

Pebbles and Branching Programs for Tree Evaluation

STEPHEN COOK, University of Toronto
 PIERRE MCKENZIE, Université de Montréal
 DUSTIN WEHR, University of Toronto
 MARK BRAVERMAN, University of Toronto
 RAHUL SANTHANAM, University of Edinburgh

We introduce the *tree evaluation problem*, show that it is in **LogDCFL** (and hence in **P**), and study its branching program complexity in the hope of eventually proving a superlogarithmic space lower bound. The input to the problem is a rooted, balanced d -ary tree of height h , whose internal nodes are labeled with d -ary functions on $[k] = \{1, \dots, k\}$, and whose leaves are labeled with elements of $[k]$. Each node obtains a value in $[k]$ equal to its d -ary function applied to the values of its d children. The output is the value of the root. We show that the standard black pebbling algorithm applied to the binary tree of height h yields a deterministic k -way branching program with $O(k^h)$ states solving this problem, and we prove that this upper bound is tight for $h = 2$ and $h = 3$. We introduce a simple semantic restriction called *thrifty* on k -way branching programs solving tree evaluation problems and show that the same state bound of $\Theta(k^h)$ is tight for all $h \geq 2$ for deterministic thrifty programs. We introduce fractional pebbling for trees and show that this yields nondeterministic thrifty programs with $\Theta(k^{h/2+1})$ states solving the Boolean problem “determine whether the root has value 1”, and prove that this bound is tight for $h = 2, 3, 4$. We also prove that this same bound is tight for unrestricted nondeterministic k -way branching programs solving the Boolean problem for $h = 2, 3$.

Categories and Subject Descriptors: F.2.0 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity

General Terms: Theory

Additional Key Words and Phrases: branching programs, logDCFL, log space, lower bounds

Contents

1	Introduction	2
1.1	Summary of Contributions	5
1.2	Relation to previous work	6
1.3	Organization	6
2	Preliminaries	7
2.1	Branching programs	7
2.2	One function is enough	9
2.3	Pebbling	10
3	Connecting TMs, BPs, and Pebbling	12
4	Pebbling Bounds	14

Versions of parts of this paper appeared in [Braverman et al. 2009a] and [Braverman et al. 2009b].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 0000-0000/YYYY-ARTA \$10.00

DOI 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

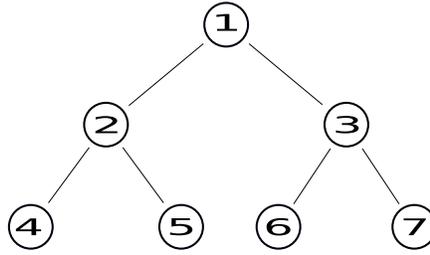


Fig. 1. A height 3 binary tree T_2^3 with nodes numbered heap style.

4.1	Previous results	14
4.2	Results for fractional pebbling	17
4.3	White sliding moves	27
5	Branching Program Bounds	27
5.1	The Nečiporuk method	28
5.2	The state sequence method	32
5.3	Thrifty lower bounds	36
6	Conclusion	41

1. INTRODUCTION

Below is a nondecreasing sequence of standard complexity classes between $\text{AC}^0(6)$ and the polynomial hierarchy.

$$\text{AC}^0(6) \subseteq \text{NC}^1 \subseteq \text{L} \subseteq \text{NL} \subseteq \text{LogCFL} \subseteq \text{AC}^1 \subseteq \text{NC}^2 \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PH} \quad (1)$$

A problem in $\text{AC}^0(6)$ is given by a uniform family of polynomial size bounded depth circuits with unbounded fan-in Boolean and mod 6 gates. As far as we can tell an $\text{AC}^0(6)$ circuit cannot determine whether a majority of its input bits are ones, and yet we cannot provably separate $\text{AC}^0(6)$ from any of the other classes in the sequence. This embarrassing state of affairs motivates this paper (as well as much of the lower bound work in complexity theory).

We propose a candidate for separating NL from LogCFL. The *Tree Evaluation problem* $FT_d(h, k)$ is defined as follows. The input to $FT_d(h, k)$ is a balanced d -ary tree of height h , denoted T_d^h (see Fig. 1). Attached to each internal node i of the tree is some explicit function $f_i : [k]^d \rightarrow [k]$ specified as k^d integers in $[k] = \{1, \dots, k\}$. Attached to each leaf is a number in $[k]$. Each internal tree node takes a value in $[k]$ obtained by applying its attached function to the values of its children. The function problem $FT_d(h, k)$ is to compute the value of the root, and the Boolean problem $BT_d(h, k)$ is to determine whether this value is 1.

It is not hard to show that a deterministic logspace-bounded polytime auxiliary pushdown automaton decides $BT_d(h, k)$, where d, h and k are input parameters. This implies by [Sudborough 1978] that $BT_d(h, k)$ belongs to the class LogDCFL of languages logspace reducible to a deterministic context-free language. The latter class lies between L and LogCFL, but its relationship with NL is unknown (see [Mahajan 2007] for a recent survey). We conjecture that $BT_d(h, k)$ does not lie in NL. A proof would separate NL and LogCFL, and hence (by (1)) separate NC^1 and NC^2 .

Thus we are interested in proving superlogarithmic space upper and lower bounds (for fixed degree $d \geq 2$) for $BT_d(h, k)$ and $FT_d(h, k)$. Notice that for each constant $k =$

$k_0 \geq 2$, $BT_d(h, k_0)$ is an easy generalization of the Boolean formula value problem for balanced formulas, and hence it is in NC^1 and L . Thus it is important that k be an unbounded input parameter.

We use branching programs (BPs) as a nonuniform model of Turing machine space: A lower bound of $s(n)$ on the number of BP states implies a lower bound of $\Theta(\log s(n))$ on Turing machine space, but to go the other way, i.e., to deduce BP size lower bounds from Turing machine space lower bounds, we would need to defeat a Turing machine supplied with an advice string for each input length. Thus BP state lower bounds are stronger than TM space lower bounds, but we do not know how to take advantage of the uniformity of TMs to get the supposedly easier lower bounds on TM space. In this paper all of our lower bounds are nonuniform and all of our upper bounds are uniform.

In the context of branching programs we think of d and h as fixed, and we are interested in how the number of states required grows with k . To indicate this point of view we write the function problem $FT_d(h, k)$ as $FT_d^h(k)$ and the Boolean problem $BT_d(h, k)$ as $BT_d^h(k)$. For this it turns out that k -way BPs are a convenient model, since an input for $BT_d^h(k)$ or $FT_d^h(k)$ is naturally presented as a tuple of elements in $[k]$. Each nonfinal state in a k -way BP queries a specific element of the tuple, and branches k possible ways according to the k possible answers.

It is natural to assume that the inputs to Turing machines are binary strings, so 2-way BPs are a closer model of TM space than are k -way BPs for $k > 2$. But every 2-way BP is easily converted to a k -way BP with the same number of states, and every k -way BP can be converted to a 2-way BP with an increase of only a factor of k in the number of states, so for the purpose of separating L and P we may as well use k -way BPs.

Of course the number of states required by a k -way BP to solve the Boolean problem $BT_d^h(k)$ is at most the number required to solve the function problem $FT_d^h(k)$. In the other direction it is easy to see (Lemma 2.3) that $FT_d^h(k)$ requires at most a factor of k more states than $BT_d^h(k)$. From the point of view of separating L and P a factor of k is not important. Nevertheless it is interesting to compare the two numbers, and in some cases (Corollary 5.2) we can prove tight bounds for both: For deterministic BPs solving height 3 trees they differ by a factor of $\log k$ rather than k .

The best (i.e. fewest states) algorithms that we know for deterministic k -way BPs solving $FT_d^h(k)$ come from black pebbling algorithms for trees: If p pebbles suffice to pebble the tree T_d^h then $O(k^p)$ states suffice for a BP to solve $FT_d^h(k)$ (Theorem 3.4). This upper bound on states is tight (up to a constant factor) for trees of height $h = 2$ or $h = 3$ (Corollary 5.2), and we suspect that it may be tight for trees of any height.

There is a well-known generalization of black pebbling called black-white pebbling which naturally simulates nondeterministic algorithms. Indeed if p pebbles suffice to black-white pebble T_d^h then $O(k^p)$ states suffice for a nondeterministic BP to solve $BT_d^h(k)$. However the best lower bound we can obtain for nondeterministic BPs solving $BT_2^3(k)$ (see Figure 1) is $\Omega(n^{2.5})$, whereas it takes 3 pebbles to black-white pebble the tree T_2^3 . This led us to rethink the upper bound, and we discovered that there is indeed a nondeterministic BP with $O(k^{2.5})$ states which solves $BT_2^3(k)$. The algorithm comes from a black-white pebbling of T_2^3 using only 2.5 pebbles: It places a half-black pebble on node 2, a black pebble on node 3, and adds a half white pebble on node 2, allowing the root to be black-pebbled (see Figure 2 on page 18).

This led us to the idea of fractional pebbling in general, a natural generalization of black-white pebbling. A fractional pebble configuration on a tree assigns two nonnegative real numbers $b(i)$ and $w(i)$ totalling at most 1, to each node i in the tree, with appropriate rules for removing and adding pebbles. The idea is to minimize the max-

imum total pebble weight on the tree during a pebbling procedure which starts and ends with no pebbles and has a black pebble on the root at some point.

It turns out that nondeterministic BPs nicely implement fractional pebbling procedures: If p pebbles suffice to fractionally pebble T_d^h then $O(k^p)$ states suffice for a nondeterministic BP to solve $BT_d^h(k)$. The idea is that if node i has a fraction $b(i) + w(i)$ pebbles then the corresponding BP configuration remembers a fraction $b(i) + w(i)$ of the $\log k$ bits specifying the value of node i , where $b(i)$ bits are verified and $w(i)$ bits are conjectured. After much work we have not been able to improve upon this $O(k^p)$ upper bound for any $d, h \geq 2$. We prove it is optimal for trees of height 3 (Corollary 5.2).

We can prove that for fixed degree d the number of pebbles required to pebble (in any sense) the tree T_d^h grows as $\Theta(h)$, so the p in the above best-known upper bound of $O(k^p)$ states grows as $\Theta(h)$. This and the following fact motivate further study of the complexity of $FT_d^h(k)$.

