and Cleve algorithm can be used to multiplying d numbers by applying this same idea recursively to a binary tree of dept d giving a width k^2 branching program with length $4^{depth} = 4^{\log_2 d} = d^2$. It would be nice, but I don't however think that this branching program fits into our $r = d^2$ framework. I can imagine that it fits into our $r = 2^d$ stages. For each $S \subseteq [d]$, let $p_S = z + \sum_{j \in S} x_j$ (likely with some subtractions). Then $q_S = Q_S(\times, p_S) = (p_S)^d$. Note that $q_{[d]} = (z + \sum_{j \in [d]} x_j)^d$ contains $\Pi_{j \in [d]} x_j$ as one of its terms. With $s = |\{z\}| = 2^d$ and $r = 2^d$, this would give $rs \geq |\{z\}|k^d$ as we want as our lower bound. Alternatively the model could be changed to allow P_i to depend on the previously computed $q_{i-1} = \Pi_{j \in [i-1]} x_j$ and use this to compute $q_i = q_{i-1} \times x_i$.

5 A Better Proof for d-Ary Tree of Height h

Consider the full d-ary tree of height h problem. The $n = 2^{h-1}$ leaves u are labeled with variables x_u. The vector of these is denoted $\vec{x}$. The internal nodes v are labeled with functions f_v. The vector of these is denoted $\vec{f}$.

Jeff and Steve's restriction on the branching program is that it does not read f twice in a row. The restriction here will be more complicated. Then we get close to the pebble bound.

Consider the computation path on input instance $\left\langle \vec{x}, \vec{f} \right\rangle$. For each internal node v in the tree, there must be a state along this path that queries $\vec{f}_v(\vec{a}_v)$ on the values $\vec{a}_v = \langle a_1, \ldots, a_d \rangle$ that arise from this instance at this node v. Otherwise, its value can be changed changing the required output. Colour such states **red**. Colour **black** the other $\vec{f}$ reading states that are querying $\vec{f}_v(a'_1, \ldots, a'_d)$ on the values $\langle a'_1, \ldots, a'_d \rangle$ that do not arise from input $\vec{x}$. Similarly colour the values in the matrix defining f either red or black. Colour **green** the leaf reading states.

The way the natural branching program completes its computation is to read the value x_u at some last leaf node u of the tree and then follow the path from it up to the root of the tree reading the appropriate value of f_v at each node v as it is reached. This means that the natural branching program has a sub-computation path consisting of a green state followed by $h-1$ red states. Now lets go back to consider the computation path on input instance $\left\langle \vec{x}, \vec{f} \right\rangle$ in our arbitrary branching program. For each leaf node u of the tree, consider the one green state and the $h-1$ red states corresponding to reading the path in the tree from u to the root. This set of states we will call a **tree path of red states**. We do know that each of the states in this set appears somewhere in the computation path, but we don't know where or in what order they will appear. Let γ_u denote the first state from this set that appears in this computation path and let δ_u denote the last. Let ℓ_u denote the total number states in this sub-computation path from γ_u to δ_u. Let $\ell = \min_u \ell_u$ and $\gamma = \gamma_u$ and $\delta = \delta_u$ be the corresponding states.

Theorem 4 *The* **condition** *on the branching program is that ℓ is bounded. Then number of states needed will be at least $s \geq \binom{\ell}{h}^{-1} k^{(d-1)(h-1)}$.*

This is much better than the previous version of the theorem which also had a k^{-b} factor, where b is the number of black states.

Note for the natural branching program, γ_u to δ_u in the computation path only contains the one green and $h-1$ red states themselves, and hence $\ell = h$ and $\binom{\ell}{h} = 1$ and the lower bound is (more or less) tight. There are only $n = 2^{h-1}$ leaf nodes of the tree and about the same number of internal nodes, each of which has only one function values that needs to be queried. Suppose the total length of each computation path in the branching program is at most c times this (i.e. the relevant variables are not read more than c times each and there are not that many black states). Then $\ell \leq c2^h$ so that $\binom{\ell}{h}$ is effectively a constant with respect to k (assuming c is). The worst case, however, is bad. Each function f_v is defined by k^d different values. Hence, even if they are only read once each there could be $\ell = k^d$ black states in a computation path. Then $\binom{\ell}{h} \approx k^{hd}$ which completely kills the result.

Proof: [With some handwavey parts] Alice knows an entire input instance $\left\langle \vec{f}, \vec{x} \right\rangle$. Alice sends Bob $\log s + \log \binom{\ell}{h} + N - (h-1)(d-1)\log k$ bits from which Bob learns all N bits needed to specify $\left\langle \vec{f}, \vec{x} \right\rangle$. This gives $s \geq \binom{\ell}{h}^{-1} k^{(d-1)(h-1)}$.

Alice's method is to use the branching program, which they both know. Alice uses $\log s$ bits to tell Bob state γ. Then she tells him enough about the input that Bob can traverse this computation path from state γ to δ. Alice sends $\log \binom{\ell}{h}$ bits to tell Bob which of these are the h red/green states giving the path from leaf u in the tree to the root. Each of these $h-1$ red states queries $f_v(a_1, \ldots, a_d)$ at specific values a_i that correspond to the values that reach node v of the tree on the actually instance held by Alice. Each of these values a_i restricts the set of inputs $\left\langle \vec{f}, \vec{x} \right\rangle$ by a factor of k. Hence, this provides $(h-1)d \log k$ disjoint bits of information about the instance $\left\langle \vec{f}, \vec{x} \right\rangle$ that Alice then does not have to provide.

