Pebbling and Branching Programs Solving the Tree Evaluation Problem

Dustin Wehr

February 15, 2010

Abstract

We study restricted computation models related to the *tree evaluation problem*. The TEP was introduced in earlier work as a simple candidate for the (*very*) long term goal of separating $\mathbf{L}$ and $\mathbf{LogDCFL}$. The input to the problem is a rooted, balanced binary tree of height h, whose internal nodes are labeled with binary functions on $[k] = \{1, \ldots, k\}$ (each given simply as a list of k^2 elements of $[k]$), and whose leaves are labeled with elements of $[k]$. Each node obtains a value in $[k]$ equal to its binary function applied to the values of its children. The output is the value of the root. The first restricted computation model, called *fractional pebbling*, is a generalization of the black/white pebbling game on graphs, and arises in a natural way from the search for good upper bounds on the size of nondeterministic branching programs solving the TEP - for any fixed h, if the binary tree of height h has fractional pebbling cost at most p, then there are nondeterministic branching programs of size $O(k^p)$ solving the height h TEP. We prove a lower bound on the fractional pebbling cost of d-ary trees that is tight to within an additive constant for each fixed d. The second restricted computation model we study is a semantic restriction on (non)deterministic branching programs solving the TEP – *thrifty* branching programs. Deterministic (resp. nondeterministic) thrifty BPs suffice to implement the best known algorithms, based on black pebbling (resp. fractional pebbling), for the TEP. In earlier work, for each fixed h a lower bound on the size of thrifty deterministic branching programs was proved that is tight for sufficiently large k. We give an alternative proof that achieves the same bound for all k and h. We also show the bound still holds in a less-restricted model.

1 Introduction

The motivations for this paper are those of [BCM$^+$09a], and the goals are to extend and improve on the results given there (with the exception of Theorem 5, which appeared there verbatim). But from a wider view, what we want is to improve our understanding of $\mathbf{L}$ in the hope that this will help in eventually separating it from (apparently) larger classes. We study the tree evaluation problem (TEP), which was defined in [BCM$^+$09b] and shown to be in $\mathbf{LogDCFL}$.

The function version of the *Tree Evaluation problem* $FT^h(k)$ is defined as follows. Let T^h be the balanced binary tree of height h (see Fig. 1). For each internal node i of T^h the input includes

a function $f_i : [k]\times[k] \to [k]$ specified as k^2 integers in $[k] = \{1, \ldots, k\}$. For each leaf the input includes an integer in $[k]$. We can then say that each internal tree node takes a value in $[k]$ by applying its function to the values of its children. The function problem $FT^h(k)$ is to compute the value of the root, and the decision version $BT^h(k)$ is to determine whether this value is 1.

Since $BT^h(k) \in \mathbf{LogDCFL}$, it is not hard to show that for *any* unbounded function $r(h)$, a lower bound of $\Omega(k^{r(h)})$ on the number of states for deterministic (resp. non-deterministic) branching programs solving $FT^h(k)$ or $BT^h(k)$ would separate $\mathbf{LogDCFL}$ and $\mathbf{L}$ (resp. $\mathbf{NL}$) [1]. To see this, note that inputs to $BT^h(k)$ can be encoded with $(2^{h-1}-1)k^2 \log k + 2^{h-1} \log k + O(1) = O(2^h k^2 \log k)$ bits, so it suffices to consider polynomial bounding function that are the product of a polynomial in 2^h and a polynomial in k, which $k^{r(h)}$ is not.

In [BCM+09b], the TEP was defined more-generally on balanced d-ary trees, where the functions attached to internal nodes are of type $[k]^d \to [k]$. The motivation was that tight lower bounds for height 3 and all d can be proved [BCM+09b], and proving the conjectured lower bound of $\Omega(k^7/\log k)$ states (with $h = 4$ and $d = 3$ fixed, so that the input size $n(k)$ is $O(k^3 \log k)$ bits or $O(k^3)$ $[k]$-valued variables) for unrestricted deterministic BPs would beat the best known lower bound of $\Omega(n^2/(\log n)^2)$ states for a problem in $\mathbf{NP}$, achieved using Nečiporuk's method [Neč66]. Since we are focusing on restricted computation models here, there is little to gain in including the parameter d. That being said, the fractional pebbling lower bound proved in Section 4.1 *is* given for arbitrary d.

2 Preliminaries

We write $[k]$ for $\{1, 2, \ldots, k\}$. For $h \geq 1$ we use T^h to denote the balanced binary tree of height h.

Warning: Here the *height* of a tree is the number of levels in the tree, as opposed to the distance from root to leaf. Thus T_2^2 has just 3 nodes.

We number the nodes of T^h as suggested by the heap data structure. Thus the root is node 1, and in general the children of node i are nodes $2i, 2i + 1$ (see Figure 1).

Definition 1 (Tree evaluation problems)**.**

> An input I for either the function or decision version of the problem includes: for each internal node i of T^h, a function $f_i^I : [k]\times[k] \to [k]$ represented as k^2 integers in $[k]$, and for each leaf node i, an integer $l_i^I \in [k]$.
>
> *Function evaluation problem* $FT^h(k)$: On input I, compute the value $v_1^I \in [k]$ of the root 1 of T^h, where in general $v_i^I = l_i^I$ if i is a leaf and $v_i^I = f_i^I(v_{2i}^I, v_{2i+1}^I)$ if i is an internal node.
>
> *Boolean evaluation problem* $BT^h(k)$: Accept I iff $v_1^I = 1$.

[1] Of course, doing so would actually yield the stronger result: Nonuniform $\mathbf{L} \not\subseteq \mathbf{LogDCFL}$ (resp. Nonuniform $\mathbf{NL} \not\subseteq \mathbf{LogDCFL}$).

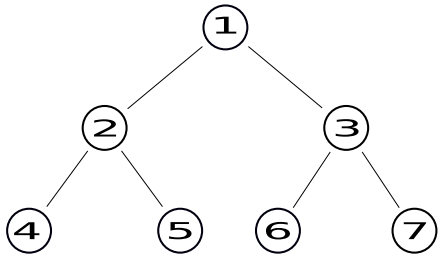


Figure 1: The height 3 binary tree T^3 with nodes numbered heap style.

2.1 Branching programs

We use the same branching program model as in [BCM+09a] and [BCM+09b].

Definition 2 (Branching programs)**.** A *nondeterministic k-way branching program* B computing a total function $g : [k]^m \to R$, where R is a finite set, is a directed rooted multi-graph whose nodes are called *states*. Every edge has a label from $[k]$. Every state has a label from $[m]$, except $|R|$ *output* sink states consecutively labeled with the elements from R. An input $(x_1, \ldots, x_m) \in [k]^m$ activates, for each $1 \leq j \leq m$, every edge labeled x_j out of every state labeled j. A *computation path* on input $\vec{x} = (x_1, \ldots, x_m) \in [k]^m$ is a directed path consisting of edges activated by $\vec{x}$ which begins with the unique start state and either ends in the final state labeled $g(x_1, \ldots, x_m)$ or is infinite. At least one such computation must end. The *size* of B is its number of states. B is *deterministic k-way* if every non-output state has precisely k outedges labeled $1, \ldots, k$.

We say that B solves a decision problem (relation) if it computes the characteristic function of the relation.

A k-way branching program computing $FT^h(k)$ or $BT^h(k)$ requires k^2 k-valued arguments for each internal node i of T^h in order to specify the function f_i, together with one k-valued argument for each leaf. Thus in the notation of the above definition, $FT^h(k) : [k]^m \to R$ where $R = [k]$ and $m = (2^{h-1} - 1)k^2 + 2^{h-1}$. Also $BT^h(k) : [k]^m \to \{0, 1\}$.

Important: Since we only study the tree evaluation problem (TEP) here, we give the input variables mnemonic names: $f_i(a, b)$ is an input variable (called an *internal node variable*) for every internal node i and $a, b \in [k]$ and l_i is an input variable (called a *leaf variable*) for every leaf i.

For fixed h we are interested in how the number of states required for a k-way branching program to compute $FT^h(k)$ and $BT^h(k)$ grows with k. This is why we write h in the superscript of $FT^h(k)$ and $BT^h(k)$. We define $\#\mathsf{detFstates}^h(k)$ (resp. $\#\mathsf{ndetFstates}^h(k)$) to be the mininum number of states required for a deterministic (resp. nondeterministic) k-way branching program to solve $FT^h(k)$. Similarly we define $\#\mathsf{detBstates}^h(k)$ and $\#\mathsf{ndetBstates}^h(k)$ to be the number of states required to $BT^h(k)$.

Thrifty programs are a restricted form of k-way branching programs for solving tree evaluation problems, introduced in [BCM+09a]. Thrifty programs efficiently simulate pebbling algorithms, and implement the best known upper bounds for $\#\mathsf{ndetBstates}^h(k)$ and $\#\mathsf{detFstates}^h(k)$, and are within a factor of $\log k$ of the best known for $\#\mathsf{detBstates}^h(k)$.

Definition 3 (Thrifty branching program)**.** A deterministic k-way branching program which solves $FT^h(k)$ or $BT^h(k)$ is ***thrifty*** if during the computation on any input every query $f_i(a, b)$ to an internal node i of T^h satisfies the condition that $\langle a, b\rangle$ is the tuple of correct values for the children of node i (i.e. $v^I_{2i} = a$ and $v^I_{2i+1} = b$). A non-deterministic such program is ***thrifty*** if for every input every computation which ends in a final state satisfies the above restriction on queries.

This is a strong restriction. For example, a deterministic thrifty BP cannot, for any internal node i, iterate over all the k^2 variables that define f_i, or even just two distinct f_i variables.

In [BCM+09a] the following theorem is given, showing how upper bounds for black pebbling and fractional pebbling yield upper bounds for deterministic and nondeterministic branching programs solving the TEP. The proof can be found in [BCM+09c].

Theorem ([BCM+09a]):

(i) If T^h can be black pebbled with p pebbles, then deterministic thrifty branching programs with $O(k^p)$ states can solve $FT^h(k)$ and $BT^h(k)$.

(ii) If T^h can be fractionally pebbled with p pebbles then non-deterministic thrifty branching programs can solve $BT^h(k)$ with $O(k^p)$ states.

Also in [BCM+09a], the following lower bound was given for deterministic thrifty programs. The proof can be found in [BCM+09c].

Theorem ([BCM+09a]): For all h, for $k > \binom{2^h}{h-1}$ every deterministic thrifty branching program solving $BT^h(k)$ requires at least $1/2k^h$ states.

Theorem 4 in Section 3, which is a special case of Theorem 6 in Section 4.2, gives a small improvement on that result. The main improvement is that it gives a tight bound that holds for all pairs k and h, rather than requiring that k be much larger than h. The constant $1/2$ also goes away: **Theorem 4 :** For all h, k every deterministic thrifty branching program solving $BT^h(k)$ requires at least k^h states.

2.2 Pebbling

The pebbling game for dags was defined by Paterson and Hewitt [PH70] and was used as an abstraction for deterministic Turing machine space in [Coo74]. Black-white pebbling was introduced in [CS76] as an abstraction of non-deterministic Turing machine space (see [Nor09] for a recent survey). Fractional pebbling was introduced in [BCM+09a].

Let us first define three versions of the pebbling game. We will not be proving anything about black-white pebbling directly, but fractional pebbling is a generalization of black-white pebbling, so it will be easier to define it first. The first is a simple 'black pebbling' game: A black pebble can be placed on any leaf node, and in general if all children of a node i have pebbles, then one of the pebbles on the children can be slid to i (this is a "black sliding move')'. Any black pebble can be removed at any time. The goal is to pebble the root, using as few pebbles as possible. The second version is 'whole' black-white pebbling as defined in [CS76] with the restriction that we do not allow "white sliding moves". Thus if node i has a white pebble and each child of i has a pebble (either black or white) then the white pebble can be removed. (A white sliding move would apply

if one of the children had no pebble, and the white pebble on i was slid to the empty child. We do not allow this.) A white pebble can be placed on any node at any time. The goal is to start and end with no pebbles, but to have a black pebble on the root at some time.