FACT 1. *A lower bound of $\Omega(k^{r(h)})$ for any unbounded function $r(h)$ on the number of states required to solve $FT_d^h(k)$ implies that $\mathbf{L} \neq \mathbf{LogCFL}$ (Theorem 3.1 and Corollary 3.3).*

Proving tight bounds on the number of pebbles required to fractionally pebble a tree turns out to be much more difficult than for the case of whole black-white pebbling. However we can prove good upper and lower bounds. For binary trees of any height h we prove an upper bound of $h/2 + 1$ and a lower bound of $h/2 - 1$ (the upper bound is optimal for $h \leq 4$). These bounds can be generalized to d -ary trees (Theorem 4.4).

We introduce a natural semantic restriction on BPs which solve $BT_d^h(k)$ or $FT_d^h(k)$: A k -way BP is *thrifty* if it only queries the function $f(x_1, \dots, x_d)$ associated with a node when (x_1, \dots, x_d) are the correct values of the children of the node.

It is not hard to see that the deterministic BP algorithms that implement black pebbling are thrifty. With some effort we were able to prove a converse (for binary trees): If p is the minimum number of pebbles required to black-pebble T_2^h then every deterministic thrifty BP solving $BT_2^h(k)$ (or $FT_2^h(k)$) requires at least k^p states. Thus any deterministic BP solving these problems with fewer states must query internal nodes $f_i(x, y)$ where (x, y) are not the values of the children of node i . For the decision problem $BT_2^h(k)$ there is indeed a nonthrifty deterministic BP improving on the bound by a factor of $\log k$ (Theorem 5.1 (16)), and this is tight for $h = 3$ (Corollary 5.2). But we have not been able to improve on thrifty BPs for solving any function problem $FT_d^h(k)$.

The nondeterministic BPs that implement fractional pebbling are indeed thrifty. However here the converse is far from clear: there is nothing in the definition of *thrifty* that hints at fractional pebbling. We have been able to prove that thrifty BPs cannot beat fractional pebbling for binary trees of height $h = 4$ or less, but for general trees this is open.

It is not hard to see that for black pebbling, fractional pebbles do not help. This may explain why we have been able to prove tight bounds for deterministic thrifty BPs for all binary trees, but only for trees of height 4 or less for nondeterministic thrifty BPs.

We pose the following as another interesting open question:

Thrifty Hypothesis: Thrifty BPs are optimal among k -way BPs solving $FT_d^h(k)$.

Proving this for deterministic BPs would show $\mathbf{L} \neq \mathbf{LogDCFL}$, and for nondeterministic BPs would show $\mathbf{NL} \neq \mathbf{LogCFL}$. Disproving this would provide interesting new space-efficient algorithms and might point the way to new approaches for proving lower bounds.

The lower bounds mentioned above for unrestricted branching programs when the tree heights are small are obtained in two ways: First using the Nečiporuk method [Nečiporuk 1966] (or see [Wegener 2000]), and second using a method that analyzes the state sequences of the BP computations. Using the state sequence method we have not yet beat the $\Omega(n^2)$ deterministic branching program size barrier (neglecting log factors) inherent to the Nečiporuk method for Boolean problems, but we can prove lower bounds for function problems which cannot be matched by the Nečiporuk method (Theorems 5.5, 5.6, 5.9, 5.10). For nondeterministic branching programs with states of unbounded outdegree, we show that both methods yield a lower bound of $\Omega(n^{3/2})$ states (neglecting logs) for the decision problem BT_2^3 .

1.1. Summary of Contributions

— We introduce a family of computation problems $FT_d^h(k)$ and $BT_d^h(k)$, $d, h \geq 2$, which we propose as good candidates for separating L and NL from apparently larger complexity classes in (1). Our goal is to prove space lower bounds for these problems by proving state lower bounds for k -way branching programs which solve them. For $h = 3$ we can prove tight bounds for each $d \geq 2$ on the number of states required by k -way BPs to solve them, namely (from Corollary 5.2)

$$\begin{aligned} &\Theta(k^{(3/2)d-1/2}) \text{ for nondeterministic BPs solving } BT_d^3(k) \\ &\Theta(k^{2d-1}/\log k) \text{ for deterministic BPs solving } BT_d^3(k) \\ &\Theta(k^{2d-1}) \text{ for deterministic BPs solving } FT_d^3(k) \end{aligned}$$

— We introduce a simple and natural restriction called *thrifty* on BPs solving $FT_d^h(k)$ and $BT_d^h(k)$. The best known upper bounds for deterministic BPs solving $FT_d^h(k)$ and for nondeterministic BPs solving $BT_d^h(k)$ are realized by thrifty BPs (although deterministic nonthrifty BPs can save a factor of $\log k$ states over deterministic thrifty BPs solving the decision problem $BT_2^h(k)$). Proving even much weaker lower bounds than these upper bounds for unrestricted BPs would separate L from LogCFL (see Fact 1 above). We prove that for binary trees deterministic thrifty BPs cannot do better than implement black pebbling (this is far from obvious).¹

— We formulate the **Thrifty Hypothesis** (see above). Either a proof or a disproof would have interesting consequences.

— We introduce *fractional pebbling* as a natural generalization of black-white pebbling for simulating nondeterministic space bounded computations. We prove almost tight lower bounds for fractionally pebbling binary trees (Theorem 4.4). The best known upper bounds for nondeterministic BPs solving $FT_d^h(k)$ come from fractional pebbling, and these can be implemented by thrifty BPs. An interesting open question is to prove that nondeterministic thrifty BPs cannot do better than implement fractional pebbling. (We prove this for $h = 2, 3, 4$.)

— We use a “state sequence” method for proving size lower bounds for branching programs solving $FT_d^h(k)$ and $BT_d^h(k)$, and show that it improves on the Nečiporuk method for certain function problems.

The next major step is to prove good lower bounds for trees of height $h = 4$. If we can prove the above Thrifty Hypothesis for deterministic BPs solving the function problem (and hence the decision problem) for trees of height 4, then we would beat the $\Omega(n^2)$

¹In [Wehr 2011] coauthor Wehr solved an open problem in [Gál et al. 2008] by adapting our thrifty lower bound proof to prove an exponential lower bound on the size of semantic incremental branching programs solving GEN.

limitation mentioned above on Nečiporuk’s method. See Section 6 (Conclusion) for this argument, and a comment about the nondeterministic case.

1.2. Relation to previous work

Taitslin [Taitslin 2005] proposed a problem similar to $BT_2^h(k)$ in which the functions attached to internal nodes are specific quasi groups, in an unsuccessful attempt to prove $NL \neq P$.

Gál, Koucký and McKenzie [Gál et al. 2008] proved exponential lower bounds on the size of restricted n -way branching programs solving versions of the problem GEN. Like our problems $BT_d^h(k)$ and $FT_d^h(k)$, the best known upper bounds for solving GEN come from pebbling algorithms.

As a concrete approach to separating NC^1 from NC^2 , Karchmer, Raz and Wigderson [Karchmer et al. 1995] suggested proving that the circuit depth required to compose a Boolean function with itself h times grows appreciably with h . They proposed the *universal composition relation* conjecture, stating that an abstraction of the composition problem requires high communication complexity, as an intermediate goal to validate their approach. This conjecture was later proved in two ways, first [Edmonds et al. 2001] using innovative information-theoretic machinery and then [Håstad and Wigderson 1993] using a clever new complexity measure that generalizes the subadditivity property implicit in Nečiporuk’s lower bound method [Nečiporuk 1966; Wegener 2000]. Proving the conjecture thus cleared the road for the approach, yet no sufficiently strong unrestricted circuit lower bounds could be proved using it so far.

Edmonds, Impagliazzo, Rudich and Sgall [Edmonds et al. 2001] noted that the approach would in fact separate NC^1 from AC^1 . They also coined the name *Iterated Multiplexor* for the most general computational problem considered in [Karchmer et al. 1995], namely composing in a tree-like fashion a set of explicitly presented Boolean functions, one per tree node. Our problem $FT_d^h(k)$ can be considered as a generalization of the Iterated Multiplexor problem in which the functions map $[k]^d$ to $[k]$ instead of $\{0, 1\}^d$ to $\{0, 1\}$. This generalization allows us to focus on getting lower bounds as a function of k when the tree is fixed.

For time-restricted branching programs, Borodin, Razborov and Smolensky [Borodin et al. 1993] exhibited a family of Boolean functions that require exponential size to be computed by nondeterministic syntactic read- k times BPs. Later Beame, Saks, Sun, and Vee [Beame et al. 2003] exhibited such functions that require exponential size to be computed by randomized BPs whose computation time is limited to $o(n\sqrt{\log n / \log \log n})$, where n is the input length. However all these functions can be computed by polynomial size BPs when time is unrestricted.

In the present paper we consider branching programs with no time restriction such as read- k times. However the smallest size deterministic BPs known to us that solve $FT_d^h(k)$ implement the black pebbling algorithm, and these BPs happen to be (syntactic) read-once.

1.3. Organization

The paper is organized as follows. Section 2 defines the main notions used in this paper, including branching programs and pebbling. Section 3 relates pebbling and branching programs to Turing machine space, noting in particular that a k -way BP size lower bound of $\Omega(k^{\text{function}(h)})$ for $BT_d^h(k)$ would show $L \neq \text{LogCFL}$. Section 4 proves upper and lower bounds on the number of pebbles required to black, black-white and fractionally pebble the tree T_d^h . These pebbling bounds are exploited in Section 5 to prove upper bounds on the size of branching programs. BP lower bounds are obtained us-

ing the Nečiporuk method in Subsection 5.1. Alternative proofs to some of these lower bounds using the “state sequence method” are given in Subsection 5.2. An example of a function problem for which the state sequence method beats the Nečiporuk method is given in Theorems 5.5 and 5.9. Subsection 5.3 contains bounds for thrifty branching programs.

2. PRELIMINARIES

We assume some familiarity with complexity theory, such as can be found in [Goldreich 2008]. We write $[k]$ for $\{1, 2, \dots, k\}$. For $d, h \geq 2$ we use T_d^h to denote the balanced d -ary 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_d^h as suggested by the heap data structure. Thus the root is node 1, and in general the children of node i are (when $d = 2$) nodes $2i, 2i + 1$ (see Figure 1).