At the beginning of the communication, Alice prematurely assumes that Bob already has these $(h-1)d \log k$ bits and sends Bob $N - (h-1)d \log k$ bits to tell him all the rest of the instance.

Bob, however, does not learn the path of red states until it is able to traverse the computation path from γ to δ. This is complicated by the fact that Bob needs some of this learned information in order traverse in order to learn this information. For example, again consider the natural branching program. State γ is the green state that reads x_u. Bob needs to be given the value of x_u by Alice so that he can follow this edge to the next red state that is reading the function f_v labeling the parent of u. Once Bob knows this red state is querying $f_v(\vec{x}_v)$, he learns the values $\vec{x}_v = \left\langle x_{\langle v,1 \rangle}, ..., x_{\langle v,d \rangle} \right\rangle$ that arise from the Alice's instance. One of these learned values, however, is the value of x_u that he just learned. Hence, he really only learns $d-1$ new values. This same thing repeats all the way up the tree from leaf u to the root. This is why Alice assume that Bob learns $(h-1)(d-1)$ values from the $h-1$ red states.

The handwavey part of the proof is that there are not any other complications of this sort, of Bob needing to know information to traverse before he learns it. Certainly any black state reveals information about the instance that is not learned from the $h-1$ red states and hence Alice will immediately be giving Bob its value well before Bob will need to transverse out of this state. ■

6 A Tuple of Functions

Consider this easier problem: The output is the tuple $\left\langle f_1(x_{\langle 1,1 \rangle}, ..., x_{\langle 1,d \rangle}), ..., f_d(x_{\langle d,1 \rangle}, ..., x_{\langle d,d \rangle}) \right\rangle$ with inputs being the f and the x. The obvious upper bound is as follows. After the branching program computes the outputs of the first $d-1$ functions, it must have k^{d-1} branches to remember their answers. Then it branches another k^{d-1} times to remember the first $d-1$ leaf values $\left\langle x_{\langle d,1 \rangle}, ..., x_{\langle d,d-1 \rangle} \right\rangle$. Hence, there are $k^{2(d-1)}$ leaf reading states that read the last leaf value $x_{\langle d,d \rangle}$. This then branches on its value to $k^{2(d-1)+1}$ states reading the specified value of f_d. We HOPE to prove a similar lower bound.

Theorem 5 *Computing* $\left\langle f_1(x_{\langle 1,1 \rangle}, ..., x_{\langle 1,d \rangle}), ..., f_d(x_{\langle d,1 \rangle}, ..., x_{\langle d,d \rangle}) \right\rangle$ *requires* $s \geq k^{2d-1}$ *function reading states and* $s \geq ?? \cdot k^{2d-2}$ *leaf reading states.*

*** OOPS I found a bug in the leaf reading state proof. Hence I give the function reading state proof first *****

Proof: [Function Reading States] Suppose Alice and Bob both know a branching program for this problem. Alice knows an entire input instance $\left\langle \vec{f}, \vec{x} \right\rangle$. She starts by telling Bob a specially chosen function reading state δ along this instance's computation path. She then tells Bob all but $2d-1$ of the $n = d(k^d + d)$ different $[k]$ values in the instance. From this, Bob is able to learn all n values in the instance. It follows that $\log s + (n - 2d + 1) \log k \geq n \log k$, giving that $s \geq k^{2d-1}$.

Consider the computation path on input $\left\langle \vec{x}, \vec{f} \right\rangle$. For each of the functions f_i for $i \in [d]$, there must be a state along this path that queries $f_i(\vec{x}_i)$ on the values $\vec{x}_i = \left\langle x_{\langle i,1 \rangle}, ..., x_{\langle i,d \rangle} \right\rangle$ that arise from the input. Otherwise, its value can be changed changing the required output. Colour such values $f_i(\vec{x}_i)$ and such states **red**. Colour **black** the other $\vec{f}$ reading states that are querying $f_i(\vec{x}'_i)$ on different values. Colour **green** the leaf reading states. Like Steve, let δ be the last red state in this instance's computation path.

Alice tells Bob state δ and enough about the input that Bob can traverse this computation path from state δ on to the output node. This of course requires telling Bob the value queried at each state along this path. For example, so that Bob can transit out of δ, she tells him the function value queried there. The state δ itself, however, is special because it is red. This means it is labeled with the values $\vec{x}_i = \left\langle x_{\langle i,1 \rangle}, ..., x_{\langle i,d \rangle} \right\rangle$ that actually appear in Alice's input instance. Bob, knowing that this state is red, now knows these d values. Hence, Alice need not separately tell him them. Bob will not encounter any other such red states, because δ was the last. Hence, Alice need not separately tell Bob the other $d-1$ outputs $f_{i'}(\vec{x}_{i'})$. Bob does, however, learn these values when he reaches the output node, because they are part of the output. Hence, as stated Alice need only tell Bob all but $2d-1$ of the $[k]$ values in the instance.