The third is *fractional pebbling*, which generalises whole black-white pebbling by allowing each node i to have a *black value* $b(i)$ and a *white value* $w(i)$ such that $b(i) + w(i) \leq 1$. The total pebble value (i.e. $b(i) + w(i)$) of each child of a node i must be 1 before the black value of i is increased or the white value of i is decreased. Figure 2 shows the sequence of configurations for an optimal fractional pebbling of the binary tree of height three using 2.5 pebbles.

Our motivation for choosing these definitions is that we want pebbling algorithms for trees to closely correspond to k-way branching program algorithms for the tree evaluation problem. If, as in the survey by Razborov [Raz91], we instead used *switching and rectifier networks* instead of nondeterministic branching programs, where input variable labels are on the edges rather than the nodes, and a node can have any number of out-edges, and the size of the program is defined as the number of edges, then we would get better upper bounds by using a variant of fractional pebbling where the following analogue of "white sliding moves" are allowed: Suppose you want to remove white value from an internal node i by first increasing the white value of one or both of the children of i. With white sliding moves, you can combine those two moves. A precise definition is given in [BCM$^+$09c], where it is also shown that the height 4 binary tree can be fractionally pebbled using white sliding moves with $8/3$ pebbles, from which it follows that there are switching and rectifier networks with $O(k^{8/3})$ edges that solve $BT^4(k)$. In contrast, it is shown in [BCM$^+$09c] that 3 pebbles are necessary and sufficient using our chosen definition of fractional pebbling.

Now we give the formal definition of fractional pebbling, and then define the other two notions as restrictions on fractional pebbling.

Definition 4 (Pebbling)**.** A *fractional pebble configuration* on a rooted d-ary tree T is an assignment of a pair of real numbers $(b(i), w(i))$ to each node i of the tree, where

$$0 \leq b(i), w(i) \tag{1}$$

$$b(i) + w(i) \leq 1 \tag{2}$$

Here $b(i)$ and $w(i)$ are the *black pebble value* and the *white pebble value*, respectively, of i, and $b(i) + w(i)$ is the *pebble value* of i. The number of pebbles in the configuration is the sum over all nodes i of the pebble value of i. The legal pebble moves are as follows (always subject to maintaining the constraints (1), (2)): (i) For any node i, decrease $b(i)$ arbitrarily, (ii) For any node i, increase $w(i)$ arbitrarily, (iii) For every node i, if each child of i has pebble value 1, then decrease $w(i)$ arbitrarily, increase $b(i)$ arbitrarily, and simultaneously decrease the black pebble values of the children of i arbitrarily. [2]

A *fractional pebbling* of T using p pebbles is any sequence of (fractional) pebbling moves on nodes of T which starts and ends with every node having pebble value 0, and at some point the root has black pebble value 1, and no configuration has more than p pebbles.

[2] It is easy to show that we can require, without increasing the pebbling cost, that every type (ii) move to increase $w(i)$ so that $b(i) + w(i) = 1$, and a type (iii) move to decrease $w(i)$ to 0, but we will not need to use that fact here.

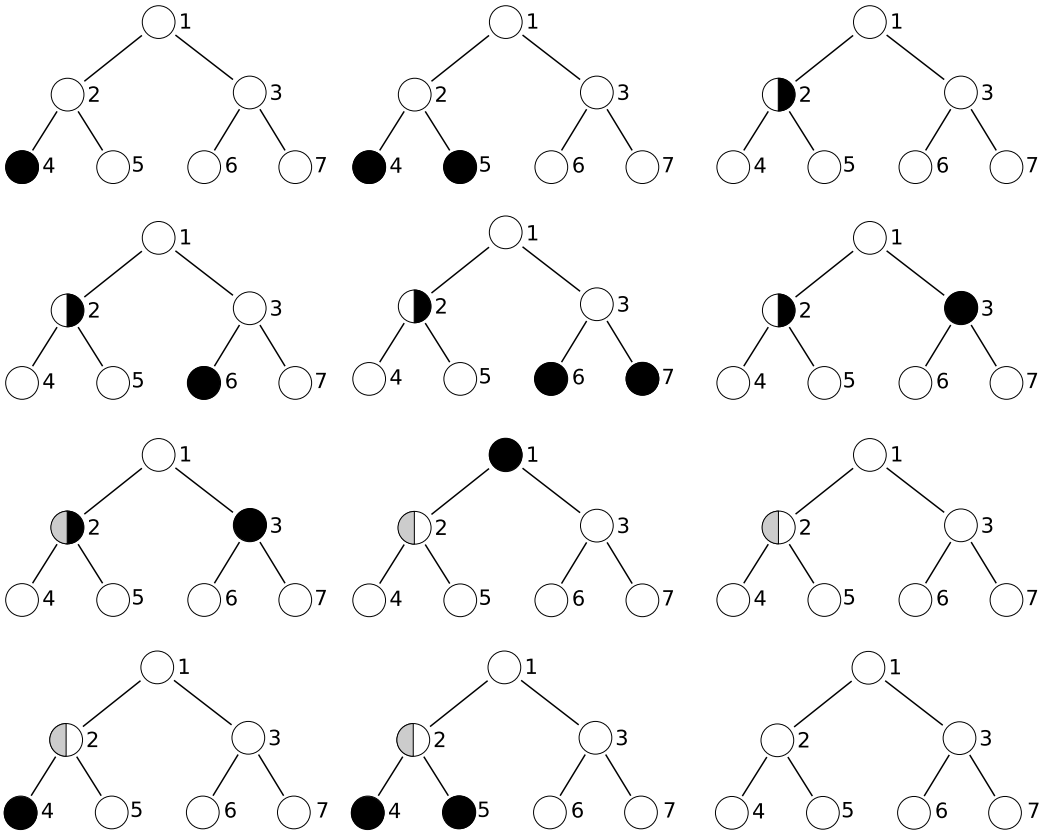


Figure 2: An optimal fractional pebbling sequence for the height 3 tree using 2.5 pebbles, all configurations included. The grey half circle means the *white* value of that node is .5, whereas unshaded area means absence of pebble value. So for example in the seventh configuration, node 2 has black value .5 and white value .5, node 3 has black value 1, and the remaining nodes all have black and white value 0.

A *whole black-white pebbling* of T is a fractional pebbling of T such that $b(i)$ and $w(i)$ take values in $\{0, 1\}$ for every node i and every configuration. A *black pebbling* is a black-white pebbling in which $w(i)$ is always 0.

Notice that rule (iii) does not quite treat black and white pebbles dually, since the pebble values of the children must each be 1 before any decrease of $w(i)$ is allowed. A true dual move would allow increasing the white pebble values of the children so they all have pebble value 1 while simultaneously decreasing $w(i)$. In other words, we allow black sliding moves, but disallow white sliding moves. The reason for this (as mentioned above) is that non-deterministic branching programs can simulate the former, but not the latter.

We use $\#\mathsf{Bpebbles}(T)$, $\#\mathsf{BWpebbles}(T)$, and $\#\mathsf{FRpebbles}(T)$ respectively to denote the minimum number of pebbles required to black pebble T, black-white pebble T, and fractional pebble T. Bounds for these values are given in [BCM+09a][3]. For example, $\#\mathsf{Bpebbles}(T^h) = h$, $\#\mathsf{BWpebbles}(T^h) = \lceil h/2 \rceil + 1$, and $\#\mathsf{FRpebbles}(T^h) \leq h/2 + 1$ (see [BCM+09c] for proofs). In particular $\#\mathsf{FRpebbles}(T^3) = 2.5$ (see Figure 2).

3 Thrifty Branching Programs and Pebbling

It is easy to show that the determinstic thrifty BPs we get from pebbling have $O(k^h)$ states, for all h. The next theorem shows there is a simple expression for the exact number of states. We do not know how to beat this upper bound for any k and h, even by an additive constant.

Theorem 1. *There are $(k+1)^h$ state deterministic thrifty BPs solving $FT^h(k)$.*

Proof. For $h = 1$ you have the start state that queries the single input variable l_1, with an edge out to each of the k output states.

For $h \geq 2$, we start with $k+1$ copies $B_0, B_1 ..., B_k$ of the BP that computes $FT^{h-1}(k)$. Here is the idea. We will use B_0 to compute the value of the left subtree, and for each $a \in [k]$ we use B_a to compute the value of the right subtree while remembering the value of the left subtree. At the level just before the output states, for each $\langle a, b \rangle \in [k]^2$ there is a state that queries $f_1(a, b)$.

Now for the formal definition. We will combine $B_0, B_1, \ldots, B_k$ in such a way that $B_1, ..., B_k$ are pairwise disjoint, and for all $a \in [k]$, B_0 and B_a intersect in exactly one state; namely, for all $a \in [k]$, if $q_{0,a}$ is the output state of B_0 labeled a, and q_a is the start state of B_a, then we remove $q_{0,a}$ and for each of the now-dangling B_0-edges e, we connect the free end of e to q_a.

Now change the state labels of B_0 so that whenever it queries $f_i(b_1, b_2)$ (resp. l_i) for some $i \in T^{h-1}$, it instead queries $f_{\sigma_2(i)}(b_1, b_2)$ (resp. $l_{\sigma_2(i)}$) where σ_2 maps node labels of T^{h-1} to node labels of the subtree of T^h rooted at node 2, in the obvious way. Similarly, for each a in $[k]$, change the state labels of B_a so that whenever it queries $f_i(b_1, b_2)$ (resp. l_i) for some $i \in T^{h-1}$, it instead queries the variable $f_{\sigma_3(i)}(b_1, b_2)$ (resp. $l_{\sigma_3(i)}$) where σ_3 is like σ_2 except it maps node labels of T^{h-1} to node labels of the subtree of T^h rooted at node 3.

Next, for each a, b in $[k]$, change the b labeled output state of B_a into a state that queries $f_1(a, b)$. Finally, add in the obvious way (there is only one way) k new output states that receive edges from

[3] And also for arbitrary degree d

the k^2 former output states of $B_1, ..., B_k$. That completes the definition of the BP the computes $FT^h(k)$. Its size $s(h,k)$ is given by

$$s(h,k) = (k+1)\, s(h-1,k) - k + k = (k+1)^h$$

Where the $-k$ is for the states $q_1, \ldots, q_k$ that get counted twice in the expression $(k+1)\, s(h-1,k)$ and the $+k$ is for the new output states. □

And indeed, we can show the bound is tight for height 2 (it is obviously tight for height 1). In Section 5 we conjecture that $(k+1)^3$ is tight for height 3 as well.

Theorem 2. *Every BP solving $FT^2(k)$ has at least $(k+1)^2$ states.*

Proof. There are at least k^2 states that query the root, since for all a, b there is at least one state that queries $f_1(a,b)$. There are k output states.

Let E^* be the inputs such that f_1 is $+ \bmod k$. Let Q^* be the states q such that q is the last leaf querying state on the computation path of some $I \in E^*$. We can show Q^* has size at least k. Let g be the function that maps each input in E^* to its last leaf querying state. Since $|E^*| = k^2$, it suffices to show that $|g^{-1}(q)| \leq k$ for every q in Q^*. Let $I_{a,b}$ be the unique input in E^* with $\langle l_2^I, l_3^I \rangle = \langle a, b \rangle$. Let $q \in Q^*$ be arbitrary. Consider the case that q queries l_3 – the other case is similar. Then it suffices to show that for every b, there is at most one a such that $I_{a,b}$ is in $g^{-1}(q)$. Just observe that if two inputs in E^* reach q then they have the same output state, and the label a' of the output state determines the unique a such that $a' = a + b \mod k$.

Now we want to show there is at least one state that queries a leaf and is not in Q^*. Since all the inputs in E^* agree on the f_i variables, there is a unique state q that is the first leaf querying state visited by any of them. Because $+ \bmod k$ is a quasigroup, every input in E^* must query l_2 and l_3 each at least once. So for every $I \in E^*$ there is a leaf querying state on the computation path of I after q that queries a leaf variable. Hence $q \notin Q^*$. That is $k^2 + k + k + 1 = (k+1)^2$ states total. □

Let the depth of a deterministic branching program be the maximum number of states visited by any input, with the output state included. The thrifty programs we get from pebbling have depth 2^h, and it is easy to show that depth 2^h is required, regardless of size; just note that Lemma 1 holds without the depth restriction. In fact, we can show thrifty programs are the *only* fastest determinstic BPs solving $BT^h(k)$.