DEFINITION 2.1 (TREE EVALUATION PROBLEMS). *Given: The tree T_d^h with each non-leaf node i independently labeled with a function $f_i : [k]^d \rightarrow [k]$ and each leaf node independently labeled with an element from $[k]$, where $d, h, k \geq 2$.*

Function evaluation problem $FT_d^h(k)$: Compute the value $v_1 \in [k]$ of the root 1 of T_d^h , where in general $v_i = a$ if i is a leaf labeled a and $v_i = f_i(v_{j_1}, \dots, v_{j_d})$ if the children of i are j_1, \dots, j_d .

Boolean problem $BT_d^h(k)$: Decide whether $v_1 = 1$.

2.1. Branching programs

A family of branching programs serves as a nonuniform model of a Turing machine. For each input size n there is a BP B_n in the family which models the machine on inputs of size n . The states (or nodes) of B_n correspond to the possible configurations of the machine for inputs of size n . Thus for $s(n) \in \Omega(\log n)$, if the machine computes in space $s(n)$ then B_n has $2^{O(s(n))}$ states.

Many variants of the branching program model have been studied (see in particular the survey by Razborov [Razborov 1991] and the book by Ingo Wegener [Wegener 2000]). Our definition below is inspired by Wegener [Wegener 2000, p. 239], by the k -way branching program of Borodin and Cook [Borodin and Cook 1982] and by its nondeterministic variant [Borodin et al. 1993; Gál et al. 2008]. We depart from the latter however in two ways: nondeterministic branching program labels are attached to states rather than edges (because we think of branching program states as Turing machine configurations) and cycles in branching programs are allowed (because our lower bounds apply to this more general model²).

DEFINITION 2.2 (BRANCHING PROGRAMS). *A nondeterministic k -way branching program B computing a total function $g : [k]^m \rightarrow R$, 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 $|R|$ final sink states consecutively labelled with the elements from R . An input $(x_1, \dots, x_m) \in [k]^m$ activates, for each $1 \leq j \leq m$, every edge labelled x_j out of every state labelled j . A computation on input $\vec{x} = (x_1, \dots, x_m) \in [k]^m$ is a directed path consisting of edges activated by \vec{x} which begins with the unique start state (the root), and either it is infinite, or it ends in the final state labelled $g(x_1, \dots, x_m)$,*

²A BP with cycles can be simulated by an acyclic BP by at most squaring the number of states. Hence this distinction is not important for separating L and P.

or it ends in a nonfinal state labelled j with no outedge labelled x_j (in which case we say the computation aborts). At least one such computation must end in a final state. The size of B is its number of states. B is deterministic k -way if every non-final state has precisely k outedges labelled $1, \dots, k$. B is binary if $k = 2$.

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

A k -way branching program computing the function $FT_d^h(k)$ requires as input k^d many k -ary arguments for each internal node i of T_d^h in order to specify the function f_i , together with one k -ary argument for each leaf. Thus in the notation of Definition 2.1, $FT_d^h(k): [k]^m \rightarrow R$ where $R = [k]$ and $m = \frac{d^{h-1}-1}{d-1} \cdot k^d + d^{h-1}$. Also $BT_d^h(k): [k]^m \rightarrow \{0, 1\}$.

For fixed d, h we are interested in how the number of states required for a k -way branching program to compute $FT_d^h(k)$ and $BT_d^h(k)$ grows with k . We define $\#detFstates_d^h(k)$ (resp. $\#ndetFstates_d^h(k)$) to be the minimum number of states required for a deterministic (resp. nondeterministic) k -way branching program to solve $FT_d^h(k)$. Similarly we define $\#detBstates_d^h(k)$ and $\#ndetBstates_d^h(k)$ to be the number of states for solving $BT_d^h(k)$.

The next lemma shows that the function problem is not much harder to solve than the Boolean problem.

LEMMA 2.3.

$$\begin{aligned} \#detBstates_d^h(k) &\leq \#detFstates_d^h(k) \leq (k-1) \cdot \#detBstates_d^h(k) \\ \#ndetBstates_d^h(k) &\leq \#ndetFstates_d^h(k) \leq (k-1) \cdot \#ndetBstates_d^h(k) \end{aligned}$$

PROOF. The left inequalities are obvious. For the others, we can construct a branching program solving the function problem from a sequence of $k-1$ programs solving Boolean problems, where the i th program determines whether the value of the root node is i . \square

Next we introduce thrifty programs, a restricted form of k -way branching programs for solving tree evaluation problems. Thrifty programs efficiently simulate pebbling algorithms, and implement the best known upper bounds for $\#ndetBstates_d^h(k)$ and $\#detFstates_d^h(k)$, and are within a factor of $\log k$ of the best known upper bounds for $\#detBstates_d^h(k)$. In Section 5 we prove tight lower bounds for deterministic thrifty programs which solve $BT_d^h(k)$ and $FT_d^h(k)$.

DEFINITION 2.4 (THRIFTY BRANCHING PROGRAM). A deterministic k -way branching program which solves $FT_d^h(k)$ or $BT_d^h(k)$ is thrifty if during the computation on any input every query $f_i(\vec{x})$ to an internal node i of T_d^h satisfies the condition that \vec{x} is the tuple of correct values for the children of node i . A nondeterministic such program is thrifty if for every input every computation which ends in a final state satisfies the above restriction on queries.

Note that the restriction in the above definition is semantic, rather than syntactic. It somewhat resembles the semantic restriction used to define incremental branching programs in [Gál et al. 2008]. However we are able to prove strong lower bounds using our semantic restriction, but in [Gál et al. 2008] a syntactic restriction was needed to prove lower bounds.

2.2. One function is enough

It turns out that the complexities of $FT_d^h(k)$ and $BT_d^h(k)$ are not much different if we require all functions assigned to internal nodes to be the same.³ To denote this restricted version of the problems we replace F by \hat{F} and B by \hat{B} . Thus $\hat{F}T_d^h(k)$ is the function problem for T_d^h when all node functions are the same, and $\hat{B}T_d^h(k)$ is the corresponding Boolean problem. To specify an instance of one of these new problems we need only give one copy of the table for the common node function \hat{f} , together with the values for the leaves.

THEOREM 2.5. *Let $N = (d^h - 1)/(d - 1)$ be a constant denoting the number of nodes in the tree T_d^h . Any Nk -way branching program \hat{B} solving $\hat{F}T_d^h(Nk)$ (resp. $\hat{B}T_d^h(Nk)$) can be transformed to a k -way branching program B solving $FT_d^h(k)$ (resp. $BT_d^h(k)$), where B has no more states than \hat{B} and B is deterministic iff \hat{B} is deterministic. Also for each $d \geq 2$ the decision problem $BT_d(h, k)$ is log space reducible to $\hat{B}T_d(h, k)$ (where h, k are input parameters).*

PROOF. Given an instance I of $FT_d^h(k)$ (or $BT_d^h(k)$) we can find a corresponding instance \hat{I} of $\hat{F}T_d^h(Nk)$ (or $\hat{B}T_d^h(Nk)$) by coding the set of all functions f_i associated with internal nodes i in I by a single function \hat{f} associated with each node of \hat{I} . Here we represent each element of $[Nk]$ by a pair $\langle i, x \rangle$, where $i \in [N]$ represents a node in T_d^h and $x \in [k]$. We want to satisfy the following Claim:

Claim: If a node i has a value x in I then node i has value $\langle i, x \rangle$ in \hat{I} .

Thus if i is a leaf node, then we define the leaf value for node i in \hat{I} to be $\langle i, x \rangle$, where x is the value of leaf i in I .

We define the common internal node function \hat{f} as follows. If nodes i_1, \dots, i_d are the children of node j in T_d^h , then

$$\hat{f}(\langle i_1, x_1 \rangle, \dots, \langle i_d, x_d \rangle) = \langle j, f_j(x_1, \dots, x_d) \rangle \quad (2)$$

The value of \hat{f} is irrelevant (make it $\langle 1, 1 \rangle$) if nodes i_1, \dots, i_d are not the children of j .

An easy induction on the height of a node i shows that the above **Claim** is satisfied.

Note that the value x of the root node 1 in I is easily determined by the value $\langle 1, x \rangle$ of the root in \hat{I} . We specify that the pair $\langle 1, 1 \rangle$ has value 1 in $[Nk]$, so I is a YES instance of the decision problem $BT_d^h(k)$ iff \hat{I} is a YES instance of $\hat{B}T_d^h(Nk)$.

To complete the proof of the last sentence in the theorem we note that the number of bits needed to specify I is $\Theta(Nk^d \log k)$, and the number of bits to specify \hat{I} is dominated by the number to specify \hat{f} , which is $O((Nk)^d \log(Nk))$. Thus the transformation from I to \hat{I} is length-bounded by a polynomial in length of its argument, and it is not hard to see that it can be carried out in log space.

Now we prove the first part of the theorem. Given an Nk -way BP \hat{B} solving $\hat{B}T_d^h(Nk)$ (resp. $\hat{F}T_d^h(Nk)$) we can find a corresponding k -way BP B solving $BT_d^h(k)$ (resp. $FT_d^h(k)$) as follows.

The idea is that on input instance I , B acts like \hat{B} on input \hat{I} . Thus for each state \hat{q} in \hat{B} that queries a leaf node i , the corresponding state q in B queries i , and for each possible answer $x \in [k]$, B has an outedge labelled x corresponding to the edge from \hat{q} labelled $\langle i, x \rangle$. If \hat{q} queries \hat{f} at arguments as in (2) (where i_1, \dots, i_d are the children of node j) then q queries $f_j(x_1, \dots, x_d)$ and for each $x \in [k]$, q has an outedge labelled

³We thank Yann Strozecki, who posed this question.

x corresponding to the edge from \hat{q} labelled $\langle j, x \rangle$. If i_1, \dots, i_d are not the children of j , then the node q is not necessary in B , since the answer to the query is always the default $\langle 1, 1 \rangle$.