Concern: Suppose you have a matrix f_i of k^d values from $[k]$. How many bits does it take to tell you all of them but one when it is not known ahead of time which one is not being given. Note it takes $\log k^d$ bits to tell you which $\log k$ bits you are not going to tell! Solution: From the state δ, Bob learns which value is not being given to him, so Alice can just send a list of the other $k^d - 1$ other values and Bob can figure out which are which. ■

Proof: [Leaf Reading States with Bug] The proof is similar. Again let δ be the last red state of this instance's computation path and let γ be the first leaf reading state before it. Alice tells Bob state γ and enough about the input that Bob can traverse this computation path from state γ to δ and then on to the output node. So that Bob can transit out of γ, she tells him the value of the leaf variable read at γ. It is this information the changes the bound from $s \geq k^{2d-1}$ function reading states to $s \geq k^{2d-2}$ leaf reading states.

But there are more complications. Between γ and δ there may be other red states. This, however, is not a problem because for each such state, though Alice now must tell Bob the output $f_{i'}(\vec{x}_{i'})$, she no longer needs to tell him this function's d input values $\vec{x}_{i'} = \left\langle x_{\langle i',1 \rangle}, ..., x_{\langle i',d \rangle} \right\rangle$. (Though there is some complication about knowing which states are red.)

Lets just assume that from γ to the output state, δ is the only red state and the rest are black. Note there are $d \cdot (k^d - 1)$ black functions values. Hence there may in fact be this many black states between γ and the output. How does Bob know which of these is the red state δ? It may take $\log k^d$ bits for Alice to tell Bob. This kills the result, namely $\log s + \log k^d + (n - 2d + 2) \log k \geq n \log k$, giving that $s \geq k^{-d} \cdot k^{2(d-1)} = k^{d-1}$. ■

7 Extending Steve's Proof from Section 8

Consider the full d-ary tree of height h problem. The $n = 2^{h-1}$ leaves v are labeled with variables x_v. The vector of these is denoted $\vec{x}$. The internal nodes v are labeled with functions f_v. The vector of these is denoted $\vec{f}$.

Jeff and Steve's restriction on the branching program is that it does not read f twice in a row. The restriction here will be more complicated. Then we get close to the pebble bound.

Let $\vec{f}$ be the plus function on each node. Yes an easy function to compute. Consider the computation path on input $\left\langle \vec{x}, \vec{f} \right\rangle$. For each internal node v in the tree, there must be a state

along this path that queries $\vec{f}_v(\vec{x}')$ on the values $\vec{x}' = \langle a_1, \ldots, a_d \rangle$ that arise from input $\vec{x}$ at this node v. Otherwise, its value can be changed changing the required output. Colour such states **red**. Colour **black** the other $\vec{f}$ reading states that are querying $\vec{f}_v(a'_1, \ldots, a'_d)$ on the values $\langle a'_1, \ldots, a'_d \rangle$ that do not arise from input $\vec{x}$. Colour **green** the leaf reading states.

Theorem 6 (Generalized Steve's Proof) *The* **condition** *on the branching program is that for every $\vec{x}$, there is a sub-computation path of some short length $\ell = r + b + g$ from some green state $\gamma_{\vec{x}}$ to some red state $\delta_{\vec{x}}$ within which the number of red states is large, say $r = h$ (and not redoing info), the number of black states is small, say b, and the number of green states is small, say g. Then number of leaf reading states needed will be at least $\binom{\ell}{r,b,g}^{-1} k^{(d-1)r-b+1}$. (The first factor is consider to be a constant.)*

For example, in Steve's proof, this sub path has $d = 2$, $r = 1$, $b = 0$, and $g = 1$ and the lower bound is the same $k^{(d-1)r-b+1} = k^2$. In the obvious upper bound, the last thing that the branching program does is to read the right most leaf variable x_n and then queries the h functions along the path from this leaf to the root of the tree. This gives that this sub path has $r = h$, $b = 0$, and $g = 1$ and the lower bound is $k^{(d-1)h+1}$ closely matching the pebble bound.

Proof: The state $\gamma_{\vec{x}}$ will contain $k^{\ell-1}\binom{\ell}{r,b,g}$ pockets, one of which contains $\vec{x}$. We claim that each such pocket in the s leaf reading states of the branching program will contain at most k^{n-dr-g} inputs $\vec{x}$. Given there are k^n such inputs $\vec{x}$, this then proves that there are at least k^{dr+g} pockets. Hence there are at least $s \geq k^{dr+g} / \left[k^{\ell-1}\binom{\ell}{r,b,g}\right] = \binom{\ell}{r,b,g}^{-1} k^{(d-1)r-b+1}$ states. Recall $\ell = r + b + g$. What remains is to prove the claim.
Proof of claim: Starting at state $\gamma_{\vec{x}}$, there are at most $k^{\ell-1}$ states $\delta_{\vec{x}}$ that are at the end of a path of containing ℓ states. Then there are $\binom{\ell}{r,b,g}$ choices of which of these $\ell = r + b + g$ states in this path will be red, black, and green. For each of these $k^{\ell-1}\binom{\ell}{r,b,g}$ choices, state $\gamma_{\vec{x}}$ will have a pocket. Each of these r red states queries $f_v(a_1, \ldots, a_d)$ at specific values a_i. Each of these values a_i restricts the set of inputs $\vec{x}$ by a factor of k. The edge out of each of the g green nodes specifies the value of one variable x_v restricting the set of input $\vec{x}$ by another factor of k. Hence, the k^n values of $\vec{x}$ are restricted by a factor of k^{dr+g}. Of course we have to make sure that these restrictions don't repeat the same information. The statement of the claim follows. ■

Total of s states: If the entire branching program has at most s states, then the tree of $k^{\ell-1}$ paths of length ℓ can actually only contain s states. But there are about s^r ways to choose r of these to be red. This only gives $s \geq k^{dr}/s^r$, giving $s \geq k^d$.