Theorem 3. *For all h, k every deterministic branching program of depth at most 2^h computing $BT^h(k)$ (or $FT^h(k)$) is thrifty.*

Proof. Let E_0 be the inputs all of whose internal node functions are quasigroups, and E_1 the inputs that query each node exactly once.

Lemma 1. *Every input in E_0 queries each of its thrifty variables.*

Proof. Suppose $I \in E_0$ does not query its thrifty i variable. Let X be the thrifty i variable of I. For each $a \neq v_i^I$ there is an input I_a identical to I except $X^{I_a} = a$. Define the function $F_i^I : [k] \to [k]$ by

$$F_i^I := \begin{cases} \text{identity} & \text{if } i = 1 \\ f_j^I(F_j^I, v_{2j+1}^I) & \text{if } i = 2j \\ f_j^I(v_{2j}^I, F_j^I) & \text{if } i = 2j+1 \end{cases}$$

Since F_i^I is a permutation, the root values of the inputs I_a are all different from each other and from v^I. If $v_1^I = 1$ then let J be any of the I_a, and otherwise let J be the unique I_a such that $v_1^{I_a} = 1$. Then $J \in BT^h(k)$ iff $I \notin BT^h(k)$. But their computation paths are the same, a contradiction. □

Lemma 2. *Every input in E_0 is thrifty (queries only its thrifty variables).*

Proof. Because of the depth restriction, if an input queries each of its thrifty variables, then it is thrifty. So this lemma follows from Lemma 1. □

Lemma 3. *Every input in E_1 is thrifty.*

Proof. Suppose there is some I in E_1 that is not thrifty. For each node j, let X_j be the unique j variable that I queries. Since I is not thrifty, there is an internal node i such that X_i is not the thrifty f_i variable of I. Let i^* be such a node of minimum height. Since the computation path of I constrains only one value of each internal node function, we can choose an input $J \in E_0$ such that $X_j^I = X_j^J$ for all nodes j. J is thrifty by Lemma 2. In particular, X_{i^*} is the thrifty f_{i^*} variable of J. Since X_{i^*} is not the thrifty f_{i^*} variable of I, it must be that $v_{2i^*}^I \neq v_{2i^*}^J$ or $v_{2i^*+1}^I \neq v_{2i^*+1}^J$. Wlog assume it is the first case. By our choice of i^* and the assumption that I queries every node, we know I queries all its thrifty T_{2i} variables. Since the computation paths of I and J are identical, and J is thrifty, we have that I and J have the same thrifty T_{2i} variables. But then the only way to have $v_{2i^*}^I \neq v_{2i^*}^J$ is if there is a T_{2i} variable X that is thrifty for I (and so also for J) such that $X^I \neq X^J$. This contradicts the definition of J. □

Let $E_2 := E - E_1$. Fix I in E_2. Let P^I be the maximum length initial segment of the computation path of I such that there is some J in E_0 for which P^I is also an initial segment of the computation path of J. Fix such a J. Since I is not in E_1, there must be some i such that I does not query any i variable. So by Lemma 1, we know P^I cannot be the entire computation path of I (because then it would be the entire computation path of J). So the last state q_t of P^I cannot be its output state. Let q_{t+1} be the next state that I visits and e_t the edge I takes from q_t to q_{t+1}. Let X_t be the variable queried by q_t and $i_t := \mathsf{var}(X_t)$. There must be at least one J in E_0 that follows P^I (note the definition allows P^I to be a single state). Let q'_{t+1} be the next state visited by J. Since J disagrees with I on X_t, it must be that q_t is the first state on the computation path of I that queries X_t. On the other hand, there must have been a state q_s before q_t on P^I that queries an i_t variable X_s distinct from X_t; otherwise, there would be a J' in E_0 such that P^I, e_t, q_{t+1} is an initial segment of the computation path of J', contradicting the maximality of P^I. So now we know that J queries two distinct i_t variables. But J is in E_1 (since $E_0 \subseteq E_1$), so this contradicts Lemma 2. □

Now we give a tight lower bound for deterministic thrifty BPs. As discussed in section 2.1, this improves on an earlier result in [BCM+09a], which gives a lower bound of $\frac{1}{2}k^h$ for all h and all $k > \binom{2^h}{h-1}$.

Theorem 4. *For any h, k, every deterministic thrifty branching program solving $BT^h(k)$ has at least k^h states.*

Fix a deterministic thrifty BP B that solves $BT^h(k)$. Let E be the inputs to B. Let Vars be the set of k-valued input variables (so $|E| = k^{|\mathsf{Vars}|}$). Let Q be the states of B. If i is an internal node then the i variables are $f_i(a, b)$ for $a, b \in [k]$, and if i is a leaf node then there is just one i variable l_i. We sometimes say "f_i variable" just as an in-line reminder that i is an internal node. Let $\mathsf{var}(q)$ be the input variable that q queries. Let node be the function that maps each variable X to the node i such that X is an i variable, and each state q to $\mathsf{node}(\mathsf{var}(q))$. When it is clear from the context that q is on the computation path of I, we just say "q queries i" instead of "q queries the thrifty i variable of I".

Fix an input I, and let P be its computation path. We will choose n states on P as **critical states** for I, one for each node. Note that I must visit a state that queries the root (i.e. queries the thrifty root variable of I), since otherwise the branching program would make a mistake on an input J that is identical to I except $f_1^J(v_2^I, v_3^I) := k - f_1^I(v_2^I, v_3^I)$; hence $J \in BT_2^h(k)$ iff $I \notin BT_2^h(k)$. So, we can choose the root critical state for I to be the last state on P that queries the root. The remainder of the definition relies on the following small lemma:

Lemma 4. *For any J and internal node i, if J visits a state q that queries i, then for each child j of i, there is an earlier state on the computation path of J that queries j.*

Proof. Suppose otherwise, and wlog assume the previous statement is false for $j = 2i$. For every $a \neq v_{2i}^J$ there is an input J_a that is identical to J except $v_{2i}^{J_a} = a$. But the computation paths of J_a and J are identical up to q, so J_a queries a variable $f_i(a, b)$ such that $b = v_{2i+1}^{J_a}$ and $a \neq v_{2i}^{J_a}$, which contradicts the thrifty assumption. □

Now we can complete the definition of the critical states of I. For i an internal node, if q is the node i critical state for I then the node $2i$ (resp. $2i + 1$) critical state for I is the last state on P before q that queries $2i$ (resp. $2i + 1$).

Now we assign a pebbling sequence to each state on P, such that the set of pebbled nodes in each configuration is a minimal cut of the tree or a subset of some minimal cut (and once it becomes a minimal cut, it remains so), and any two adjacent configurations are either identical, or else the later one follows from the earlier one by a valid pebbling move. This assignment can be described inductively by starting with the last state on P and working backwards. Note that implicitly we will be using the following fact:

Fact 1. *For any input I, if j is a descendant of i then the node j critical state for I occurs earlier on the computation path of I than the node i critical state for I.*

The pebbling configuration for the output state has just a black pebble on the root. Assume we have defined the pebbling configurations for q and every state following q on P, and let q' be the

state before q on P. If q' is not critical, then we make its pebbling configuration be the same as that of q. If q' is critical then it must query a node i that is pebbled in q. The pebbling configuration for q' is obtained from the configuration for q by removing the pebble from i and adding pebbles to $2i$ and $2i+1$ (if i is an internal node - otherwise you only remove the pebble from i).

In the above definition of the pebbling configurations, consider the first critical state we define that queries a height 2 node (working backwards – so the first critical state we define queries the root). We use r^I to denote this state and call it the **supercritical state** of I. Since the pebbling configurations up to r^I (again, working backwards) are minimal cuts of the tree, and the children of $\mathsf{node}(r^I)$ are included, it is not hard to see that there must be at least h pebbled nodes. We refer to these nodes as the **bottleneck nodes** of I. Define the **bottleneck path** of $r \in R$ to be the path from $\mathsf{node}(r)$ to the root. The bottleneck path of $I \in E$ is the bottleneck path of r^I. This is the main property of the pebbling sequences that we need:

Fact 2. *For any input I, if non-root node i with parent j is pebbled at a state q on P^I, then the node j critical state q' of I occurs later on P^I, and there is no state (critical or otherwise) between q and q' on P^I that queries i.*

Let R be the states that are supercritical for at least one input. Let E_r be the inputs with supercritical state r. Now we can state the main lemma.

Lemma 5. *For every $r \in R$, there is an injective function from E_r to $[k]^{|\mathsf{Vars}|-h}$.*

The lemma gives us that $|E_r| \leq k^{|\mathsf{Vars}|-h}$ for every $r \in R$. Since $\{E_r\}_{r\in R}$ is a partition of E, there must be at least $|E|/k^{|\mathsf{Vars}|-h} = k^h$ sets in the partition, i.e. there must be at least k^h supercritical states. So the theorem follows from the lemma.

Fix $r \in R$ and let $D := E_r$. Let $i_{\mathsf{sc}} := \mathsf{node}(r)$. Since r is thrifty for every I in D, there are values $v^D_{2i_{\mathsf{sc}}}$ and $v^D_{2i_{\mathsf{sc}}+1}$ such that $v^I_{2i_{\mathsf{sc}}} = v^D_{2i_{\mathsf{sc}}}$ and $v^I_{2i_{\mathsf{sc}}+1} = v^D_{2i_{\mathsf{sc}}+1}$ for every I in D. We are going to define a procedure INTERADV that takes as input a $[k]$-string (the advice), tries to interpret it as the code of an input in D, and when successful outputs that input. We want to show that for every $I \in D$ we can choose $\mathsf{adv}^I \in [k]^{|\mathsf{Vars}|-h}$ such that $\textsc{InterAdv}(\mathsf{adv}^I)\downarrow = I$. Of course, choosing adv^I for each I yields the injective function required to prove the lemma.

During the execution of INTERADV we maintain a current state q, a partial function v^* from nodes to $[k]$, and a set of nodes U_{L}. Once we have added a node to U_{L}, we never remove it, and once we have added $v^*(i) := a$ to the definition of v^*, we never change $v^*(i)$. We have reached q by following a *consistent partial computation path* starting from r, meaning there is at least one input in D that visits exactly the states and edges that we visited between r and q. So initially $q = r$. Intuitively, $v^*(i)\downarrow = a$ for some a when we have "committed" to interpreting the advice we have read so-far as being the initial segment of *some* complete advice string adv^I for an input I with $v^I_i = a$. Initially v^* is undefined everywhere. As the procedure goes on, we may often have to use an element of the advice in order to set a value of v^*; however, by exploiting the properties of the critical state sequences, for each $I \in D$, when given the complete advice adv^I for I there will be at least h nodes U^I_{L} that we "learn" without directly using the advice. Such an oppurtunity arises when we visit a state that queries some variable $f_i(b_1, b_2)$ and we have not yet committed to a value for at least one of $v^*(2i)$ or $v^*(2i+1)$ (if both then, we learn two nodes). When this happens, we

add that child or children of i to U_{L} (the L stands for "learned"). So initially U_{L} is empty. There is a loop in the procedure INTERADV that iterates until $|U_{\mathsf{L}}| = h$. Note that the children of i_{sc} will be learned immediately. Let $v^*(D)$ be the inputs in D consistent with v^*, i.e. $I \in v^*(D)$ iff $I \in D$ and $v_i^I = v^*(i)$ for every $i \in \mathsf{Dom}(v^*)$.

Following is the complete pseudocode for INTERADV. We also state the most-important of the invariants that are maintained.