In case \hat{B} is solving the function problem $\hat{F}T_d^h(Nk)$ then each output state labelled $\langle 1, x \rangle$ is relabelled x in B (recall that the root of T_d^h is number 1). Any output state q labelled $\langle i, x \rangle$ where $i > 1$ will never be reached in B (since the value of the root node of \hat{I} always has the form $\langle 1, x \rangle$) so q can be deleted. For any edge in \hat{B} leading to q the corresponding edge in B can lead anywhere. \square

Similarly to Theorem 2.5 it is easy to see that the problems $FT_d^h(k)$ and $BT_d^h(k)$ can be reduced to the case that the degree $d = 2$ by increasing the height h by a factor of $\lceil \log_2 d \rceil$ and increasing k to $k' = k^{d'}$, where $d' < d$. The idea is to replace each node v in T_d^h by a binary tree T_v of height $\lceil \log_2 d \rceil$ whose first d leaves correspond to the d children of v . The value of the root of T_v is that of v , and the value of a non-root internal node u of T_v is the tuple of values of the leaves of T_v which are descendants of u . The function f'_v which corresponds to the node v in the binary tree satisfies (assuming d is a power of 2)

$$f'_v(\langle x_1, \dots, x_{d/2} \rangle, \langle x_{d/2+1}, \dots, x_d \rangle) = f_v(x_1, \dots, x_d)$$

The function associated with a non-root internal node of T_v just concatenates tuples with appropriate padding with 1's.

One goal of this paper is to draw attention to the tree evaluation problem and to encourage further attempts at showing $BT_d(h, k) \notin \mathbf{L}$. By the preceding paragraph this is equivalent to showing $BT_2(h, k) \notin \mathbf{L}$, and by Theorem 2.5 this is equivalent to showing $\hat{B}T_d(h, k) \notin \mathbf{L}$. Further our suggested method is to try proving for each fixed h a lower bound of $\Omega(k^{r(h)})$ on the number of states required for a k -way BP to solve $FT_d^h(k)$, where $r(h)$ is any unbounded function (see Corollary 3.3 below). Again according to Theorem 2.5 (since N is a constant) technically speaking we may as well assume that all the node functions in the instance of $FT_d^h(k)$ are the same. However in practice this assumption is not helpful in proving a lower bound. For example Theorem 5.10 states that k^3 states are required for a deterministic k -way BP to solve $FT_2^3(k)$, and the proof assigns three different functions to the three internal nodes of the binary tree of height 3.

2.3. Pebbling

The pebbling game for DAGs (directed acyclic graphs) was defined by Paterson and Hewitt [Paterson and Hewitt 1970] and was used as an abstraction for deterministic Turing machine space in [Cook 1974]. Black-white pebbling was introduced in [Cook and Sethi 1976] as an abstraction of nondeterministic Turing machine space (see [Nordström 2009] for a recent survey).

Here we define and use three versions of the pebbling game for DAGs with one root (i.e. one sink node). The first is a simple ‘black pebbling’ game: A black pebble can be placed on any leaf (i.e. source node), and in general if all children of a node i (where a child of i is a node with an edge to 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 [Cook and Sethi 1976] 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 a new game called *fractional pebbling*, which generalizes whole black-white pebbling by allowing the black and white pebble value of a node to be any real number between 0 and 1. However the total pebble value 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 illustrates two configurations in 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. A black pebble on a node means that the corresponding branching program state knows the value of the node, and a white pebble (applicable to nondeterministic BPs) means that state has a specific conjecture for the value of the node (which must later be verified). A fractional pebble means that the state knows or conjectures that fraction of the $\log k$ bits is the value.

We start by formally defining fractional pebbling, and then define the other two notions as restrictions on fractional pebbling.

DEFINITION 2.6 (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*

$$0 \leq b(i), w(i) \tag{3}$$

$$b(i) + w(i) \leq 1 \tag{4}$$

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 (3), (4)): (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 $w(i)$ to 0, increase $b(i)$ arbitrarily, and simultaneously decrease the black pebble values of the children of i arbitrarily.

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.

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 nondeterministic branching programs can simulate the former, but not the latter. However white sliding moves are a natural dual to black sliding moves and we give a formal definition and examples in Section 4.3.

We use $\#\text{pebbles}(T)$, $\#\text{BWpebbles}(T)$, and $\#\text{FRpebbles}(T)$ 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 Section 4. For example for $d = 2$ we have $\#\text{pebbles}(T_2^h) = h$, $\#\text{BWpebbles}(T_2^h) = \lceil h/2 \rceil + 1$, and $\#\text{FRpebbles}(T_2^h) \leq h/2 + 1$. In particular $\#\text{FRpebbles}(T_2^3) = 2.5$ (see Figure 2).

3. CONNECTING TMS, BPS, AND PEBBLING

Let $FT_d(h, k)$ be the same as $FT_d^h(k)$ except now the inputs vary with both h and k , and we assume the input to $FT_d(h, k)$ is a binary string X which codes h and k and codes each node function f_i for the tree T_d^h by a sequence of k^d binary numbers and each leaf value by a binary number in $[k]$, so X has length

$$|X| = \Theta(d^h k^d \log k) \quad (5)$$

The output is a binary number in $[k]$ giving the value of the root.

The problem $BT_d(h, k)$ is the Boolean version of $FT_d(h, k)$: The input is the same, and the instance is true iff the value of the root is 1.

Obviously $BT_d(h, k)$ and $FT_d(h, k)$ can be solved in polynomial time, but we can prove a stronger result.

THEOREM 3.1. *The problem $BT_d(h, k)$ is in LogDCFL, even when d is given as an input parameter.*

PROOF. By [Sudborough 1978] it suffices to show that $BT_d(h, k)$ is solved by some deterministic auxiliary pushdown automaton M in log space and polynomial time. The algorithm for M is to use its stack to perform a depth-first search of the tree T_d^h , where for each node i it keeps a partial list of the values of the children of i on its stack, until it obtains all d values, at which point it computes the value of i and pops its stack, adding that value to the list for the parent node.

Note that the length n of an input instance is about $d^h k^d \log k$ bits, so $\log n > d \log k$, so M has ample space on its work tape to write all d values of the children of a node i . \square

The best known upper bounds on branching program size for $FT_d^h(k)$ grow as $k^{\Omega(h)}$. The next result shows (Corollary 3.3) that any lower bound with a nontrivial dependency on h in the exponent of k for deterministic (resp. nondeterministic) BP size would separate L (resp. NL) from LogDCFL.

THEOREM 3.2. *For each $d \geq 2$, if $BT_d(h, k)$ is in L (resp. NL) then there is a constant c_d and a function $f_d(h)$ such that $\#\det\text{Fstates}_d^h(k) \leq f_d(h)k^{c_d}$ (resp. $\#\text{ndet}\text{Fstates}_d^h(k) \leq f_d(h)k^{c_d}$) for all $h, k \geq 2$.*

PROOF. By Lemma 2.3 it suffices to prove this for $\#\det\text{Bstates}_d^h(k)$ and $\#\text{ndet}\text{Bstates}_d^h(k)$ instead of $\#\det\text{Fstates}_d^h(k)$ and $\#\text{ndet}\text{Fstates}_d^h(k)$. In general a Turing machine which can enter at most C different configurations on all inputs of a given length n can be simulated (for inputs of length n) by a binary (and hence k -ary) branching program with C states. Each Turing machine using space $O(\log n)$ has at most n^c possible configurations on any input of length $n \geq 2$, for some constant c . By (5) the input for $BT_d(h, k)$ has length $n = \Theta(d^h k^d \log k)$, so there are at most $(d^h k^d \log k)^{c'}$ possible configurations for a log space Turing machine solving $BT_d(h, k)$, for some constant c' . So we can take $f_d(h) = d^{c'h}$ and $c_d = c'(d + 1)$. \square

COROLLARY 3.3. *Fix $d \geq 2$ and any unbounded function $r(h)$. If $\#\det\text{Fstates}_d^h(k) \in \Omega(k^{r(h)})$ then $BT_d(h, k) \notin \text{L}$. If $\#\text{ndet}\text{Fstates}_d^h(k) \in \Omega(k^{r(h)})$ then $BT_d(h, k) \notin \text{NL}$.*

The next result connects pebbling upper bounds with upper bounds for thrifty branching programs.

THEOREM 3.4. *(i) If T_d^h can be black pebbled with p pebbles, then deterministic thrifty branching programs with $O(k^p)$ states can solve $FT_d^h(k)$ and $BT_d^h(k)$.*

(ii) If T_d^h can be fractionally pebbled with p pebbles then nondeterministic thrifty branching programs can solve $BT_d^h(k)$ with $O(k^p)$ states.

PROOF. Consider the sequence C_0, C_1, \dots, C_τ of pebble configurations for a black pebbling of T_d^h using p pebbles. We may as well assume that the root is pebbled in configuration C_τ , since all pebbles could be removed in one more step at no extra cost in pebbles. We design a thrifty branching program B for solving $FT_d^h(k)$ as follows. For each pebble configuration C_t , program B has k^p states; one state for each possible assignment of a value from $[k]$ to each of the p pebbles. Hence B has $O(k^p)$ states, since τ is a constant independent of k . Consider an input I to $FT_d^h(k)$, and let v_i be the value in $[k]$ which I assigns to node i in T_d^h (see Definition 2.1). We design B so that on I the computation of B will be a state sequence $\alpha_0, \alpha_1, \dots, \alpha_\tau$, where the state α_t assigns to each pebble the value v_i of the node i that it is on. (If a pebble is not on any node, then its value is 1.)

For the initial pebble configuration no pebbles have been assigned to nodes, so the initial state of B assigns the value 1 to each pebble. In general if B is in a state α corresponding to configuration C_t , and the next configuration C_{t+1} places a pebble j on node i , then the state α queries the node i to determine v_i , and moves to a new state which assigns v_i to the pebble j and assigns 1 to any pebble which is removed from the tree. Note that if i is an internal node, then all children of i must be pebbled at C_t , so the state α ‘knows’ the values v_{j_1}, \dots, v_{j_d} of the children of i , so α queries $f_i(v_{j_1}, \dots, v_{j_d})$.

When the computation of B reaches a state α_τ corresponding to C_τ , then α_τ determines the value of the root (since C_τ has a pebble on the root), so B moves to a final state corresponding to the value of the root.

The argument for the case of whole black-white pebbling is similar, except now the value for each white pebble represents a guess for the value v_i of the node it is on. If the pebbling algorithm places a white pebble j on a node at some step, then the corresponding state of B nondeterministically moves to any state in which the values of all pebbles except j are the same as before, but the value of j can be any value in $[k]$. If the pebbling algorithm removes a white pebble j from a node i , then the corresponding state has a guess v'_i for the value of i , and either i is a leaf, or all children of i must be pebbled. The corresponding state of B queries i to determine its true value v_i . If $v_i \neq v'_i$ then the computation aborts (i.e. all outedges from the state have label v'_i). Otherwise B assigns j the value 1 and continues.