8 A Direct Proof of Jeff's June 13, 2016 Theorem 2

Theorem 7 (Jeff's Theorem) *Any deterministic k-ary branching program computing $f(x_4 + x_5, x_6 + x_7)$ that does not read f twice in a row, when f is part of the input, requires $s \geq k^2$ leaf-reading states.*

Jeff's proof was based on analyzing a game with 5 players. We use his ideas to give a direct proof.

Proof: (direct) Let B be such a program. Fix the function f. We will associate with each leaf input $\ell = (a_4, a_5, a_6, a_7)$, a leaf-reading state γ_ℓ as follows: Let δ_ℓ be the first state of the

computation $C(\ell, f)$ on input (ℓ, f) which queries $f(a_4 + a_5, a_6 + a_7)$ (note that δ_ℓ must exist, since otherwise we could change the correct answer without changing the computation).

Let γ_ℓ be the state immediately preceding δ_ℓ in the computation $C(\ell, f)$. Since f cannot be read twice in a row, γ_ℓ queries some leaf input x_i and follows the edge labeled a_i to δ_ℓ.

CLAIM: State γ_ℓ can be associated with at most k^2 different leaf inputs (x_4, x_5, x_6, x_7).

The Theorem follows from the CLAIM, since there are k^4 different leaf inputs and $k^4/k^2 = k^2$.

PROOF of CLAIM: γ_ℓ has at most k successor states corresponding to the k values of the input variable x_i that it queries. If γ_ℓ is associated with input (x_4, x_5, x_6, x_7) then one of these k successor states, say δ', must query $f(x_4 + x_5, x_6 + x_7)$. But δ' can play this role for at most k leaf inputs (x_4, x_5, x_6, x_7) since x_i is fixed and so are the values of $x_4 + x_5$ and $x_6 + x_7$. ■

9 Adding g Layer

The *Tree Evaluation Problem* $FT_d^h(k)$ is the d-ary tree of height h. Each internal node i is labeled with a functions $f_i : [k]^d \Rightarrow [k]$. External nodes with elements from $[k]$.

Neciporuk[2] gives that a deterministic k-way branching program solving $FT_d^h(k)$ requires at least $\frac{d^{h-2}-1}{4(d-1)^2} \cdot k^{2d-1} \approx d^{h-4} \cdot k^{2d-1}$ states.

This paper considers considers $h = 4$. Computing a d-ary g at the bottom level takes k^d states. I am assuming that Steve's proof gives that the total number of states needed to compute $h = 4$ is at least k^d times the number of leaf reading states to compute $h = 3$. [3] Theorem 8 states that the later is at least $k^{d-(1/d)}$. This then gives abound for $h = 4$ of $k^{2d-(1/d)}$.

Actually do you multiply by k^d when adding g's or k^{d-1}. When $d = 2$, Steve was telling us to multiply by k^2. But the upper bound is $O(k^{(d-1)h-d+2})$, which looks like multiplying by k^{d-1}.

If you multiplying by k^{d-1}, then for $h = 4$ we only get $k^{2d-(1/d)-1}$ which is worse than Neciporuk.

Well that is a disappointment. My understanding is that you said that a k^4 lower bound for $h = 4$ and $d = 2$, would beat Neciporuk. But your Problem 1 clearly states that that is only an important step.

You need a $k^{3d.2} = k^{10}$ bound for $h = 4$ and $d = 4$. Wow! This is so different. If I am not mistaken, when computing $g(z_1, ..., z_d)$, you need to remember all z_i, before you can compute the g. This increased the number of states needed by a factor of k^{d-1}. But when computing $+(z_1, ..., z_d)$, you only have to remember the partial sum. This increased the number of states needed by a factor of k^1. You put at + at the second level. Hence I don't think k^{3d-2} needed but only $k^{2d-?}$.

10 Introduction

We count the number of the number of leaf reading states in a deterministic branching program. We consider computing $f\left(\sum_{j\in[r]} x_{\langle 1,j\rangle}, \ldots, \sum_{j\in[r]} x_{\langle \ell,j\rangle}\right)$, i.e. we consider the tree evaluation problem in which f has ℓ children, each of which is a + with in-degree r, each of which is a leaf variable $x_{\langle i,j\rangle}$. Each node is a value over $[k]$. The input consists of f and the leaves $x_{\langle i,j\rangle}$. Note that $\sum_{j\in[r]} x_{\langle i,j\rangle}$ can be computed with only k states by remembering the partial sum. Hence the whole function requires k^ℓ states. We give a lower bound of $k^{\ell-1/r}$. If the + was replaced by something harder, then $k^{\ell+r-1}$ may be needed. Sadly, Section 15 shows that even if the tree was ℓ-ary with height

[2] Figure 6, page 29 of Steve's paper

[3] Actually, I do not recall seeing the proof of this.

h with an arbitrary functions $f, g, .. : [k]^\ell \Rightarrow [k]$ at each node, our lower bound techniques cannot prove a bound larger than k^ℓ.