Procedure INTERADV($\vec{a} \in [k]^*$):

1: $q := r$, $U_{\mathsf{L}} := \emptyset$, $v^* :=$ undefined everywhere.
2: **Loop Invariant:** If N elements of $\vec{a}$ have been used, then $|\mathsf{Dom}(v^*)| = N + |U_{\mathsf{L}}|$.
3: **while** $|U_{\mathsf{L}}| < h$ **do**
4: $\quad i := \mathsf{node}(q)$
5: $\quad$ **if** i is an internal node and $2i \notin \mathsf{Dom}(v^*)$ or $2i+1 \notin \mathsf{Dom}(v^*)$ **then**
6: $\quad\quad$ let b_1, b_2 be such that $\mathsf{var}(q) = f_i(b_1, b_2)$.
7: $\quad\quad$ **if** $2i \notin \mathsf{Dom}(v^*)$ **then**
8: $\quad\quad\quad$ $v^*(2i) := b_1$ and $U_{\mathsf{L}} := U_{\mathsf{L}} + 2i$.
9: $\quad\quad$ **end if**
10: $\quad\quad$ **if** $2i+1 \notin \mathsf{Dom}(v^*)$ and $|U_{\mathsf{L}}| < h$ **then**
11: $\quad\quad\quad$ $v^*(2i+1) := b_2$ and $U_{\mathsf{L}} := U_{\mathsf{L}} + (2i+1)$.
12: $\quad\quad$ **end if**
13: $\quad$ **end if**
14: $\quad$ **if** $i \notin \mathsf{Dom}(v^*)$ **then**
15: $\quad\quad$ let a be the next unused element of $\vec{a}$.
16: $\quad\quad$ $v^*(i) := a$.
17: $\quad$ **end if**
18: $\quad$ $q :=$ the state reached by taking the edge out of q labeled $v^*(i)$.
19: **end while**
20: let $\vec{b}$ be the next $|\mathsf{Vars}| - |\mathsf{Dom}(v^*)|$ unused elements of v^*.
21: let $I_1, \ldots, I_{|v^*(D)|}$ be the inputs in $v^*(D)$ sorted according to some globally fixed order on E.
22: if $\vec{b}$ is the t-largest string in the lexiocgraphical ordering of $[k]^{|\mathsf{Vars}|-|\mathsf{Dom}(v^*)|}$, and $t \leq |v^*(D)|$, then return I_t.[4]

If the loop finishes, then there are at most $|E|/|\mathsf{Dom}(v^*)| = k^{|\mathsf{Vars}|-|\mathsf{Dom}(v^*)|}$ inputs in $v^*(D)$. So for each of the inputs I enumerated on line 21, there is a way of setting $\vec{a}$ so that I will be chosen on line 22.

Recall we are trying to show that for every I in D there is a string $\mathsf{adv}^I \in [k]^{|\mathsf{Vars}|-h}$ such that INTERADV$(\vec{a})\downarrow = I$. This is easy to see under the assumption that there is such a string that makes the loop finish while maintaining the loop invariant; since the loop invariant ensures we have used $|\mathsf{Dom}(v^*)| - h$ elements of advice when we reach line 20, and since line 20 is the last time when the advice is used, in all we use at most $|\mathsf{Vars}| - h$ elements of advice. To remove that assumption, first observe that for each I, we can set the advice to some adv^I so that $I \in g(D)$ is maintained

[4] See after this code for argument that $|v^*(D)| \leq k^{|\mathsf{Vars}|-|\mathsf{Dom}(v^*)|}$.

when INTERADV is run on $\vec{a}^I$. Moreover, for that adv^I, we will never use an element of advice to set the value of a bottleneck node of I, and I has at least h bottleneck nodes. Note, however, that this does not necessarily imply that U_{L}^I (the h nodes U_{L} we obtain when running INTERADV on adv^I) is a subset of the bottleneck nodes of I. Finally, note that we are of course implicitly using the fact that no advice elements are "wasted"; each is used to set a different node value.

Corollary 1. *For any h, k, every deterministic thrifty branching program solving $BT_2^h(k)$ has at least $\sum_{2 \le l \le h} k^l$ states.*

Proof. The previous theorem only counts states that query height 2 nodes. The same proof is easily adapted to show there are at least k^{h-l+2} states that query height l nodes, for $l = 2, \ldots, h$. Those $h - 1$ state sets are disjoint, so we can sum the bounds. □

4 Main Results

4.1 Fractional Pebbling Lower Bound

The proof of Theorem 5 proceeds by reducing the problem of proving lower bounds on the fractional pebbling cost for balanced binary trees, to the problem of proving lower bounds on the black-white pebbling costs for a family of DAGs. In doing so, we are essentially discretizing the fractional pebbling problem; the main construction has a parameter c that determines how many nodes in the dag are used to "simulate" each node in the tree. We will use the next lemma (due to S. Cook) to conclude that we can always make c large enough that we don't "lose anything".

Lemma 6. *For every finite DAG there is an optimal fractional B/W pebbling in which all pebble values are rational numbers. (This result is robust independent of various definitions of pebbling; for example with or without sliding moves, and whether or not we require the root to end up pebbled.)*

Proof. Consider an optimal B/W fractional pebbling algorithm. Let the variables $b_{v,t}$ and $w_{v,t}$ stand for the black and white pebble values of node v at step t of the algorithm.

Claim: We can define a set of linear inequalities with 0 - 1 coefficients which suffice to ensure that the pebbling is legal.

For example, all variables are non-negative, $b_{v,t} + w_{b,t} \le 1$, initially all variables are 0, and finally the nodes have the values that we want, node values remain the same on steps in which nothing is added or subtracted, and if the black value of a node is increased at a step then all its children must be 1 in the previous step, etc.

Now let p be a new variable representing the maximum pebble value of the algorithm. We add an inequality for each step t that says the sum of all pebble values at step t is at most p.

Any solution to the linear programming problem:

Minimize p subject to all of the above inequalities

gives an optimal pebbling algorithm for the graph. But Every LP program with rational coefficients has a rational optimal solution (if it has any optimal solution). □

Now we are ready to prove the lower bound. We know this bound is not tight for heights at most 4. This is easy to see for height 2 (the bound should be d, but the theorem gives $d/2-1$), and proofs of the tight bounds for heights 3 and 4 are given in [BCM+09c].

Theorem 5. *The fractional pebbling cost for the degree d, height h tree is at least $(d-1)h/2-d/2$.*

Proof. The high-level strategy for the proof is as follows. Given d and h, we transform the tree T_d^h into a DAG $G_{d,h}$ such that a lower bound on $\#\mathsf{BWpebbles}(G_{d,h})$ gives a lower bound for $\#\mathsf{FRpebbles}(T_d^h)$. To analyze $\#\mathsf{BWpebbles}(G_{d,h})$, we use a result of Klawe [Kla85], who shows that for a DAG G that satisfies a certain "niceness" property, $\#\mathsf{BWpebbles}(G)$ can be given in terms of $\#\mathsf{Bpebbles}(G)$ (and the relationship is tight to within a constant less than one). The black pebbling cost is typically easier to analyze. In our case, $G_{d,h}$ does not satisfy the niceness property as-is, but just by removing some edges from $G_{d,h}$, we get a new DAG $G'_{d,h}$ which is nice. We then show how to exactly compute $\#\mathsf{Bpebbles}(G'_{d,h})$ which yields a lower bound on $\#\mathsf{BWpebbles}(G_{d,h})$, and hence on $\#\mathsf{FRpebbles}(T_d^h)$.

We first motivate the construction $G_{d,h}$ and show that the whole black-white pebbling number of $G_{d,h}$ is related to the fractional pebbling number of T_d^h.

We first use Lemma 6 to "discretize" the fractional pebble game. The following are the rules for the discretized game, where c is a parameter:

- For any node v, decrease $b(v)$ or increase $w(v)$ by $1/c$.
- For any node v, including leaf nodes, if all the children of v have value 1, then increase $b(v)$ or decrease $w(v)$ by $1/c$.

By Lemma 6, we can assume all pebble values are rational, and if we choose c large enough it is not a restriction that pebble values can only be changed by $1/c$. Since sliding moves are not allowed, the pebbling cost for this game is at most one more than the cost of fractional pebbling with black sliding moves.

Now we show how to construct $G_{d,h}$ (for an example, see figure 3). We will split up each node of T_d^h into c nodes, so that the discretized game corresponds to the whole black-white pebble game on the new graph. Specifically, the cost of the whole black-white pebble game on the new graph will be exactly c times the cost of the discretized game on T_d^h.

In place of each node v of T_d^h, $G_{d,h}$ has c nodes $v[1], \ldots, v[c]$; having c' of the $v[i]$ pebbled simulates v having value c'/c. In place of each edge (u, v) of T_d^h is a copy of the complete bipartite graph (U, V), where U contains nodes $u[1] \ldots u[c]$ and V contains nodes $v[1] \ldots v[c]$. If u was a parent of v in the tree, then all the edges go from V to U in the corresponding complete bipartite graph. Finally, a new "root" is added at height $h+1$ with edges from each of the c nodes at height h[5]. So every node at height $h-1$ and lower has c parents, and every internal node except for the root has dc children.

[5]The reason for this is quite technical: Klawe's definition of pebbling is slightly different from ours in that it requires that the root remain pebbled. Adding a new root forces there to be a time when all c of the height h nodes, which represent the root of T_d^h, are pebbled. Adding one more pebble to $G_{d,h}$ changes the relationship between the cost of pebbling T_d^h and the cost of pebbling $G_{d,h}$ by a negligible amount.

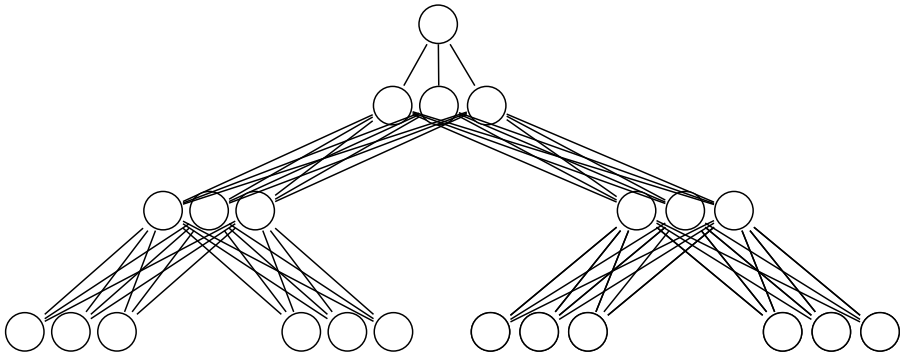


Figure 3: Illustration to accompany the definition of $G_{d,h}$. This is $G_{2,3}$ with parameter $c = 3$

To lower bound $\#\mathsf{BWpebbles}(G_{d,h})$, we will use Klawe's result [Kla85]. Klawe showed that for "nice" graphs G, the black-white pebbling cost of G (with black and white sliding moves) is at least $\lfloor \#\mathsf{Bpebbles}/2 \rfloor + 1$. Of course, the black-white pebbling cost without sliding moves is at least the cost with them. We define what it means for a graph to be nice in Klawe's sense.

Definition 5. A DAG G is nice if the following conditions hold:

1. If u_1, u_2 and u are nodes of G such that u_1 and u_2 are children of u (i.e., there are edges from u_1 and u_2 to u), then the cost of black pebbling u_1 is equal to the cost of black pebbling u_2
2. If u_1 and u_2 are children of u, then there is no path from u_1 to u_2 or from u_2 to u_1.
3. If $u, u_1, \ldots, u_m$ are nodes none of which has a path to another, then there are node-disjoint paths $P_1, \ldots, P_m$ such that P_i is a path from a leaf (a node with in-degree 0) to u_i and there is no path between u and any node in P_i.

$G_{d,h}$ is not nice in Klawe's sense. We will delete some edges from $G_{d,h}$ to produce a nice graph $G'_{d,h}$ and we will analyze $\#\mathsf{Bpebbles}(G'_{d,h})$. Note that a lower bound on $\#\mathsf{BWpebbles}(G'_{d,h})$ is also a lower bound on $\#\mathsf{BWpebbles}(G_{d,h})$.

The following definition will help in explaining the construction of $G'_{d,h}$ as well as for specifying and proving properties of certain paths.