When B reaches a state α corresponding to a pebble configuration C_t for which the root has a black pebble j , then α knows whether or not the tentative value assigned to the root is 1. All future states remember whether the tentative value is 1. If the computation successfully (without aborting) reaches a state α_τ corresponding to the final pebble configuration C_τ , then B moves to the final state corresponding to output 1 or output 0, depending on whether the tentative root value is 1.

Now we consider the case in which C_0, \dots, C_τ represents a fractional pebbling computation. If $b(i), w(i)$ are the black and white pebbled values of node i in configuration C_t , then a state α of B corresponding to C_t will remember a fraction $b(i) + w(i)$ of the $\log k$ bits specifying the value v_i of the node i , where the fraction $b(i)$ of bits are verified, and the fraction $w(i)$ of bits are conjectured. In general these numbers of bits are not integers, so they are rounded up to the next integer. This rounding introduces at most two extra bits for each node in T_d^h , for a total of at most $2T$ extra bits, where T is the number of nodes in T_d^h . Since the sum over all nodes of all pebble values is at most p , the total number of bits that need to be remembered for a given pebble configuration is at most $p \log k + 2T$, where T is a constant. Associated with each step in the fractional pebbling there are $2^{p \log k + 2T} = O(k^p)$ states in the branching program, one

for each setting of these bits. These bits can be updated for each of the three possible fractional pebbling moves (i), (ii), (iii) in Definition 2.6 in a manner similar to that for whole black-white pebbling.

It is easy to see that in all cases the branching programs described satisfy the thrifty requirement that an internal node is queried only at the correct values for its children (or, in the black-white and fractional cases, the program aborts if an incorrect query is made because of an incorrect guess for the value of a white-pebbled node). \square

COROLLARY 3.5. $\#detFstates_d^h(k) = O(k^{\#pebbles(T_d^h)})$ and $\#ndetFstates_d^h(k) = O(k^{\#FRpebbles(T_d^h)})$.

4. PEBBLING BOUNDS

4.1. Previous results

We start by summarizing what is known about whole black and black-white pebbling numbers as defined at the end of Definition 2.6 (i.e. we allow black sliding moves but not white sliding moves).

The following are minor adaptations of results and techniques that have been known since work of Loui, Meyer auf der Heide and Lengauer-Tarjan [Loui 1979; auf der Heide 1979; Lengauer and Tarjan 1980] in the late '70s. They considered pebbling games where sliding moves were either disallowed or permitted for both black and white pebbles, in contrast to our results below.

We always assume $h \geq 2$ and $d \geq 2$.

THEOREM 4.1. $\#pebbles(T_d^h) = (d-1)h - d + 2$.

PROOF. For $h = 2$ this gives $\#pebbles(T_d^2) = d$, which is obviously correct. In general we show $\#pebbles(T_d^{h+1}) = \#pebbles(T_d^h) + d - 1$, from which the theorem follows.

The following pebbling strategy gives the upper bound: Let the root be node 1 and the children be $2 \dots d+1$. Pebble the nodes $2 \dots d+1$ in order using the optimal number of pebbles for T_d^{h-1} , leaving a black pebble at each node. Note that for the black pebble game, the complexity of pebbling in the game where a pebble remains on the root is the same as for the game where the root has a black pebble on it at some point. The maximum number of pebbles at any point on the tree is $d - 1 + \#pebbles(T_d^{h-1})$. Now slide the black pebble from node 1 to the root, and then remove all pebbles.

For the lower bound, consider the time t at which the children of the root all have black pebbles on them. There must be a final time t' before t at which one of the subtrees rooted at $2, 3, \dots, d+1$ had $\#pebbles(T_d^h)$ pebbles on it. This is because pebbling any of these subtrees requires at least $\#pebbles(T_d^h)$ pebbles, by definition. At time t' , all the other subtrees must have at least 1 black pebble each on them. If not, then there is a subtree T which does not, and it would have to be pebbled before time t , which contradicts the definition of t' . Thus at time t' , there are at least $\#pebbles(T_d^h) + d - 1$ pebbles on the tree. \square

THEOREM 4.2. For $d = 2$ and d odd:

$$\#BWpebbles(T_d^h) = \lceil (d-1)h/2 \rceil + 1 \quad (6)$$

For d even:

$$\#BWpebbles(T_d^h) \leq \lceil (d-1)h/2 \rceil + 1 \quad (7)$$

When d is odd, this number is the same as when white sliding moves are allowed.

PROOF. We divide the proof into three parts: Part I proves (6) when d is odd, Part II proves (7) when d is even (which implies the upper bound for (6) when $d = 2$), and Part III proves the lower bound in (6) when $d = 2$.

Part I:

We show (6) when d is odd.

For $h = 2$ this gives $\#BW\text{pebbles}(T_d^2) = d$, which is obviously correct. In general for odd d we show

$$\#BW\text{pebbles}(T_d^{h+1}) = \#BW\text{pebbles}(T_d^h) + (d-1)/2 \quad (8)$$

from which the theorem follows for this case.

For the upper bound for the left hand side, we strengthen the induction hypothesis by asserting that during the pebbling there is a *critical time* at which the root has a black pebble and there are at most $\#BW\text{pebbles}(T_d^h) - (d-1)/2$ pebbles on the tree (counting the pebble on the root). This can be made true when $h = 2$ by removing all the pebbles on the leaves after the root is pebbled.

To pebble the tree T_d^{h+1} , note that we are allowed $(d-1)/2$ extra pebbles over those required to pebble T_d^h . Start by placing black pebbles on the left-most $(d-1)/2$ children of the root, and removing all other pebbles. Now go through the procedure for pebbling the middle principal subtree, stopping at the critical time, so that there is a black pebble on the middle child of the root and at most $\#BW\text{pebbles}(T_d^h) - (d-1)/2$ pebbles on the middle subtree. Now place white pebbles on the remaining $(d-1)/2$ children of the root, slide a black pebble to the root, and remove all black pebbles on the children of the root. This is the critical time for pebbling T_d^{h+1} : note that there are at most $\#BW\text{pebbles}(T_d^h)$ pebbles on the tree (we removed the black pebble on the root of the middle subtree).

Now remove the pebble on the root and remove all pebbles on the middle subtree by completing its pebbling (keeping the $(d-1)/2$ white pebbles on the children in place). Finally remove the remaining $(d-1)/2$ white pebbles one by one, simply by pebbling each subtree, and removing the white pebble at the root of the subtree instead of black-pebbling it.

To prove the lower bound for the left hand side of (8), we strengthen the induction hypothesis so that now a black-white pebbling allows white sliding moves, and the root may be pebbled by either a black pebble or a white pebble. (Note that for the base case the tree T_d^2 still requires d pebbles.) Consider such a pebbling of T_d^{h+1} which uses as few moves as possible. Consider a time t at which all children of the root have pebbles on them (i.e. just before the root is black pebbled or just after a white pebble on the root is removed). For each child i , let t_i be a time at which the tree rooted at i has $\#BW\text{pebbles}(T_d^h)$ pebbles on it. We may assume

$$t_2 < t_3 < \dots < t_{d+1}$$

Let $m = (d+3)/2$ be the middle child. If $t_m < t$ then each of the $(d-1)/2$ subtrees rooted at i for $i < m$ has at least one pebble on it at time t_m , since otherwise the effort made to place $\#BW\text{pebbles}(T_d^h)$ pebbles on it earlier is wasted. Hence (8) holds for this case. Similarly if $t_m > t$ then each of the $(d-1)/2$ subtrees rooted at i for $i > m$ has at least one pebble on it at time t_m , since otherwise the effort to place T_d^h pebbles on it later is wasted, so again (8) holds.

Part II:

We prove (7) for even degree d :

$$\#BW\text{pebbles}(T_d^h) \leq \lceil (d-1)h/2 \rceil + 1$$

For $h = 2$ the formula gives $\#BW\text{pebbles}(T_d^2) = d$, which is obviously correct. For $h = 3$ the formula gives $\#BW\text{pebbles}(T_d^3) = (3/2)d$, which can be realized by black-pebbling $d/2 + 1$ of the root's children and white-pebbling the rest. In general it suffices to prove the following recurrence:

$$\#BW\text{pebbles}(T_d^{h+2}) \leq \#BW\text{pebbles}(T_d^h) + d - 1 \quad (9)$$

We strengthen the induction hypothesis by asserting that during the pebbling of T_d^h there is a *critical time* at which the root has a black pebble and there are at most $\#BW\text{pebbles}(T_d^h) - (d - 1)$ pebbles on the tree (counting the pebble on the root). This is easy to see when $h = 2$ and $h = 3$.

We prove the recurrence as follows. We want to pebble T_d^{h+2} using $d - 1$ more pebbles than is required to pebble T_d^h . Let us call the children of the root c_1, \dots, c_d . We start by placing black pebbles on $c_1, \dots, c_{d/2}$. We illustrate how to do this by showing how to place a black pebble on $c_{d/2}$ after there are black pebbles on nodes $c_1, \dots, c_{d/2-1}$. At this point we still have $d/2$ extra pebbles left among the original $d - 1$. Let us assign the names c'_1, \dots, c'_d to the children of $c_{d/2}$. Use the $d/2$ extra pebbles to put black pebbles on $c'_1, \dots, c'_{d/2}$. Now run the procedure for pebbling the subtree rooted at $c'_{d/2+1}$ up to the critical time, so there is a black pebble on $c'_{d/2+1}$. Now place white pebbles on the remaining $d/2 - 1$ children of $c_{d/2}$, slide a black pebble up to $c_{d/2}$, remove the remaining black pebbles on the children of $c_{d/2}$, and complete the pebbling procedure for the subtree rooted at $c'_{d/2+1}$, so that subtree has no pebbles. Now remove the white pebbles on the remaining $d/2 - 1$ children of $c_{d/2}$.