The following helps to justify why the lower bound is less than k^ℓ. When $r = 2$ change the problem so that the final output f is a single bit or equivalently we are only needing the branching program to learn one bit about the output. We have a surprising upper bound of $k^{\ell-0.27}$ moving away from the obvious upper bound of k^ℓ closer to our lower bound of $k^{\ell-0.5}$.

Theorem 8 *Any deterministic branching program computing* $f\left(\sum_{j\in[r]} x_{\langle 1,j\rangle}, \ldots, \sum_{j\in[r]} x_{\langle \ell,j\rangle}\right)$ *requires* s *leaf reading states:*

	$f \in [k]$	$f \in [2]$
$r \geq 2$	$k^{\ell-1/r} \leq s \leq k^\ell$	$k^{\ell-1/r}/\log k \leq s \leq k^\ell$
$r = 2$	"	" $\leq s \leq k^{\ell-0.27}$

I am quite sure that the branching program upper bound for $f(X + Y, U + V) \in [2]$ requires at at most $s \leq k^{1.73}$ leaf reading states and at most $k^{3.73}$ f reading states. The bound on the number of leaf reading states is a blow to Steve's leaf reading conjecture. When Steve adds a g layer to increase a k^2 leaf reading lower bound to a k^4 over all state lower bound, this upper bound contradicts this by giving $k^{3.73}$. Of course, the upper bound outputs a only a single bit where Steve's function outputs $\log k$ bits. Converting from a branching program computing a single bit to one computing $\log k$ bits, generally multiplies the the number of leaf reading states by a factor of k because it either has to be done in parallel or the previously computed bits of the output need to be remembered. If, on the other hand, the branching program were allowed to output the $\log k$ bits one at a time, then this only multiplies the number of leaf reading states by $\log k$. The latter is what our information theoretic lower bound reflects.

We also consider some special cases.

Theorem 9 *FLFL*

Theorem 10 *Any deterministic branching program,* that does not read f twice in a row*, computing* $f(x_4 + x_5, x_6 + x_7)$*, when* f *is part of the input, requires at least* $s \geq k^2$ *leaf reading states.*

This restriction says that a computation path can values $f(c_1, c_2)$ for many different tuples $c = \langle c_1, c_2\rangle$, but after such a read, it much read a leaf value before it can read $f(a', b')$ for another tuple $c' = \langle a', b'\rangle$. The effect is that for any single "block" of f reads, the branching program learns $f(c)$ for one and only c. This difference differentiates the hard and the easy versions of the following game.

11 Intuition

I just want to check to see if you understood what I meant that each part of f needs to be learned locally in some part of the branching program. This was already known in that for each of the k^d possible input values to f, f needs to be queried on these values somewhere in the branching program. This proves that the number of f reading states is at least k^d. But we are shifting from counting the number of f reading states to counting the number of leaf reading states (which possibly could have been a factor of k smaller).

The way the lower bound goes is that for each of the s leaf reading states u, the path from u to the next leaf reading state v communicates $\log(s)$ bits. The union of these $s\log(s)$ bits communicate all $k^d \log k$ bits of f according to the communication game. Locally the state u of the branching program is responsible for some of this communication.

We show that the number of leaf reading states is not a factor of k smaller then the number of f reading states, only possibly a factor of $k^{1/d}$ smaller. The reason we loose the $k^{1/d}$ is because the player responsible for state u also knows the value of the leaf in question – though it is not clear how this helps.

12 Games

We define a game that is a *number in the hand* game as each of the players has an input that the others do not know and is a *number on the forehead* game as they all know a number on the referee's forehead and he in turn knows their inputs.

Definition 11 *The* $\langle k, \ell, r\rangle$ function game *is defined as follows. The input consists of a* ℓ*-input function* $f : [k]^\ell \Rightarrow [k]$ *and* ℓr *inputs* $x_{\langle i,j\rangle} \in [k]$ *for* $i \in [\ell]$ *and* $j \in [r]$*. The required output is* $f\left(\sum_{j\in[r]} x_{\langle 1,j\rangle}, \ldots, \sum_{j\in[r]} x_{\langle \ell,j\rangle}\right)$*. There are* ℓr *players and a referee. Player* $\langle i,j\rangle$ *knows* $\left\langle f, x_{\langle i,j\rangle}\right\rangle$*. Simultaneously they each send some* $s_{\langle i,j\rangle}$ *bits to the referee. There is zero communication between them. For free, lets assume the referee knows the indexes* $x_{\langle i,j\rangle}$ *but knows nothing about* f *except what the players send him. He needs to learn the output. The cost is the total number of bits* $s = \sum_{i\in[\ell],j\in[r]} s_{\langle i,j\rangle}$ *sent to him.*

Lemma 12 *The* $\langle k, \ell, r\rangle$ *function game requires* $s \geq k^{\ell-1/r} \log k$ *bits.*