Definition 6. For $u \in G_{d,h}$, let $T_d^h(u)$ be the node in T_d^h such that $T_d^h(u)[i] = u$ for some $i \leq c$. For $v, v' \in T_d^h$, we say $v < v'$ if v is visited before v' in an inorder traversal of T_d^h. For $u, u' \in G_{d,h}$, we say $u < u'$ if $T_d^h(u) < T_d^h(u')$ or if for some $v \in T_d^h$, $u = v[i]$, $u' = v[j]$, and $i < j$.

$G'_{d,h}$ is obtained from $G_{d,h}$ by removing $c-1$ edges from each internal node except the root, as follows (for an example, see figure 4). For each internal node v of T, consider the corresponding nodes $v[1], v[2], \ldots, v[c]$ of $G_{d,h}$. Remove the edges from $v[i]$ to its $i-1$ smallest and $c-i$ largest children. So in the end each internal node except the root has $c(d-1)+1$ children.

We first analyze $\#\mathsf{Bpebbles}(G'_{d,h})$ and then show that it is nice. We show that $\#\mathsf{Bpebbles}(G'_{d,h}) = c[(d-1)(h-1)+1]$. Note that an upper bound of $c[(d-1)(h-1)+1]$ is attained using a simple recursive algorithm similar to that used for the binary tree.

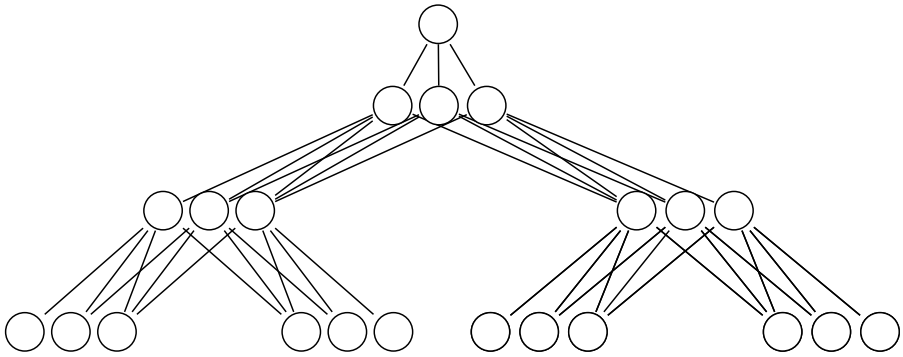


Figure 4: Illustration to accompany the definition of $G'_{d,h}$. This is $G'_{2,3}$ with parameter $c = 3$

For the lower bound, consider the earliest time t when all paths from a leaf to the root are blocked. Figure 5 is an example of the type of pebbling configuration that we are about to analyze. The last pebble placed must have been placed at a leaf, since otherwise $t-1$ would be an earlier time when all paths from a leaf to the root are blocked. Let P be the newly-blocked path from a leaf to the root. Consider the set $S = \{u \in G'_{d,h} \mid u \notin P \text{ and } u \text{ is a child of a node in } P\}$ of size $c(d-1)(h-1) + (c-1) = c[(d-1)(h-1)+1] - 1$ (the $c-1$ is contributed by nodes at height h). We will give a set of mutually node-disjoint paths $\{P_u\}_{u \in S}$ such that P_u is a path from a leaf to u and P_u does not intersect P. At time $t-1$, there must be at least one pebble on each P_u, since otherwise there would still be an open path from a leaf to the root at time t. Also counting the leaf node that is pebbled at t gives c[(d-1)(h-1) + 1] pebbles.

Definition 7. The left-most (right-most) path to u is the unique path ending at u determined by choosing the smallest (largest) child at every level.

Definition 8. $P(l)$ is the node of path P at height l, if it exists.

For each $u \in S$ at height l, if u is less than (greater than) $P(l)$ then make P_u the left-most (right-most) path to u. Now we need to show that the paths $\{P_u\}_{u \in S} \cup \{P\}$ are disjoint. The following fact is clear from the definition of $G'_{d,h}$.

Lemma 7. *For any $u, u' \in G'_{d,h}$, if $u < u'$ then the smallest child of u is not a child of u', and the largest child of u' is not a child of u.*

First we show that P_u and P are disjoint. The following lemma will help now and in the proof that $G'_{d,h}$ is nice.

Lemma 8. *For $u, v \in G'_{d,h}$ with $u < v$, if there is no path from u to v or from v to u then the left-most path to u does not intersect any path to v from a leaf, and the right-most path to v does not intersect any path to u from a leaf.*

Proof. Suppose otherwise and let P'_u be the left-most path to u, and P'_v a path to v that intersects P'_u. Since there is no path between u and v, there is a height l, one greater than the height where the two paths first intersect, such that $P'_u(l), P'_v(l)$ are defined and $P'_u(l) < P'_v(l)$. But then from

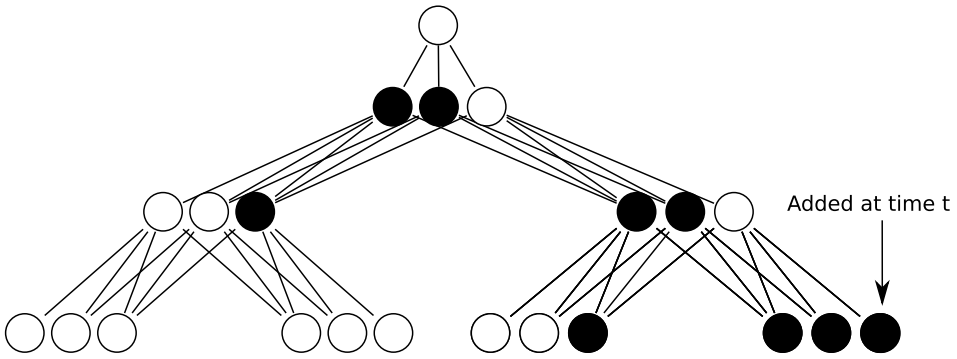


Figure 5: A possible black pebbling bottleneck of $G'_{2,3}$, with $c = 3$

Lemma 7 $P'_u(l-1) \neq P'_v(l-1)$, a contradiction. The proof for the second part of the lemma is similar. □

That P_u and P are disjoint follows from using Lemma 8 on u and the sibling of u in P.

Next we show that for distinct $u, u' \in S$, P_u does not contain u'. Suppose it does. Assume P_u is the left-most path to u (the other case is similar). Since $u \neq u'$, there must be a height l such that $P_u(l)$ is defined and $P_u(l-1) = u'$. From the definition of S, we know $P(l)$ is also a parent of u'. From the construction of P_u, since we assumed P_u is the left-most path to u, it must be that $P_u(l) < P(l)$. But then Lemma 7 tells us that u' cannot be a child of $P(l)$, a contradiction.

The proof that P_u and $P_{u'}$ do not intersect is by contradiction. Assuming that there are $u, u' \in S$ such that P_u and $P_{u'}$ intersect, there is a height l, one greater than the height where they first intersect, such that $P_u(l) \neq P_{u'}(l)$. Note that P_u and $P_{u'}$ are both left-most paths or both right-most paths, since otherwise in order for them to intersect they would need to cross P. But then from Lemma 7 $P_u(l-1) \neq P_{u'}(l-1)$, a contradiction.

See Figure 5 for an example of a bottleneck of the specified structure for $G'_{d,h}$ corresponding to the height 3 binary tree, with $c = 3$:

The last step is to prove that $G'_{d,h}$ is nice. There are three properties specified in Definition 5. Property 2 is obviously satisfied. For property 1, the argument used to give the black pebbling lower bound of $c[(d-1)(h-1)+1]$ can be used to give a black pebbling lower bound of $c(d-1)(l-1)+1$ for any node at height $l \leq h$ (the 1 is for the last node pebbled, and recall the root is at height $h+1$), and that bound is tight. For property 3, choose P_i to be the left-most (right-most) path from u_i if u_i is less than (greater than) u. Then use Lemma 8 on each pair of nodes in $\{u, u_1, \ldots, u_m\}$.

Since $\#\mathsf{Bpebbles}(G'_{d,h}) = c[(d-1)(h-1)+1]$, we have

$$\#\mathsf{BWpebbles}(G_{d,h}) \geq \#\mathsf{BWpebbles}(G'_{d,h}) \geq c[(d-1)(h-1)+1]/2$$

and thus that the pebbling cost for the discretized game on T_d^h is at least $(d-1)(h-1)/2 + .5$, which implies $\#\mathsf{FRpebbles}(T_d^h) \geq (d-1)(h-1)/2 - .5$. □

4.2 Less-Thrifty Branching Programs

4.2.1 Thrifty BPs with Wrong-Wrong Queries

A variable $f_i(a, b)$ is **wrong-wrong** for input I iff $a \neq v^I_{2i}$ and $b \neq v^I_{2i+1}$. The next theorem shows that querying wrong-wrong variables does not help.

Theorem 6. *For any $h, k \geq 2$, if B is a deterministic BP that solves $BT^h(k)$ such that each input only queries variables that are thrifty or wrong-wrong for it, then B has at least k^h states.*

Proof. We use the definitions and conventions introduced in the first paragraph of the proof of Theorem 4. The proof of the following lemma is similar to that of Lemma 4 (page 10)[6]:

Lemma 9. *For any J and internal node i, there is at least one state q on the computation path of J that queries the thrifty i variable of J, and for every such q, for each child j of i, there is a state on the computation path of J before q that queries the thrifty j variable of J.*

Recall that for the thrifty lower bound, to each input we assigned one "critical state" for each node, and a pebbling configuration to each critical state, such that the n pebbling configurations made a valid pebbling sequence. This was so even if the thrifty branching program was constructed based on a pebbling sequence of length greater than n. Now we will not be selecting critical states, and we will assign pebbling sequences with length possibly greater than n. It may be helpful to note that this way of assigning pebbling sequences will have the following property:

Remark Let S be a complete pebbling sequence for T^h such that the root is pebbled only once, and a pebble is removed from a non-root node i only during a move that places a pebble on the parent of i. For any k, if $B_{S,k}$ is the thrifty deterministic BP for solving $FT^h(k)$ that implements S in the natural way[7], then for every input I to $B_{S,k}$, we will assign pebbling sequence S to I.

In the end, this will result in a cleaner proof; in particular, we will be able to say that when we interpret the advice for I, every node that gets "learned" is a bottleneck node of I (see Fact 3).

We define the pebbling sequence for $I \in E$ by following the computation path of I from beginning to end, associating the t-th thrifty state q_t visited by I with the t-th pebbling configuration C_t, such that C_{t+1} is either identical to C_t or follows from C_t by applying a valid pebbling move. Let $q_1, \ldots, q_{t^*}$ be the thrifty states on the computation path of I, up to the first state q_{t^*} that queries the thrifty root variable of I. Note that q_1 must query a leaf by Lemma 9. We associate q_1 with the empty configuration C_1.

Assume we have defined the configurations $C_1, \ldots, C_t$ for the first $t \geq 1$ thrifty states, and assume $C_1, \ldots, C_t$ is a valid sequence of configurations (where adjacent identical configurations are allowed), but neither it nor any prefix of it is a complete pebbling sequence. We also maintain that for all $t' \leq t$, if $\mathsf{node}(q_{t'})$ is internal, then its children are pebbled in $C_{t'}$ and it is not. Let

[6] Also this lemma is proved in a more-general context on page 25

[7] We are talking about a particular family of thrifty BPs $\{B_{S,k}\}$, without taking the time to give a precise definition. $B_{S,k}$ has $|S|$ non-output layers (where $|S|$ is the number of moves in S), and if a pebble is placed on i in the l-th move of S when there are p pebbles on the tree, then there are k^p states in layer l of $B_{S,k}$, all of which query a node i variable.