At this point there are black pebbles on nodes $c_1, \dots, c_{d/2}$, and no other pebbles on the tree. We now place a black pebble on $c_{d/2+1}$ as follows. Let us assign the names c''_1, \dots, c''_d to the children of $c_{d/2+1}$. Use the remaining $d/2 - 1$ extra pebbles to place black pebbles on $c''_1, \dots, c''_{d/2-1}$. Now run the pebble procedure on the subtree rooted at $c''_{d/2}$ up to the critical time, so $c''_{d/2}$ has a black pebble. Now place white pebbles on the remaining $d/2$ children of $c_{d/2+1}$, slide a black pebble up to $c_{d/2+1}$, remove the remaining black pebbles on the children of $c_{d/2+1}$, place white pebbles on the remaining $d/2 - 1$ children of the root, slide a black pebble up to the root, and remove the remaining black pebbles from the children of the root.

This is now the critical time for the procedure pebbling T_d^{h+2} . There is a black pebble on the root, $d/2 - 1$ white pebbles on the children of the root, $d/2$ white pebbles on the children of $c_{d/2+1}$, and at most $\#BW\text{pebbles}(T_d^h) - d$ pebbles on the subtree rooted at $c''_{d/2}$ (we've removed the black pebble on $c''_{d/2}$), making a total of at most $\#BW\text{pebbles}(T_d^h)$ pebbles on the tree.

Now remove the black pebble from the root and complete the pebble procedure for the subtree rooted at $c''_{d/2}$ to remove all pebbles from that subtree. There remain $d/2 - 1$ white pebbles on the children of the root and $d/2$ white pebbles on the children of $c_{d/2+1}$, making a total of $d - 1$ white pebbles. Now remove each of the white pebbles on the children of $c_{d/2+1}$ by pebbling each of these subtrees in turn. Finally we can remove each of the remaining $d/2 - 1$ white pebbles on the children of the root by a process similar to the one used to place $d/2$ black pebbles on the children of the root at the beginning of the procedure (we now in effect have one more pebble to work with).

Part III:

Finally we give the lower bound for the case $d = 2$:

$$\#BW\text{pebbles}(T_2^h) \geq \lceil h/2 \rceil + 1$$

Clearly 2 pebbles are required for the tree of height 2, and it is easy to show that 3 pebbles are required for the height 3 tree.

In general it suffices to show that the binary tree T of height $h + 2$ requires at least one more pebble than the binary tree of height h . Suppose otherwise, and consider a pebbling of T that uses the minimum number of pebbles required for the tree of height h , and assume that the pebbling is as short as possible. Let t_1 be a time when the root has a black pebble. For $i = 3, 4, 5$ there must be a time t_i when all the pebbles are on the subtree rooted at node i . This is because node i must be pebbled at some point, and if the pebble is white then right after the white pebble is removed we could have placed a black pebble in its place (since we do not allow white sliding moves).

Suppose that $\{t_1, t_3, t_4, t_5\}$ are ordered such that

$$t_{i_1} < t_{i_2} < t_{i_3} < t_{i_4}$$

Then t_1 cannot be either t_{i_3} or t_{i_4} since otherwise at time t_{i_2} there are no pebbles on the subtree rooted at node i_1 and hence its earlier pebbling was wasted (since the root has yet to be pebbled). Similarly if t_1 is either t_{i_1} or t_{i_2} then at time t_{i_3} there are no pebbles on the subtree rooted at i_4 , and since the root has already been pebbled the later pebbling of this subtree is wasted. \square

4.2. Results for fractional pebbling

The concept of fractional pebbling is new. Determining the minimum number p of pebbles required to fractionally pebble T_d^h is important since $O(k^p)$ is the best known upper bound on the number of states required by a nondeterministic BP to solve $FT_d^h(k)$ (see Theorem 3.4). It turns out that proving fractional pebbling lower bounds is much more difficult than proving whole black-white pebbling lower bounds. We are able to get exact fractional pebbling numbers for the binary tree of height 4 and less, but the best general lower bound comes from a nontrivial reduction to a paper by Klawe [Klawe 1985] which proves bounds for the pyramid graph. This bound is within $d/2 + 1$ pebbles of optimal for degree d trees (at most 2 pebbles from optimal for binary trees).

Our proof of the exact value of $\#\text{FRpebbles}(T_2^4) = 3$ led us to conjecture that any nondeterministic BP computing $BT_2^4(k)$ requires $\Omega(k^3)$ states. In section 5 we provide evidence for that conjecture by proving that any nondeterministic *thrifty* BP requires $O(k^3)$ states. The lower bound for height 3 and any degree follows from the lower bound of $\Omega(k^{\frac{3}{2}d - \frac{1}{2}})$ states for nondeterministic branching programs computing $BT_d^3(k)$ (Corollary 5.2).

We start by presenting a general result showing that fractional pebbling can save at most a factor of two over whole black-white pebbling for any DAG (directed acyclic graph). (Here the pebbling rules for a DAG are the same as for a tree, where we require that every sink node (i.e. every ‘root’) must have a whole black pebble at some point.) We will not use this result, but it does provide a simple proof of weaker lower bounds than those given in Theorem 4.4 below.

THEOREM 4.3. *If a DAG D has a fractional pebbling using p pebbles, then it has a black-white pebbling using at most $2p$ pebbles.*

PROOF. Given a sequence P of fractional pebbling moves for a DAG D in which at most p pebbles are used, we define a corresponding sequence P' of pebbling moves in which at most $2p$ pebbles are used. The sequence P' satisfies the following invariant with respect to P .

(♠) A node v has a black pebble (resp. a white pebble) on it at time t with respect to P' iff $b(v) \geq 1/2$ (resp. $w(v) > 1/2$) at time t with respect to P .

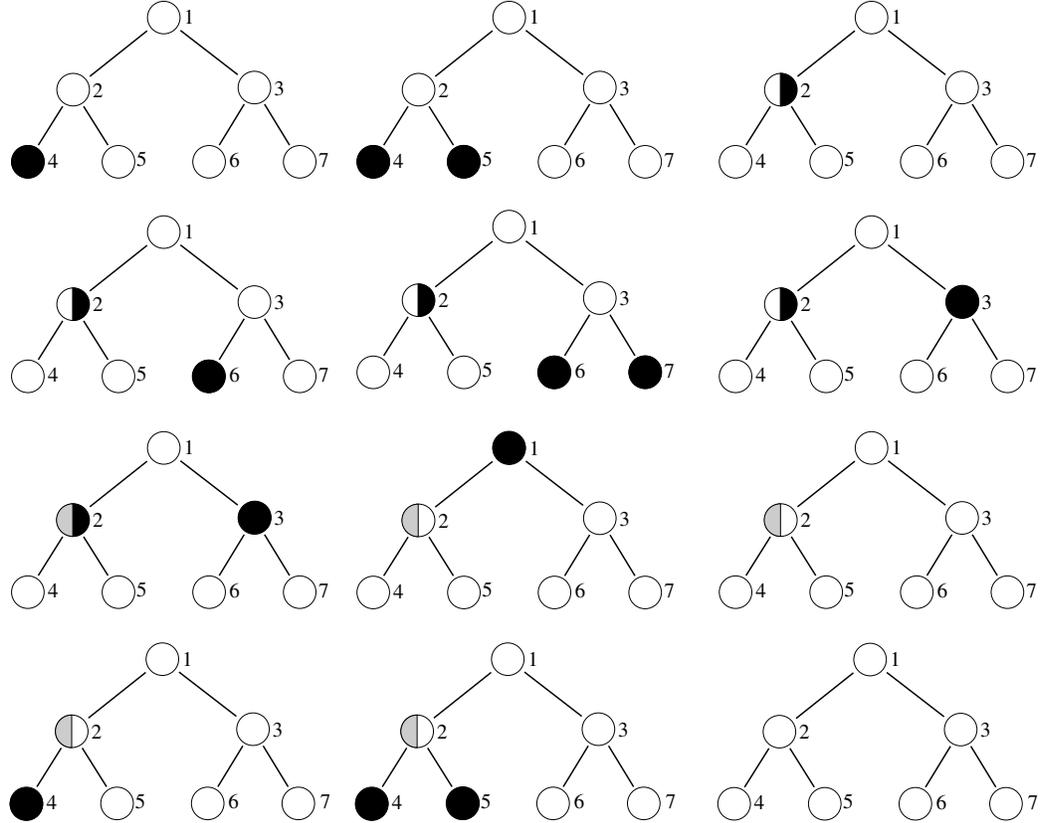


Fig. 2. An optimal fractional pebbling sequence for the height 3 tree using 2.5 pebbles, all configurations included (except the empty starting config). The grey half circle means the *white* value of that node is .5, whereas unshaded area means absence of pebble value. So for example in the seventh 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.

An important consequence of this invariant is that if at time t in P node v satisfies $b(v) + w(v) = 1$ then at time t in P' node v is pebbled.

We describe when a pebble is placed or removed in P' . At the beginning, there are no pebbles on any nodes. P' simulates P as follows. Assume there is a certain configuration of pebbles on D , placed according to P' after time $t - 1$; we describe how P 's move at time t is reflected in P' . If in the current move of P , $b(v)$ (resp. $w(v)$) increases to $1/2$ or greater (resp. greater than $1/2$) for some node v , then the current pebble, if any, on v , is removed and a black pebble (resp. a white pebble) is placed on v in P' . Note that this is always consistent with the pebbling rules. If in the current configuration of P' there is a black (resp. white) pebble on a vertex v , and in the current move of P , $b(v)$ (resp. $w(v)$) falls below $1/2$, then the pebble on v is removed. Again, this is always consistent with the pebbling rules for the black-white pebble game and the fractional black-white pebble game. For all other kinds of moves of P , the configuration in P' does not change.

If P is a valid sequence of fractional pebbling moves, then P' is a valid sequence of pebbling moves. We argue that the cost of P' is at most twice the cost of P , and that if there is a point at which the root has black pebble value 1 with respect to P , then

there is a point at which the root is black-pebbled in P' . These facts together establish the theorem.

To demonstrate these facts, we simply observe that the invariant (\spadesuit) holds by induction on the time t for the simulation we defined. This implies that at any point t , the number of pebbles on D with respect to P' is at most the number of nodes v for which $b(v) + w(v) \geq 1/2$ with respect to P , and is therefore at most twice the total value of pebbles with respect to P at time t . Hence the cost of pebbling D using P' is at most twice the cost of pebbling D using P . Also, if there is a time t at which the root r has black pebble value 1 with respect to P , then $b(r) \geq 1/2$ at time t , so there is a black pebble on r with respect to P' at time t . \square

The next result presents our best-known bounds for fractionally pebbling trees T_d^h .