Proof: Suppose we have an algorithm for the $\langle k, \ell, r\rangle$ function game that communicates s bits. From this we design an algorithm in which the referee learns from the players $f(z_1, \ldots, z_\ell)$ for each $z_1, \ldots, z_\ell \in [k]$. Knowing that this requires the communication of $k^\ell \log k$ bits, gives us a bound on s. For each $i \in [\ell]$, the $\log k$ bits of z_i are partitioned into r parts. For $j \in [r]$, Player $\langle i,j\rangle$ will use his value $x_{\langle i,j\rangle}$ to specify the bits in the j^{th} part via the sum $z_i = \sum_{j\in[r]} x_{\langle i,j\rangle}$. Define $X_j = \left\{x_j = a \cdot (2^{\log(k)/r})^{j-1} \mid a \in \{0,1\}^{\log(k)/r}\right\}$ to be the set of values with zeros in the bits out side of his block. Note that each value of z_i can be made from combinations of these $x_{\langle i,j\rangle} \in X_j$, namely $\left\{z_i \in \{0,1\}^{\log(k)}\right\} = \left\{z_i = \sum_{j\in[r]} x_{\langle i,j\rangle} \mid x_{\langle i,1\rangle}, \ldots, x_{\langle i,r\rangle} \in X_j\right\}$. Knowing f, Player $\langle i,j\rangle$ for each $x_{\langle i,j\rangle} \in X_j$ sends the referee the $s_{\langle i,j\rangle}$ bits that he sends in the $\langle k, \ell, r\rangle$ function game protocol when knowing $\left\langle f, x_{\langle i,j\rangle}\right\rangle$. Note, because the players do not communicate with each other, we can know what one player will do without specifying the other player's inputs. The total number of bits sent is $\sum_{i\in[\ell],j\in[r]} |X_j| \cdot s_{\langle i,j\rangle} = |\{0,1\}^{\log(k)/r}| \cdot s = k^{1/r} \cdot s$. For each $z_1, \ldots, z_\ell \in [k]$, the referee can select one message from each player and determine $f(z_1, \ldots, z_\ell)$. Hence, the players must be sending $k^{1/r} \cdot s \geq k^\ell \log k$ bits, giving $s \geq k^{\ell-1/r} \log k$ as required. ■

Definition 13 *An* easier five person game *is the same except each player for each* $c = \langle c_1, c_2\rangle$ *either communicates all of* $f(c)$ *or none of it. Let* s_ℓ *denote number of* c *for which player* P_ℓ *reveals* $f(c)$ *and* $s = \sum_{\ell\in\{4,5,6,7\}} s_\ell$*.*

Lemma 14 *For the easier five person game to succeed,* $s \geq k^2$*.*

Proof: (Argument) If player P_4 knew $c_* = \langle x_4 + x_5, x_6 + x_7 \rangle$, he would give the value of $f(c_*)$. But he does not. Knowing x_4 alone gives him no information about which c needs $f(c)$ revealed. Hence, he gives $f(c)$ for a seemingly arbitrary set of c. Same with the other three players. The player P_c knows $c_* = x_4 + x_5$ and $f(c)$ for an "arbitrary" set of $\frac{k}{5}$ values of c. Does he necessarily know the value $f(c_*)$ for the correct c_*?

In the non-easy version of the game, things are harder because a player can communicate strange things like the parity of $f(c)$ over all c. But still knowing one of the leaves does not help the first four players. They can only communicate some s' bits about f. If they don't specify the full matrix f contains $k^2 \log k$ bits, then player P_c might not know the answer. ■

Proof: (Easy Version of Game) We merge players P_4 and P_6 into one $P_{\langle 4,6 \rangle}$. For each setting of $\langle x_4, x_6 \rangle$, let $C_{\langle x_4,x_6 \rangle}$ denote the set of $c = \langle c_1, c_2 \rangle$ for which this player reveals $f(c)$. By definition $s_{\langle x_4,x_6 \rangle} = |C_{\langle x_4,x_6 \rangle}|$. Similarly, define $C_{\langle x_5,x_7 \rangle}$. Form a bipartite graph with the k^2 nodes $u_{\langle x_4,x_6 \rangle}$ on the left and the k^2 nodes $v_{\langle x_5,x_7 \rangle}$ on the right. For each value of $\langle x_4, x_6 \rangle \in [k]^2$ and $c = \langle c_1, c_2 \rangle \in C_{\langle x_4,x_6 \rangle}$, we add a red edge edge between node $u_{\langle x_4,x_6 \rangle}$ and node $v_{\langle x_5,x_7 \rangle} = v_{\langle c_1 - x_4, c_2 - x_6 \rangle}$. The meaning of this edge is that on input $\langle x_4, x_5, x_6, x_7 \rangle$, player P_c learns the value of $f(c_*)$ from player $P_{\langle x_4,x_6 \rangle}$. The total number of such red edges is $|\{u_{\langle x_4,x_6 \rangle}\}| \times |C_{\langle x_4,x_6 \rangle}| = k^2 \times s_{\langle x_4,x_6 \rangle}$. Similarly add $k^2 \times s_{\langle x_5,x_7 \rangle}$ blue edges between $v_{\langle x_5,x_7 \rangle}$ and $u_{\langle x_4,x_6 \rangle} = u_{\langle c_1 - x_5, c_2 - x_7 \rangle}$ meaning that on input $\langle x_4, x_5, x_6, x_7 \rangle$, player P_c learns the value of $f(c_*)$ from player $P_{\langle x_5,x_7 \rangle}$. If by way of contradiction, $s = s_{\langle x_4,x_6 \rangle} + s_{\langle x_5,x_7 \rangle} < k^2$, then the total number of edges added is less than $k^2 \times k^2$. Hence, there is in input $\langle x_4, x_5, x_6, x_7 \rangle$ without an edge between $u_{\langle x_4,x_6 \rangle}$ and $v_{\langle x_5,x_7 \rangle}$. On this input, player P_c learns $f(c_*)$ from neither player. ■

13 Lower Bound on the Number of Leaf Reading States

The following lemma translates from the results about the communication game to results about the branching program. Note that $s \log s \geq k^2 \log k$ gives that $s \geq \frac{1}{2}k^2$. The $\frac{1}{2}$ comes from $\log s \approx \log k^2 = 2 \log k$.