$i := \mathsf{node}(q_t)$. By the I.H. i is not pebbled in C_t. We define C_{t+1} by saying how to obtain it by modifying C_t:

1. If i is the root, then clearly $t+1 = t^*$, and by the I.H. nodes 2 and 3 are pebbled. Put a pebble on the root and remove the pebbles from nodes 2 and 3. This completes the definition of the pebbling sequence for I.
2. If i is a non-root internal node, then by the I.H. both children of i are pebbled. For each child j of i: if there is a state q' after q_t that queries the thrifty i variable of I, and no state between q_t and q' that queries the thrifty j variable of I, then leave the pebble on j, and otherwise remove it.
3. If i is not the root. then place a pebble on i iff there is a state q' after q_t that queries the thrifty $\mathsf{par}(i)$ variable of I and there is no state between q_t and q' that queries the thrifty i variable of I.

As before, we define the supercritical state r^I of I to be the first thrifty state on the computation of I whose associated pebbling configuration (the **bottleneck nodes** of I) has at least one node blocking every path from the root to a leaf. Let R be the states that are supercritical for at least one input, and for each $r \in R$ let E_r be the inputs with supercritical state r. As before, using the argument for the black pebbling lower bound of h pebbles for T^h, we get that each $r \in R$ queries a height two node – call it i^r_{sc}. For $I \in E_r$ we say i^r_{sc} is the supercritical node for I. The definition of the **bottleneck path** BnPath_r for $r \in R$ has not changed: it is the path from r to the root. We mentioned earlier that every node we "learn" for an input I is a bottleneck node of I. This is due to the next fact. For any I and q on the computation path of I, let $\mathsf{Path}^I(q)$ be the part of the computation path of I starting with q.

Fact 3. *i is a bottleneck node of $I \in E_r$* **iff** *it is not in BnPath_r and there is a state $q \in \mathsf{Path}^I(r)$ that queries the thrifty $\mathsf{par}(i)$ variable of i and no state before q in $\mathsf{Path}^I(r)$ that queries the thrifty i variable of I.*

It will be convenient to have named the following four sets of nodes:

Definition 9 ($\mathsf{SiblBnPath}_r, \mathsf{RightPath}_i, \mathsf{Learnable}_r, \mathsf{Learnable}^*_r$)**.**

- $\mathsf{SiblBnPath}_r$ is the set of nodes that are the sibling of a node in BnPath_r.
- For $i \in \mathsf{SiblBnPath}_r$, $\mathsf{RightPath}_i$ is the path from i to the right-most leaf under i (when the tree is drawn in the canonical way).
- $\mathsf{Learnable}_r$ is the set of nodes $\{2i^r_{\mathsf{sc}}, 2i^r_{\mathsf{sc}}+1\} + \bigcup_{i \in \mathsf{SiblBnPath}_r} \mathsf{RightPath}_i$, i.e. the nodes not on the bottleneck path that are the descendent of a node on the bottleneck path.
- $\mathsf{Learnable}^*_r := \mathsf{Learnable}_r - \{2i^r_{\mathsf{sc}}, 2i^r_{\mathsf{sc}}+1\}$.

It is not hard to see that every $I \in E_r$ has at least one bottleneck node in $\mathsf{RightPath}_j$ for each of the $h-2$ nodes $j \in \mathsf{SiblBnPath}_r$ (this observation is used in the black pebbling lower bound argument mentioned above).

Let G be the set of partial functions from Vars to $[k]$. At least when $k=2$ these are commonly called *restrictions* (of $BT^h(k)$), so we will refer to them as restrictions. For $g \in G$ and $D \subseteq E$ we write $g(D)$ for the inputs in D consistent with g – i.e. $g(D) := \{I \in D \mid \forall X \in \mathsf{Dom}(g).\ X^I =$

$g(X)\}$. It will be convenient to further partition the sets E_r by fixing some of the variables initially. This finer partitioning appears in the statement of the main lemma:

Lemma 10 (Main Lemma)**.** *For some integer M, for every supercritical state $r \in R$, there is a set of restrictions G^r_{init} of size at most $k^{|\mathsf{Vars}|-M}$ such that $\{g_{\mathsf{init}}(E_r)\}_{g_{\mathsf{init}} \in G^r_{\mathsf{init}}}$ is a partition of E_r and for every g_{init} in G^r_{init}, there is an injective function from $g_{\mathsf{init}}(E_r)$ to $[k]^{M-h}$.*

Let us see why the theorem follows from the lemma. Since $\{g_{\mathsf{init}}(E_r)\}_{g_{\mathsf{init}} \in G^r_{\mathsf{init}}}$ is a partition of E_r, and G^r_{init} has size at most $k^{|\mathsf{Vars}|-M}$, there must be some $g^*_{\mathsf{init}} \in G^r_{\mathsf{init}}$ such that $g^*_{\mathsf{init}}(E_r)$ has size at least $|E_r|/k^{|\mathsf{Vars}|-M}$. On the other hand, from the lemma we get that *every* set $g_{\mathsf{init}}(E_r)$ in the partition has size at most k^{M-h}. Hence

$$|E_r|/k^{|\mathsf{Vars}|-M} \leq |g^*_{\mathsf{init}}(E_r)| \leq k^{M-h}$$

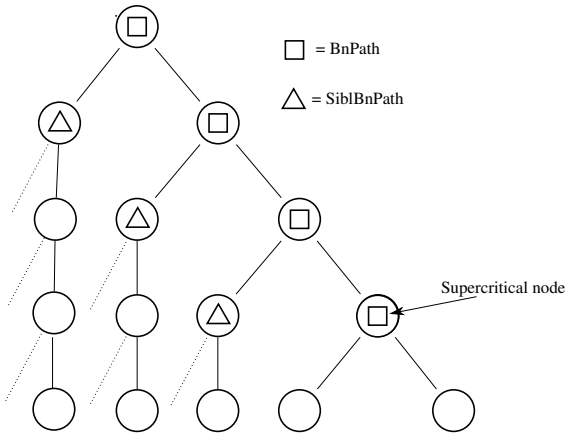
Rearranging gives $|E_r| \leq k^{|\mathsf{Vars}|}/k^h = |E|/k^h$, and this holds for all $r \in R$. Since $\{E_r\}_{r \in R}$ is a partition of E, we get that R must have size at least k^h.

Proof of Main Lemma

We use T to refer to the height h balanced binary tree, or to the set of its nodes. We use T_i to refer to the subtree of T rooted at node i, or to its nodes. For U a set of nodes, $\mathsf{Vars}(U)$ is the set of input variables corresponding to the nodes in U – i.e $\mathsf{Vars}(U) := \{X \in \mathsf{Vars} \mid X = l_i \text{ or } X = f_i(a,b) \text{ for some } i \in U \text{ and } a, b \in [k]\}$. For $D \subseteq E$ there is a partial function $i \mapsto v^D_i$ from T to $[k]$ such that $v^D_i\downarrow = a$ iff $v^I_i = a$ for every I in D. Similarly there is a partial function $X \mapsto v^D_i$ from Vars to $[k]$ such that $X^D\downarrow = a$ iff $X^I = a$ for every I in D.

The constant M mentioned in the theorem is $k(h-1)(h-2)/2 + k^2(h-1) + h$, but we are just writing that expression here for clarity; we will not be reasoning about it. For each $r \in R$, we are going to define a set G^r_{init} of at most $k^{|\mathsf{Vars}|-M}$ restrictions where each $g_{\mathsf{init}} \in G^r_{\mathsf{init}}$ is defined on some set of $|\mathsf{Vars}| - M$ variables. Before giving the precise definition of the partition, let us see where the expression for M comes from. For $(h-1)(h-2)/2 = (h-2) + (h-3) + ... + 1$ internal nodes i we will fix all but k of the k^2 variables that define the corresponding function f_i. For each of the $h-1$ nodes on the bottleneck path BnPath_r, we will not fix any of the k^2 variables that define the corresponding function. Lastly, there will be h unfixed leaf variables.

Let U^r_{fixed} be all the nodes except $\mathsf{Learnable}_r + \mathsf{BnPath}_r$. In the following drawing, which depicts the construction for the height 5 tree when $i^r_{\mathsf{sc}} = 15$ is the right-most height 2 node, the pruned nodes (the nodes in the subtrees that would be at the ends of the dashed lines) are U^r_{fixed} and the unmarked nodes plus the $\triangle$-marked nodes are $\mathsf{Learnable}_r$. The $\square$-marked nodes are BnPath_r and will have no fixed variables. The $\triangle$-marked nodes are $\mathsf{SiblBnPath}_r$ and will have $k^2 - k$ fixed variables.



Let $\hat{G}_r$ be all the restrictions g with domain $\mathsf{Vars}(U^r_{\mathsf{fixed}})$. For every $g \in \hat{G}_r$, for every internal node i in $\mathsf{Learnable}_r$, we have that $v_{2i}^{g(E_r)}$ is defined since g is defined for every T_{2i} variable. For each $g \in \hat{G}_r$ let $G_{r,g}$ be the set of extensions g' of g such that for all internal nodes i in $\mathsf{Learnable}_r$, for all $a \neq v_{2i}^{g(E_r)}$ and all b, g' is defined on $f_i(a,b)$, and $g'(E_r)$ is not empty. Finally, we take G^r_{init} to be $\bigcup_{g\in\hat{G}_r} G_{r,g}$. The size of G^r_{init} is at most $k^{|\mathsf{Vars}|-M}$.

Now fix $r \in R$ and $g_{\mathsf{init}} \in G^r_{\mathsf{init}}$ and let $D := g_{\mathsf{init}}(E_r)$. From this point on, we drop "r" from $\mathsf{Learnable}_r$, $\mathsf{Learnable}^*_r$, $\mathsf{SiblBnPath}_r$, BnPath_r, and i^r_{sc}. Since r is thrifty for every I in D, we have $v^D_{2i_{\mathsf{sc}}}\!\downarrow$ and $v^D_{2i_{\mathsf{sc}}+1}\!\downarrow$ (note r queries the variable $f_{i_{\mathsf{sc}}}(v^D_{2i_{\mathsf{sc}}}, v^D_{2i_{\mathsf{sc}}+1})$). Since we have now fixed $D = g_{\mathsf{init}}(E_r)$, when g is an extension of g_{init} we just write v_i^g and X^g instead of $v_i^{g(D)}$ and $X^{g(D)}$.

As in the proof of Theorem 4, we will define a procedure called INTERADV (short for "Interpret Advice") that takes advice in the form of a $[k]$-string and interprets it as the code of an input in D. Ultimately we want to show:

Proposition 1. *For every $I \in D$, there is some restriction g that extends g_{init} and some advice adv^I of length at most $M - h$, such that* $\textsc{InterAdv}(\mathsf{adv}^I)\!\downarrow = g$ *and $I \in g(D)$ and $|\mathsf{Dom}(g) - \mathsf{Dom}(g_{\mathsf{init}})| \geq |\mathsf{adv}^I| + h$.*

The procedure INTERADV is given precisely in pseudocode on page 23 and relies on the subprocedures given on page 23 and the following simple definition, which depends on the fixed input set D:

Definition 10 (g constrains v_i)**.** We say g constrains v_i if for some $I \in g(D)$, the thrifty i variable of I is in $\mathsf{Dom}(g)$

Recall how in the proof of Theorem 4, while reading the advice adv^I for $I \in D$, we maintain a current state $q \in \mathsf{Path}^I(r)$ and build up a set of "learned nodes" which we called U_{L}. We are still building up a set of learned nodes, though in the pseudocode we have opted not to introduce a variable for that set explicitly. The learned nodes are just those nodes j such that at some point during the execution of $\textsc{InterAdv}(\mathsf{adv}^I)$, the subprocedure LEARNNODE is called with second argument j. In the thrifty proof, to characterize how we are interpreting the prefix of the advice that we have read so-far, we only need to record at most one value per node because every input is limited to querying its n thrifty variables (in the pseudocode we used the variable v^*, a partial mapping from T to $[k]$). More precisely, we had that if $v^*(i)\!\downarrow = a$ after reading some advice

elements $\vec{b}$, then $v_i^I = a$ for *every* input I in E_r whose complete advice adv^I has $\vec{b}$ as a prefix, i.e. for every input in $v^*(E_r)$. Now that inputs can query non-thrifty variables, instead of v^* we will be building up a restriction g, where initially $g = g_{\mathsf{init}}$. However, the meaning of $g(X)\downarrow = a$ is what one would expect by analogy with v^*: if $g(X)\downarrow = a$ after reading some advice elements $\vec{b}$, then $X^I = a$ for every input I in D whose complete advice adv^I has $\vec{b}$ as a prefix, i.e. for every input in $g(D)$. As with v^* before, once we define the value g takes on a given variable, we never change it.