THEOREM 4.4.

$$(d-1)h/2 - d/2 \leq \#FRpebbles(T_d^h) \leq (d-1)h/2 + 1$$

$$\#FRpebbles(T_d^3) = (3/2)d - 1/2$$

$$\#FRpebbles(T_2^4) = 3$$

We divide the proof into several parts. First we prove the upper bound:

$$\#FRpebbles(T_d^h) \leq (d-1)h/2 + 1$$

PROOF. Let A_h be the algorithm for height $h \geq 2$. It is composed of two parts, B_h and C_h . B_h is run on the empty tree, and finishes with a black pebble on the root and $(d-1)(h-2)$ white half pebbles below the root (and of these $(d-1)(h-3)$ lie below the rightmost child of the root). Next, the black pebble on the root is removed. Then C_h is run on the result, and finishes with the empty tree. B_h and C_h both use $(d-1)h/2 + 1$ pebbles.

A'_h is the same as A_h except that it finishes with a black half pebble on the root. It does this in the most straight-forward way, by leaving a black half pebble after the root is pebbled, and so it uses $(d-1)h/2 + 1.5$ pebbles for all $h \geq 3$.

B_2 : Pebble the tree of height 2 using d black pebbles.

$B_h, h > 2$: Run A'_{h-1} on node 2 using $(d-1)(h-1)/2 + 1.5$ pebbles, and then on node 3 (if $3 \leq d$) using a total of $(d-1)(h-1)/2 + 2$ pebbles (counting the half pebble on node 2), and so on for nodes $2, 3, \dots, d$. So $(d-1)(h-1)/2 + 1 + (d-1)/2 = (d-1)h/2 + 1$ pebbles are used when A'_{h-1} is run on node d . Next run B_{h-1} on node $d+1$, using $(d-1)(h-1)/2 + 1$ pebbles on the subtree rooted at $d+1$, for $(d-1)h/2 + 1$ pebbles in total (counting the black half pebbles on node $2, \dots, d$). The result is a black pebble on node $d+1$, $(d-1)(h-3)$ white half pebbles under $d+1$, and from earlier $d-1$ black half pebbles on $2, \dots, d$, for a total of $(d-1)(h-2)/2 + 1$ pebbles. Add a white half pebble to each of $2, \dots, d$, then slide the black pebble from $d+1$ onto the root. Remove the black half pebbles from $2, \dots, d$. Now there are $(d-1)(h-2)$ white half pebbles under the root, and a black pebble on the root.

C_2 : The tree of height 2 is empty, so return.

C_h : The tree has no black pebbles and $(d-1)(h-2)$ white half pebbles. Note that if a sequence can pebble a tree with p pebbles, then essentially the same sequence can be used to remove a white half pebble from the root with $p + .5$ pebbles. C_h runs C_{h-1} on node $d+1$, resulting in a tree with only a half white pebble on each of $2, \dots, d$. This takes $(d-1)h/2 + 1$ pebbles. Then A_{h-1} is run on each of $2, \dots, d$ in turn, to remove the white half pebbles. The first such call of A_{h-1} is the most expensive, using $(d-1)(h-1)/2 + 1 + (d-1)/2 = (d-1)h/2 + 1$ pebbles. \square

. As noted earlier, the tight lower bound for height 3 and any degree:

$$\#FRpebbles(T_d^3) \geq (3/2)d - 1/2$$

follows from the asymptotically tight lower bound of $\Omega(k^{\frac{3}{2}d - \frac{1}{2}})$ states for nondeterministic branching programs computing $BT_d^3(k)$ (Corollary 5.2). We do, however, have a direct proof of $\#FRpebbles(T_2^3) \geq 5/2$:

PROOF. Assume to the contrary that there is a fractional pebbling with fewer than 2.5 pebbles. It follows that no non-leaf node i can ever have $w(i) \geq 0.5$, since the children of i must each have pebble value 1 in order to decrease $w(i)$. Since there must be some time t_1 during the pebbling sequence such that both the nodes 2 and 3 (the two children of the root) have pebble value 1, it follows that at time t_1 , $b(2) > 0.5$ and $b(3) > 0.5$. Hence for $i = 2, 3$ there is a largest $t_i \leq t_1$ such that node i is black-pebbled at time t_i and $b(i) > 0.5$ during the time interval $[t_i, t_1]$. (By ‘black-pebbled’ we mean at time $t_i - 1$ both children of i have pebble value 1, so that at time t_i the value of $b(i)$ can be increased.)

Assume w.l.o.g. that $t_2 < t_3$. Then at time $t_3 - 1$ both children of node 3 have pebble value 1 and $b(2) > 0.5$, so the total pebble value exceeds 2.5. \square

. Before we prove the lower bound for all heights, which we do not believe is tight, we prove one more tight lower bound:

$$\#FRpebbles(T_2^4) \geq 3$$

PROOF. Let C_0, C_1, \dots, C_m be the sequence of pebble configurations in a fractional pebbling of the binary tree of height 4. We say that C_t is the configuration at time t . Thus C_0 and C_m have no pebbles, and there is a first time t_1 such that C_{t_1+1} has a black pebble on the root. In general we say that step t in the pebbling is the move from C_t to C_{t+1} . In particular, if an internal node i is black-pebbled at step t then both children of i have pebble value 1 in C_t and node i has a positive black pebble value in C_{t+1} .

Note that if any configuration C_t has a whole white pebble on some internal node then both children must have pebble value 1 to remove that pebble, so some configuration will have at least pebble value 3, which is what we are to prove. Hence we may assume that no node in any C_t has white pebble value 1, and hence every node must be black-pebbled at some step.

For each node i we associate a critical time t_i such that i is black-pebbled at step t_i and hence the children of i each have pebble value 1 in configuration C_{t_i} . The time t_1 associated with the root (as above) is the first step at which the root is black-pebbled, and hence nodes 2 and 3 each have pebble value 1 in C_{t_1} . In general if t_i is the critical time for internal node i , and j is a child of i , then the critical time t_j for j is the largest $t < t_i$ such that j is black-pebbled at step t .

Sibling Assumption: We may assume w.l.o.g. (by applying an isomorphism to the tree) that if i and j are siblings and $i < j$ then $t_i < t_j$.

In general the critical times for a path from root to leaf form a descending chain. In particular

$$t_7 < t_3 < t_1$$

For each $i > 1$ we define b_i and w_i to be the black and white pebble values of node i at the critical time of its parent. Thus for all $i > 1$

$$b_i + w_i = 1 \tag{10}$$

Now let p be the maximum pebble value of any configuration C_t in the pebbling. Our task is to prove that $p \geq 3$

After the critical time of an internal node i the white pebble values of its two children must be removed. When the first one is removed both white values are present along with pebble value 1 on two children, so

$$w_{2i} + w_{2i+1} + 2 \leq p$$

In particular for $i = 1, 3$ we have

$$w_2 + w_3 + 2 \leq p \quad (11)$$

$$w_6 + w_7 + 2 \leq p \quad (12)$$

Now we consider two cases, depending on the order of t_2 and t_7 .

CASE I: $t_2 < t_7$

Then by the Sibling Assumption, at time t_7 (when node 7 is black-pebbled) we have

$$b_2 + b_6 + 2 \leq p \quad (13)$$

Now if we also suppose that w_6 is not removed until after t_1 (CASE IA) then when the first of w_2, w_6 is removed we have

$$w_2 + w_6 + 2 \leq p$$

so adding this equation with (13) and using (10) we see that $p \geq 3$ as required.

However if we suppose that w_6 is removed before t_1 (CASE IB) (but necessarily after $t_2 < t_3$) then we have

$$b_2 + b_3 + w_6 + 2 \leq p$$

then we can add this to (11) to again obtain $p \geq 3$.

CASE II: $t_7 < t_2$

Then $t_6 < t_7 < t_2 < t_3$ so at time t_2 we have

$$b_6 + b_7 + 2 \leq p$$

so adding this to (12) we again obtain $p \geq 3$. \square

. To prove the general lower bound, we need the following lemma:

LEMMA 4.5. *For every finite DAG there is an optimal fractional 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 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, $b_{v,t} + w_{v,t} \leq 1$, 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). \square

. Now we can prove the lower bound for all heights:

$$\#FRpebbles(T_d^h) \geq (d-1)(h-1)/2 - 1/2 = (d-1)h/2 - d/2 \quad (14)$$

Remark 4.6. We conjecture that the upper bound given in Theorem 4.4 is tight. It *seems* like proving this should not be much harder than proving the lower bound for black-white pebbling T_d^h . However we have not even been able to prove the weaker lower bound (14) directly. The present proof derives the lower bound from Klawe’s result (Theorem 4.8).

PROOF. The degree d and height h of the tree are fixed throughout this proof, so we will write just T instead of T_d^h .

The high-level strategy for the proof is as follows. We transform T into a DAG G such that a lower bound for $\#BWpebbles(G)$ gives a lower bound for $\#FRpebbles(T)$. To analyze $\#BWpebbles(G)$, we use a result by Klawe [Klawe 1985], who shows that for any DAG H that satisfies a certain “niceness” property (Definition 4.12), $\#BWpebbles(H)$ can be given in terms of $\#pebbles(H)$ (and the relationship is tight to within an additive constant less than one). This helps since the black pebbling cost is typically easier to analyze. In our case, G does not satisfy the niceness property as-is, but just by removing some edges from G (which cannot *increase* the black or black/white pebbling cost), we get a new DAG G' that is nice. We then compute $\#pebbles(G')$ exactly, which by Klawe’s result yields a lower bound on $\#BWpebbles(G') \leq \#BWpebbles(G)$, and hence on $\#FRpebbles(T)$.

We first motivate the construction G and show that the whole black-white pebbling number of G is related to the fractional pebbling number of T .