Lemma 15 *Any branching program computing $f(x_4 + x_5, x_6 + x_7)$ when f is part of the input containing s leaf reading states can be translated into an algorithm for the five person game with $s' = s \log s$ bits communicated*[4]*. If the branching program does not read f twice in a row, then the algorithm is also for the easy version of the game with $s = s$ values $f(c)$ revealed.*

Proof: Assume all five players know the branching program. Denote the leaf reading states of the branching program $u_1, \ldots, u_s$. For $\ell \in \{4, 5, 6, 7\}$ and for each state u_i reading x_ℓ, player P_ℓ starts at state u_j and follows the edge labeled by the value of x_ℓ and then follows any edges out of f reading states until another leaf reading state u_j is reached. The player uses $\log s$ bits to indicate u_j (or the fact that some output state was reached instead). The fifth player knowing $\langle x_4, x_5, x_6, x_7 \rangle$ and the first four player's output can easily trace out the computation path through the branching program from the start state then from states u_i to u_j until an output state is reached determining $f(x_4 + x_5, x_6 + x_7)$. (In fact, player P_c does not even need to be told $\langle x_4, x_5, x_6, x_7 \rangle$ if given $\log s$ bits to handle an f reading start state.)

To clarify, consider the following examples. If state u_i reads x_ℓ and the edge labeled with its value goes to another reading state u_j, then the index of u_j is actually known by everyone already so does not need to communicated. Hence, the lower bound actually does not need to count this

[4] Actually $s' = s \log(s + 1)$.

state u_i. As a second case, suppose that the edge from u_i labeled with x_ℓ's value goes to state v that reads the value of $f(c)$ for some fixed $c = \langle c_1, c_2 \rangle$. Suppose that the branching program does not read f twice in a row. Then each edge out of v must go to a leaf reading state u_j. When player P_ℓ communicates the index of the next reading state u_j reached, all that is communicated is the value of $f(c)$ and nothing more. Note that this is in accordance with the easy version of the game. As a third case, suppose that this f reading state v is the root of some large and complex tree of f reading states. Then what P_ℓ communicates is some complex information about the matrix f. Note that if x_ℓ had a different value, then a different edge would be followed from state u_i to the root v' of a different tree of f reading states and the information about f might be different. ■

14 Upper Bounds

I am quite sure that the branching program upper bound for $f(X+Y, U+V) \in [2]$ requires at at most $s \leq k^{1.73}$ leaf reading states and at most $k^{3.73}$ f reading states.

Proof: (**??**) Let A be a function from $[k]$ to a single bit. Babai's paper top of page 21 expresses $A(X+Y)$ as

$$A(X+Y) = \sum_{|T_1| \leq d/2} \left(\sum_{|T_1 \cup T_2| \leq d} a_{T_1 \cup T_2} Y_{T_2} \right) \cdot X_{T_1} \; + \sum_{|T_2| \leq d/2} \left(\sum_{|T_1 \cup T_2| \leq d} a_{T_1 \cup T_2} X_{T_1} \right) \cdot Y_{T_2}.$$

Define the single bit $B_{T_1}(A, Y)$ to be $\sum_{|T_1 \cup T_2| \leq d} a_{T_1 \cup T_2} Y_{T_2}$ and the single bit $C_{T_2}(A, X)$ to be $\sum_{|T_1 \cup T_2| \leq d} a_{T_1 \cup T_2} X_{T_1}$, giving

$$A(X+Y) = \sum_{|T_1| \leq d/2} B_{T_1}(A, Y) \cdot X_{T_1} \; + \sum_{|T_2| \leq d/2} C_{T_2}(A, X) \cdot Y_{T_2}$$

The following is a branching program computing $A(X+Y)$. It has a layer for each $|T_1| \leq d/2$ and for each $|T_2| \leq d/2$. The layer starts with two states knowing the sum so far computing $A(X+Y)$. The branching program branches k ways on the value of Y. For each of these branches, it has a complex tree of A reading states computing $B_{T_1}(A, Y)$. These states collapses down to four states, knowing the sum so far and knowing $B_{T_1}(A, Y)$. The branching program then branches k ways on the value of X. It then completes the layer by adding $B_{T_1}(A, Y) \cdot X_{T_1}$ into the sum. Babai and friends has only $k^{0.73}$ bits of communication and hence I assume our branching program needs only four times as many leaf reading states.