We first learn the children of i_{sc} at r; we treat this as a special case now because it is the only time when we learn two nodes while examining one state. After that we learn a node in essentially the same situation as before: we reach a state q after reading some of the advice such that:

1. q queries a variable $f_i(a_{2i}, a_{2i+1})$ that is thrifty for every $I \in g(D)$, and [8]
2. For $j = 2i$ or $j = 2i + 1$ (not both), g does not constrain v_j (j is the learned node).

We need $h - 2$ such states after r for each input in D. Let us say q is a **learning state** for $I \in D$ if both those conditions hold or if $q = r$. In fact, by the properties of g_{init}, and since after r we will only ever learn nodes in $\mathsf{Learnable}^* = \bigcup_{j \in \mathsf{SiblBnPath}} \mathsf{RightPath}_j$, we can write the previous conditions in a more informative way:

1. For some internal $i \in \mathsf{Learnable}^* + (\mathsf{BnPath} - i_{\mathsf{sc}})$, q queries a variable $f_i(a_{2i}, a_{2i+1})$ that is thrifty for every $I \in g(D)$, and
2. If i is in $\mathsf{Learnable}^*$ then g does not constrain v_{2i+1}.
   If i is in $\mathsf{BnPath} - i_{\mathsf{sc}}$ and j is the child of i in BnPath, then g does not constrain $v_{\mathsf{sibl}(j)}$.

We can be more specific still; later we will show that for each of the $h - 2$ nodes $j \in \mathsf{SiblBnPath}$, we will learn at least one node in $\mathsf{RightPath}_j$.

Let us now explain what "learning a node" entails. Temporarily fix $I \in D$. Suppose that while interpreting the advice for I we reach a state $q \in \mathsf{Path}^I(r)^I$ that is a learning state for I. So q queries the variable $f_i(a_{2i}, a_{2i+1})$ for some i in $\mathsf{Learnable}^* + (\mathsf{BnPath} - i_{\mathsf{sc}})$ and a_{2i}, a_{2i+1} in $[k]$. If i is in $\mathsf{BnPath} - i_{\mathsf{sc}}$ then let j be the child of i in BnPath, and otherwise let j be $2i + 1$. We are learning node j. If j is an internal node, then first we use the advice, if necessary, to make g total on $\mathsf{Vars}(T_{2j} + T_{2j+1})$. After that, there is one variable X that is the thrifty j variable for every $I \in g(D)$. So then we "learn" j by adding $X \mapsto a_j$ to g. The key point is that we have made progress since we used only $m = |\mathsf{Dom}(g) \,/\, \mathsf{Vars}(T_{2j} + T_{2j+1})|$ new elements of advice to define g on $m + 1$ new variables.

The main thing we still need to show is that we can define adv^I so that $\textsc{InterAdv}(\mathsf{adv}^I)$ will visit at least $h - 2$ learning states for I after r. As mentioned earlier, I has at least one bottleneck node in $\mathsf{RightPath}_i$ for each of the $h - 2$ nodes $i \in \mathsf{SiblBnPath}$. By Fact 3, for each of those bottleneck nodes j there is a state q_j^I in $\mathsf{Path}^I(r)$ that queries the thrifty $\mathsf{par}(j)$ variable of I, and no state between r and q_j^I that queries the thrifty j variable of I.

For each $i \in \mathsf{SiblBnPath}_r$, let $\hat{q}_i^I$ be the earliest state in $\mathsf{Path}^I(r)$ among the states

$$\{q_j^I \mid j \in \mathsf{RightPath}_i \text{ and } j \text{ is a bottleneck node of } I\}$$

[8] Here g is the current restriction.

and let j_i be such that $\hat{q}_i^I = q_{j_i}^I$. Then at least the nodes $\{j_i\}_{i \in \mathsf{SiblBnPath}}$ will be learned, and specifically j_i will be learned upon reaching $q_{j_i}^I$. To prove this, for $\mathsf{par}(j_i) \in \mathsf{Learnable}^*$ use Fact 3 together with the comments given in footnote 10 on page 24. For $\mathsf{par}(j_i) \in \mathsf{BnPath} - i_{\mathsf{sc}}$, use the following fact (with $j = \mathsf{sibl}(j_i)$ and $j' = \mathsf{par}(j_i)$):

Fact 4. *For all $I \in D$, if j is a non-root node in* BnPath *and j' is its ancestor in* BnPath*, then there are states in $\mathsf{Path}^I(r)$ that query the thrifty j and j' variables of I, and the first such state for j occurs before the first such state for j'.*

Pseudocode for INTERADV **and subprocedures**

The procedure FILL implements a very simple function: given inputs g, V (the advice string $\vec{a}$ and the current index into it are implicit arguments), it just uses the advice to define g on any variable in V on which it is not yet defined. We call FILL in two qualitatively distinct situations. One is when for some i, V is a single i variable X such that for every $I \in g(D)$, we have determined that either i is not a bottleneck node of I or X is not thrifty for I. That is the situation when we call FILL from INTERADV. The other situation occurs when LEARNNODE calls FILL on $\mathsf{Vars}(T_{2j} \cup T_{2j+1})$ for some j that we have decided to learn. We do this because in order to learn j, we need g to be defined on enough input variables that the inputs in $g(D)$ agree on the "name" of their thrifty j variable, i.e. we need $v_{2j}^g\!\downarrow$ and $v_{2j+1}^g\!\downarrow$.

Subprocedure FILL$(g \in G, V \subseteq \mathsf{Vars})$:

1: let $a_1, \ldots, a_m$ be the next $m = |V/\mathsf{Dom}(g)|$ elements of the advice string
2: let $X_1, \ldots, X_m$ be $V/\mathsf{Dom}(g)$ sorted according to some globally fixed order on Vars
3: add $X_1 \mapsto a_1, \ldots, X_m \mapsto a_m$ to g

Subprocedure LEARNNODE$(g \in G, j \in \mathsf{Learnable}^*, b \in [k])$:

1: **if** j is not a leaf **then**
2: FILL$(g, \mathsf{Vars}(T_{2j} + T_{2j+1}))$
3: let $X := f_j(v_{2j}^g, v_{2j+1}^g)$
4: **else**
5: let $X = l_j$
6: **end if**
7: add $X \mapsto b$ to g

Procedure INTERADV$(\vec{a} \in [k]^*)$**:**

1: // *Note the advice string $\vec{a}$ and the current index into it are implicit arguments in every call to* FILL *and* LEARNNODE.
2: $q \leftarrow r,\ g \leftarrow g_{\mathsf{init}}$
3: **while** q is not an output state **do**
4: $i \leftarrow \mathsf{node}(q)$, $X \leftarrow \mathsf{var}(q)$
5: **if** $X \notin \mathsf{Dom}(g)$ **then**
6: **if** $i = i_{\mathsf{sc}}$ **then**

add $l_{2i_{\mathsf{sc}}} \mapsto v_{2i_{\mathsf{sc}}}^{g_{\mathsf{init}}}$ and $l_{2i_{\mathsf{sc}}+1} \mapsto v_{2i_{\mathsf{sc}}+1}^{g_{\mathsf{init}}}$ to g
else if $i \in \mathsf{BnPath} - i_{\mathsf{sc}}$ **then**
let j be the child of i in BnPath [9]
let a_{2i}, a_{2i+1} be such that $X = f_i(a_{2i}, a_{2i+1})$
if $v_j^g\downarrow = a_j$ and g does not constrain $v_{\mathsf{sibl}(j)}$ **then**
// *Uses* $|\mathsf{Vars}(\mathsf{descendants}(\mathsf{sibl}(j))) \,/\, \mathsf{Dom}(g)|$ *elements of advice:*
LEARNNODE$(g, \mathsf{sibl}(j), a_{\mathsf{sibl}(j)})$
else
FILL$(g, \{X\})$ // *Uses one element of advice.*
end if
else // $i \in \mathsf{Learnable}^*$
if i is an internal node and g does not constraint v_{2i+1} **then**
let b be such that $X = f_i(v_{2i}^{g_{\mathsf{init}}}, b)$ [10]
// *Uses* $|\mathsf{Vars}(\mathsf{descendants}(2i+1)) \,/\, \mathsf{Dom}(g)|$ *elements of advice:*
LEARNNODE$(g, 2i+1, b)$
else
FILL$(g, \{X\})$ // *Uses one element of advice.*
end if
end if
$q \leftarrow$ the state reached by taking the edge out of q labeled $g(X)$
end while
return g

□

4.2.2 Less-Thrifty BPs with Additional Queried Variables

The previous result can be generalized to give gradually weaker lower bounds for gradually weaker restrictions on the model. For B a deterministic BP that solves $BT^h(k)$, for every state q of B that queries a variable $f_i(a, b)$, let $\mathsf{RightThrifty}(q)$ be the set of integers a' (including a) such that there is some input to B that visits q and has values a' and b for nodes $2i$ and $2i+1$. Likewise, let $\mathsf{LeftThrifty}(q)$ be the set of integers b' such that there is some input that visits q and has values a and b' for nodes $2i$ and $2i+1$. Theorem 6 is the special case of the following result when $\pi = 1$.

Theorem 7. *For any $h, k \geq 2$ and $\pi < k$, if B is a deterministic BP that solves $BT^h(k)$ such that $|\mathsf{LeftThrifty}(q)| \leq \pi$ and $|\mathsf{RightThrifty}(q)| \leq \pi$ for every state q that queries an internal node, then B has at least k^h/π^{h-2} states.*

Proof. We modify the proof of Theorem 6. We first need to verify that the analogue of Lemma 9 for this context holds:

[9] This makes sense because every node in BnPath other than i_{sc} has a child in BnPath.

[10] $v_{2i}^g\downarrow$ by definition of g_{init} since $2i$ is the left child of a node in $\mathsf{Learnable}$. Also $X = f_i(v_{2i}^g, b)$ for some b since g_{init} is not defined on X. Also $v_{2i+1}^g\downarrow = b$ – since X is not wrong-wrong for any $I \in g(D)$, it must be thrifty for every $I \in g(D)$.

Lemma 11. *For any I and internal node i, there is at least one state q on the computation path of I that queries the thrifty i variable of J, and for every such q, for each child j of i, there is a state on the computation path of I before q that queries the thrifty j variable of I.*

Proof. We use the strategy from the proof of Lemma 4 on page 10. I must visit at least one state that queries its thrifty root variable, since otherwise B would make a mistake on an input J that is identical to I except $f_1^J(v_2^I, v_3^I) = k - f_1^I(v_2^I, v_3^I)$. Now let q be a state on the computation path of I that queries the thrifty i variable of I, for some internal node i. Suppose the lemma does not hold for this q, and wlog assume there is no earlier state that queries the thrifty $2i$ variable of I. For every $a \neq v_{2i}^I$ there is an input J_a that is identical to J except $v_{2i}^{J_a} = a$. This implies $|\mathsf{RightThrifty}(q)| = k$, contradicting the assumption that $|\mathsf{RightThrifty}(q)| \leq \pi < k$. □

The assignment of pebbling sequences to inputs and the definition of supercritical states is the same. In fact nothing more needs to be changed until the statement of the Main Lemma, which is now:

Lemma 12 (Main Lemma)**.** *For some integer M, for every supercritical state $r \in R$, there is a set of restrictions G_{init}^r of size at most $k^{|\mathsf{Vars}|-M}$ such that $\{g_{\mathsf{init}}(E_r)\}_{g_{\mathsf{init}} \in G_{\mathsf{init}}^r}$ is a partition of E_r and for every g_{init} in G_{init}^r, there is an injective function from $g_{\mathsf{init}}(E_r)$ to $[\pi]^{h-2} \times [k]^{M-h}$.*

So in order to cope with the relaxed restrictions on the model, in addition to the $[k]$-valued advice string of length $M - h$ we now have a $[\pi]$-valued advice string of length $h - 2$. One can show the theorem follows from the lemma in the same way as in the proof of Theorem 6. Really at this point there is just one additional observation needed to adapt the proof of Theorem 6: Suppose we have a set of inputs F all of which have value a for v_{2i} (i.e. $v_{2i}^F{\downarrow} = a$), and all the inputs in F visit a state q that queries a variable $f_i(a, b)$. Then we can use the elements of $[\pi]$ to code the values of v_{2i+1} for inputs in F. More concretely, let $a_1, \ldots, a_m$ be the $m \leq \pi$ integers $\mathsf{LeftThrifty}(q)$ in increasing order. Then to each $I \in F$ we assign the index of v_{2i+1}^I in $a_1, \ldots, a_m$. Of course a similar property holds for the case when F is a set of inputs that agree on v_{2i+1}. We use this observation later to show that if we "know" the value of node $2i$ upon reaching q, then we can learn node $2i + 1$ with the help of just an element of π-valued advice, and similarly for learning node $2i$.

The definition of G_{init}^r is the same, and as before we fix $r \in R$ and $g_{\mathsf{init}} \in G_{\mathsf{init}}^r$ and then define a procedure that interprets some given advice as the code of an input in $D := g_{\mathsf{init}}(E_r)$. The analogue of Proposition 1 (page 21) is:

Proposition 2. *For every $I \in D$, there is some restriction g that extends g_{init}, a $[\pi]$-valued advice string adv_π^I of length $h - 2$ and a $[k]$-valued advice string adv_k^I of length at most $M - h$, such that $\textsc{InterAdv}(\mathsf{adv}_k^I, \mathsf{adv}_\pi^I){\downarrow} = g$ and $I \in g(D)$ and $|\mathsf{Dom}(g) - \mathsf{Dom}(g_{\mathsf{init}})| \geq |\mathsf{adv}^I| + h$.*

However, it will be convenient to instead give a procedure $\textsc{InterAdv}'$ for which the following superficially different proposition holds:

Proposition 3. *For every $I \in D$, there is some restriction g that extends g_{init}, a $[\pi]$-valued advice string adv_π^I of length at least $h - 2$ and a $[k]$-valued advice string adv_k^I of length at most $M -$*

$|\mathsf{adv}^I_\pi| - 2$*, such that* INTERADV$'(\mathsf{adv}^I_k, \mathsf{adv}^I_\pi)\downarrow = g$ *and* $I \in g(D)$ *and* $|\mathsf{Dom}(g) - \mathsf{Dom}(g_{\mathsf{init}})| \geq |\mathsf{adv}^I| + |\mathsf{adv}^I_\pi| + 2$.

To get the procedure INTERADV of Proposition 2 from the procedure INTERADV$'$ of Proposition 3, just run INTERADV$'$ until $h-2$ elements of the $[\pi]$-valued advice have been used, and then, if necessary, use elements of the $[k]$-valued advice whenever an additional element of the $[\pi]$-valued advice is required. This works since $\pi \leq k$ and $|\mathsf{adv}^I_k| \leq M - |\mathsf{adv}^I_\pi| - 2$.

Let us say q **is right-thrifty for** I if q queries a variable $f_i(a,b)$ such that $b = v^I_{2i+1}$ and $a \neq v^I_{2i}$. Similarly define **left-thrifty for** I. Previously, while interpreting the advice for I we only learned node values at states that are thrifty for I. Now we may learn node values at states that are thrifty, right-thrifty, or left-thrifty for I. As before, we always learn the children of i_{sc}, and the remaining $h-2$ nodes we learn are in $\mathsf{Learnable}^*$.

First we consider the case of learning a node in $\mathsf{SiblBnPath}$. We consider the case of learning a left child $2i$ – the case of learning a right child is similar. Let q be the first state in $\mathsf{Path}^I(r)$ that queries the thrifty $2i+1$ variable of I. *If* we learn $2i$, then we do so at the first state q' after q that queries an i variable that is thrifty or right-thrifty for I. Now we consider the case of learning a node in $\mathsf{Learnable}^* - \mathsf{SiblBnPath}$. Every node in $\mathsf{Learnable}^* - \mathsf{SiblBnPath}$ is a right child, so suppose we are learning $2i+1$. Then we do so at the first state in $\mathsf{Path}^I(r)$ that queries an i variable that is thrifty or left-thrifty for I.

As before, for each I in D and each of the $h-2$ nodes i in $\mathsf{SiblBnPath}$, we will learn at least one node in $\mathsf{RightPath}_i$ (and of course we still learn the children of the supercritical node i_{sc}). This is again proved using Facts 3 (page 19) and 4 (page 23); both still hold since we did not change the assignment of pebbling sequences to inputs.

We provide pseudocode for INTERADV$'$, just in case the reader has questions not explicitly addressed in the preceding prose. On the other hand, there is little to read since it **differs from the previous definition of** INTERADV **(4.2.1 on page 23) only in a few lines** near the two calls to LEARNNODE (specifically lines 12 - 15 and 21 - 24). The two subprocedures FILL and LEARNNODE do not use the $[\pi]$-valued advice and do not need to be modified.

Procedure INTERADV$'(\vec{a}_k \in [k]^*, \vec{a}_\pi \in [\pi]^*)$**:**

1: // *Note the advice string* $\vec{a}_k$ *and the current index into it are both implicit arguments in every call to* FILL *and* LEARNNODE.
2: $q \leftarrow r,\ g \leftarrow g_{\mathsf{init}}$
3: **while** q is not an output state **do**
4: $\quad i \leftarrow \mathsf{node}(q)$, $X \leftarrow \mathsf{var}(q)$
5: $\quad$ **if** $X \notin \mathsf{Dom}(g)$ **then**
6: $\quad\quad$ **if** $i = i_{\mathsf{sc}}$ **then**
7: $\quad\quad\quad$ add $l_{2i_{\mathsf{sc}}} \mapsto v^{g_{\mathsf{init}}}_{2i_{\mathsf{sc}}}$ and $l_{2i_{\mathsf{sc}}+1} \mapsto v^{g_{\mathsf{init}}}_{2i_{\mathsf{sc}}+1}$ to g
8: $\quad\quad$ **else if** $i \in \mathsf{BnPath} - i_{\mathsf{sc}}$ **then**
9: $\quad\quad\quad$ let j be the child of i in BnPath
10: $\quad\quad\quad$ let a_{2i}, a_{2i+1} be such that $X = f_i(a_{2i}, a_{2i+1})$
11: $\quad\quad\quad$ **if** $v^g_j\downarrow = a_j$ and g does not constrain $v_{\mathsf{sibl}(j)}$ **then**
12: $\quad\quad\quad\quad$ let z be the next element of the $[\pi]$-valued advice.

if $j = 2i + 1$ then let b be the z-th greatest integer in $\mathsf{RightThrifty}(q)$ and otherwise let b be the z-th greatest integer in $\mathsf{LeftThrifty}(q)$
// *Uses* $|\mathsf{Vars}(\mathsf{descendants}(\mathsf{sibl}(j)))$ / $\mathsf{Dom}(g)|$ *elements of* $[k]$*-valued advice:*
LEARNNODE$(g, \mathsf{sibl}(j), b)$
else
FILL$(g, \{X\})$ // *Uses one element of* $[k]$*-valued advice.*
end if
else // $i \in \mathsf{Learnable}^*$
if i is an internal node and g does not constraint v_{2i+1} **then**
let z be the next element of the $[\pi]$-valued advice.
let b be the z-th greatest integer in $\mathsf{LeftThrifty}(q)$.
// *Uses* $|\mathsf{Vars}(\mathsf{descendants}(2i + 1))$ / $\mathsf{Dom}(g)|$ *elements of* $[k]$*-valued advice:*
LEARNNODE$(g, 2i + 1, b)$
else
FILL$(g, \{X\})$ // *Uses one element of* $[k]$*-valued advice.*
end if
end if
end if
$q \leftarrow$ the state reached by taking the edge out of q labeled $g(X)$
end while
return g

□

We give one more extension of the thrifty lower bound. We introduce another parameter w: for each input I, we require that there are at most w nodes i such that I visits a state q with $|\mathsf{RightThrifty}(q)| > 1$ or $|\mathsf{LeftThrifty}(q)| > 1$. The motivation for this is that for $w = 1$ and $\pi = \log k - \log\log k$, the model includes BPs that achieve the best known upper bounds for $BT^h(k)$, namely $O(k^h/\log k)$. For those parameters the theorem gives a lower bound of $k^h/(\log k - \log\log k) = \Omega(k^h/\log k)$. In [BCM+09b] it was shown that the minimum number of states for *unrestricted* deterministic BPs solving $BT^3(k)$ is $\Theta(k^3/\log k)$.

Theorem 8. *For any $h, k \geq 2$ and $\pi < k$ and $w < h - 2$, if B is a deterministic BP that solves $BT^h(k)$ such that $|\mathsf{LeftThrifty}(q)| \leq \pi$ and $|\mathsf{RightThrifty}(q)| \leq \pi$ for every state q that queries an internal node, and such that for every input I there are at most w nodes i such that I visits a state q that queries an i-variable and has $|\mathsf{RightThrifty}(q)| > 1$ or $|\mathsf{LeftThrifty}(q)| > 1$, then B has at least k^h/π^w states.*

Proof. This is an easy modification of the proof of the previous result. For all but w of the $h - 2$ learning states q after r, we do not need to use an element of the $[\pi]$-valued advice to learn a child of $\mathsf{node}(q)$. Hence we only need a $[\pi]$-valued advice string of length w. □

5 Open Problems

The first is a problem that can, in principle, be resolved using a computer.

1. Show that for some k, h there is a deterministic branching program with fewer than $(k+1)^h$ states that solves $FT^h(k)$.

Theorem 4 suggests the following conjecture: for all h, nondeterministic thrifty branching programs solving $FT^h(k)$ require $\Omega(k^{\#\mathsf{FRpebbles}(T^h)})$ states.

2. Refute it, with or without the thrifty restriction.

References

[BCM+09a] Mark Braverman, Stephen Cook, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Fractional pebbling and thrifty branching programs. In Ravi Kannan and K Narayan Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 109–120, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BCM+09b] Mark Braverman, Stephen A. Cook, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Branching programs for tree evaluation. In Rastislav KrÃ¡lovic and Damian Niwinski, editors, *MFCS*, volume 5734 of *Lecture Notes in Computer Science*, pages 175–186. Springer, 2009.

[BCM+09c] Mark Braverman, Stephen A. Cook, Pierre McKenzie, Rahul Santhanam, and Dustin Wehr. Pebbles and branching programs for tree evaluation. A draft manuscript, available on line at `http://www.cs.toronto.edu/~sacook/homepage/pebbles.pdf`, 2009.

[Coo74] S. Cook. An observation on time-storage trade off. *J. Comput. Syst. Sci.*, 9(3):308–316, 1974.

[CS76] S. Cook and R. Sethi. Storage requirements for deterministic polynomial time recognizable languages. *J. Comput. Syst. Sci.*, 13(1):25–37, 1976.

[Kla85] M. Klawe. A tight bound for black and white pebbles on the pyramid. *J. ACM*, 32(1):218–228, 1985.

[Neč66] È. Nečiporuk. On a boolean function. *Doklady of the Academy of the USSR*, 169(4):765–766, 1966. English translation in *Soviet Mathematics Doklady* 7:4, pp. 999-1000.

[Nor09] J. Nordström. New wine into old wineskins: A survey of some pebbling classics with supplemental results. Available on line at `http://people.csail.mit.edu/jakobn/research/`, 2009.

[PH70] M. Paterson and C. Hewitt. Comparative schematology. In *Record of Project MAC Conference on Concurrent Systems and Parallel Computations*, pages 119–128, 1970. (June 1970) ACM. New Jersey.

[Raz91] A. Razborov. Lower bounds for deterministic and nondeterministic branching programs. In *8th Internat. Symp. on Fundamentals of Computation Theory*, pages 47–60, 1991.