The number of bits $B_{T_1}(A, Y) = \sum_{|T_1 \cup T_2| \leq d} a_{T_1 \cup T_2} Y_{T_2}$ that must be computed is $k^{1.73}$ because there are $k^{0.73}$ layers and in each branching program branches k ways on the value of Y. A single such value is a sum over the polynomial coordinates $a_{T_1 \cup T_2}$, each of which is a complex combination of $A(z)$ for all $z \in [k]$. But I am quite sure that each such coordinate is a simple sum of these, namely $a_{T_1 \cup T_2} = \sum_{z \in [k]} ?? A(z)$. Hence, the order of the two sums could be reversed, namely $B_{T_1}(A, Y) = \sum_{|T_1 \cup T_2| \leq d} \left[\sum_{z \in [k]} ?? f(z) \right] Y_{T_2} = \sum_{z \in [k]} \left[\sum_{|T_1 \cup T_2| \leq d} ?? f(z) \right] Y_{T_2}$. This allows $B_{T_1}(A, Y)$ to be computed with only k reads to A, for a total of $k^{2.73}$.

The branching program for $f(X+Y, U+V)$ first reads U and than V giving k states each knowing the value of $U+V$, it then follows the above protocol when $A(.)$ is the indicated column $f(U+V, .)$. This increases the number of leaf reading states to $k^{1.73}$ and f reading states to $k^{3.73}$. ■

15 Technique cannot be extended

Consider a ℓ-ary binary tree with height h with an arbitrary functions $f, g, .. : [k]^2 \Rightarrow [k]$ at each node. The pebble lower bounds would say that something like $h + \ell$ pebbles would be needed and hence the conjecture is that $(h+\ell)\log k$ space or $k^{h+\ell}$ states are needed. Our lower bound technique naturally translates into the following game. There is a player for each leaf who knows his leaf value and all of the $f, g, ..$ and simultaneously communicates to the referee who must know the output. There are ℓ^h functions $f, g, ..$ requiring at total of $\ell^h \times k^\ell \log k$ bits of information for the players to send. This gives a $\ell^h \times k^\ell$ lower bound on the number of states needed or $h \log \ell + \ell \log k$ bound on the space. I am assuming $\ell << k$. Hence, this is way too small.

16 Chat that nobody wants to read

The most powerful s' space algorithm is modeled by a branching program with $s = 2^{s'}$ states. The function $f(x_4 + x_5, x_6 + x_7)$ is an example of evaluating a binary tree of depth two, where the four leaves are labeled x_ℓ, their parents compute plus and the root computes f. Each node takes on a value from some field of size k. A lower bound for the corresponding pebble game states that d pebbles and hence $d \log k$ space or k^d states is needed to evaluate such a binary tree of depth d. Barington and friends prove that actually only $2 \log k + 2d$ space or $k^2 \times 4^d$ states is needed if all the internal nodes are plus and times. Steve and David assure us that a lower bound stating that k^2 leaf reading states are needed to compute $f(x_4 + x_5, x_6 + x_7)$ can be used to give a k^3 lower bound on the number states needed to evaluate a depth $d = 3$ tree. The function f can be thought of as k^2 variables giving the value of $f(c_1, c_2)$ for each $\langle c_1, c_2 \rangle \in [k]^2$. If the top node of that tree is included in the input then the total number of variables with values from $[k]$ will be $k^2 + 8$ and the number of bit variables will be $\ell \approx k^2 \log k$. Needing k^3 states translates to needing $\log k^3 = 3 \log k \approx 1.5 \log \ell$ space. This is slightly better than the obvious $\log \ell$ space bound and beats Barington's general $2 \log k + 2d$ upper bound when the internal functions are only plus and times and $k > 2^d$.

A branching program is a DAG. Each node represents a state the computation might be in. The number of such states is $s = 2^{\text{number of bits of computation space}}$. In such a state, the computation reads the value of some variable x and branches to another state based on this value, i.e. state u is labeled with a variable x. There is an edge from state u to state v labeled by r if the computation transitions from state r to state v when $x = r$. Let $x_1, x_2, x_3, x_4 \in [k]$ be the *leaf* values from a field of size k and let $f(c_1, c_2)$ be an arbitrary function from $[k]^2$ to $[k]$. The goal is to have a branching program compute $f(x_4 + x_5, x_6 + x_7)$. Having f part of the input means that for each c_1 and $c_2 \in [k]$, $f(c_1, c_2)$ is a variable that could be read at a state node. In fact, the size of the branching program must be at least k^2 because there must be a state reading $f(c_1, c_2)$ for each of the k^2 indexes $c = \langle c_1, c_2 \rangle \in [k]^2$. We, however, will only count *leaf reading states*, i.e. ones that reads one of the four variables x_ℓ.

The following story may help the intuition. Suppose a detective L is investigating about c_*=Cathy so he asks player F the colour of Cathy's hair to which F responds blond. A bad guy does not hear the detective's question but does hear F's answer. We do not care that he learns that Cathy's hair is blond. But we don't want the bad guy to learn that the detective is investigation about Cathy. This he inadvertently may learn, by hearing F answer "Cathy's hair is blond". F could simply answer "blond", but even then the bad guy learns that the detective is investigating about a blond person. Our solution is that in response to the detective's question, player F will give the colour of every person's hair, i.e. the answer to every question that the detective may

have asked. Though this means that bad guy learns the answer to way more questions than he normally would have learned, he does not learn which question the detective asked. Similarly, in our game. The matrix F is big needing lots of bits to describe it, so I don't mind everyone knowing this information that player F gives about $f(C_*)$. But learning the part C_* that c_* is contained in, gives too much information away about c.