We will use Lemma 4.5 to show that the following “discretized” fractional pebbling cost is almost the same as the fractional pebbling cost $\#FRpebbles$ when the parameter c is large enough:

DEFINITION 4.7 (DISCRETIZED FRACTIONAL PEBBLING). *For positive integer c , let $\#FRpebbles_c(H)$ be the cost of fractionally pebbling H when only the following moves are allowed:*

- *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 $1/c$.*

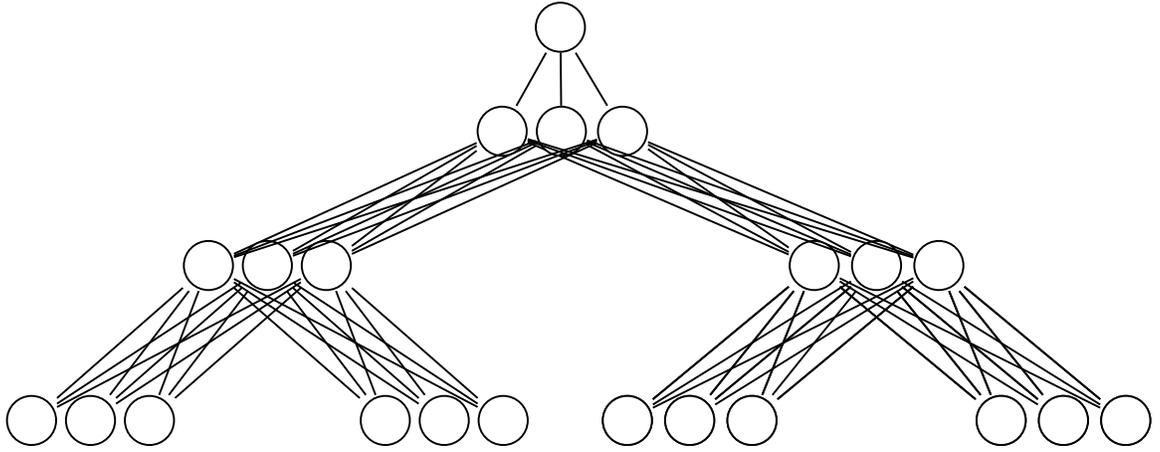
By Lemma 4.5, we can assume all pebble values are rational, and if we choose c large enough it is not a restriction that pebble values can only be changed by $1/c$. Sliding moves are not allowed in the discretized game, but it is easy to see that increases the cost by at most 1 (compared to fractional pebbling with black sliding moves). Hence we have:

FACT 2. $\#FRpebbles(T) \geq \#FRpebbles_c(T) - 1$ for sufficiently large c

Now let c be an arbitrary positive integer. We show how to construct $G = G_c$.⁴ We will split up each node of T into c nodes, so that the discretized fractional pebbling game on T corresponds to the whole black-white pebbling game on G .⁵ 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 .

⁴We don’t write the subscript since c is fixed throughout the argument.

⁵For an example, see figure 3.

Fig. 3. G for the height 3 binary tree with $c = 3$

The idea is to use c whole-pebble-taking nodes to “simulate” each fractional-pebble-taking node of T . For example, if $c = 20$ and we have a configuration of T where node u has black value $4/20$ and white value $6/20$, then in the corresponding configuration of G , 4 (resp. 6) of the 20 nodes dedicated to simulating u are whole black (resp. white) pebbled. More precisely, in place of each node v of T , G has c nodes $v[1], \dots, v[c]$; having any $c' \leq c$ of those nodes pebbled simulates v having value c'/c in the discretized fractional pebbling game. In place of each edge (u, v) of T is a copy of the complete bipartite graph (U, V) , where U contains nodes $u[1] \dots u[c]$ and V contains nodes $v[1] \dots v[c]$. To clarify: if u is 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 .⁶

So every node in G at height $h - 1$ and lower has c parents, and every internal (i.e. non-leaf) node except for the root has dc children. By construction we get:⁷

FACT 3. $\#FRpebbles_c(T) \geq (\#BWpebbles(G) - 1)/c$

From Facts 2 and 3, our goal follows if we can show

$$\#BWpebbles(G) \geq c((d - 1)(h - 1) + 1)/2 + 1$$

For that we will use Theorem 4.8 (from [Klawe 1985]), stated below. The statement of the theorem depends on Klawe’s definition of *nice* DAGs (Definition 4.12), which is stated later when we finally get around to proving that a DAG is nice (Proposition 4.12.1).

THEOREM 4.8 ([KLAWE 1985]). *If H is a nice DAG, then*

$$\#BWpebbles(H) \geq \lfloor \#pebbles(H)/2 \rfloor + 1$$

⁶The 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 , are pebbled. This affects the relationship between $\#FRpebbles_c(T)$ and $\#BWpebbles(G)$ very slightly, as indicated by Fact 3.

⁷For clarification, we note that $\#BWpebbles(G)/c \geq \#FRpebbles_c(T)$.

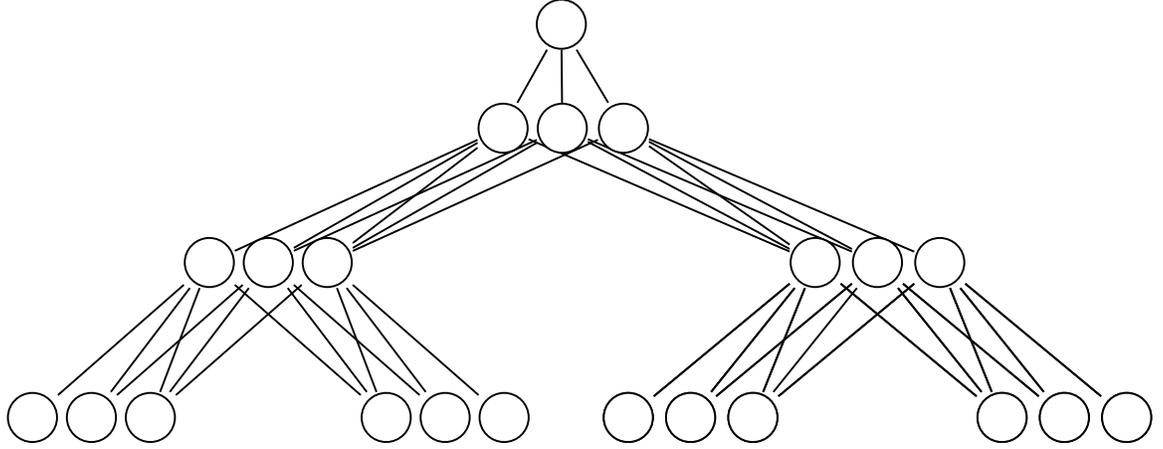


Fig. 4. G' for the height 3 binary tree with $c = 3$

G is not nice in Klawe's sense. We will delete some edges from G to produce a nice DAG G' and then we will analyze $\#pebbles(G')$. Note that deleting edges cannot increase the black-white pebbling cost, and so we have:

FACT 4. $\#BWpebbles(G') \leq \#BWpebbles(G)$

The following definition will help in explaining the construction of G' as well as for specifying and proving properties of certain paths.

DEFINITION 4.9. For $u \in G$, let $T(u)$ be the node in T from which u was generated (i.e. $T(u)[i] = u$ for some $i \leq c$). For $v, v' \in T$, we say $v <_T v'$ if v is visited before v' in an inorder traversal of T . For $u, u' \in G$, we say $u <_G u'$ if $T(u) <_T T(u')$ or if for some $v \in T, i < j \leq c$ have $u = v[i], u' = v[j]$.

G' is obtained from G 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 $v[1], v[2], \dots, v[c]$ of G . Remove the edges from $v[i]$ to its $i - 1$ smallest and $c - i$ largest children. So in the end each internal node of G' except the root has $c(d - 1) + 1$ children.

By Theorem 4.8 (with $H = G'$) and Fact 4, it now remains to lower bound $\#pebbles(G')$ (Proposition 4.9.1 with c even), and then show that G' is nice (Proposition 4.12.1).

PROPOSITION 4.9.1. $\#pebbles(G') = c((d - 1)(h - 1) + 1)$

It will be convenient to rewrite the expression as $(c - 1) + c(d - 1)(h - 1) + 1$. The upper bound is attained using a simple recursive algorithm similar to that used for T_d^h (Theorem 4.1).

For the lower bound, consider the earliest time t when all paths from a leaf to the root are blocked (a path is blocked if at least one of its nodes is pebbled). 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_{BN} be a newly-blocked

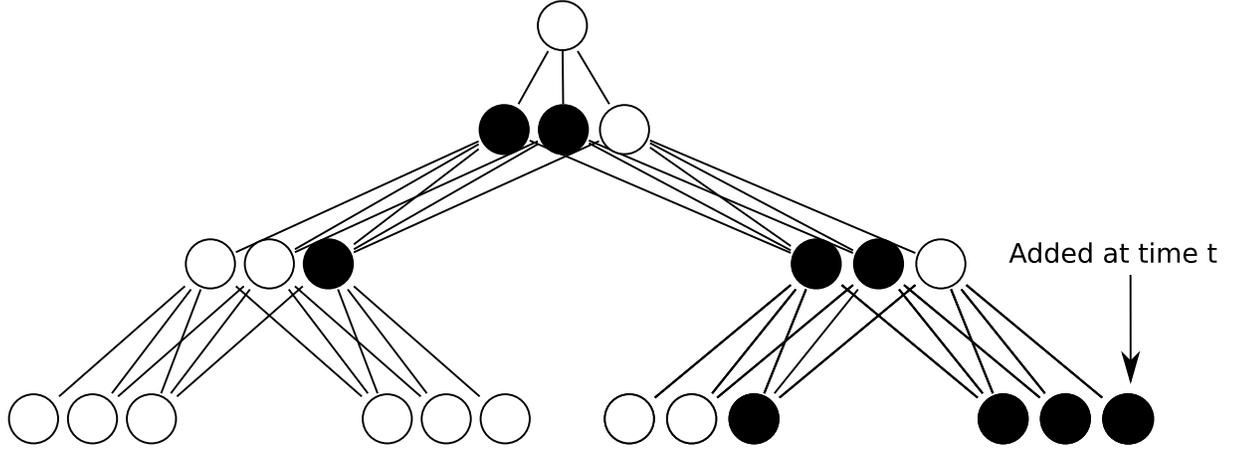


Fig. 5. A possible black pebbling bottleneck of G' for the height 3 binary tree with $c = 3$

path from a leaf to the root (the *BottleNeck* path). Consider the set

$$S = \{u \in G' \mid u \text{ is a child of a node in } P_{\text{BN}} \text{ and } u \notin P_{\text{BN}}\}$$

of size $(c - 1) + c(d - 1)(h - 1)$.⁸

CLAIM 1. *There is a set of pairwise node-disjoint paths $\{P_u\}_{u \in S}$ such that for every $u \in S$, P_u is a path from a leaf to u and P_u does not intersect P_{BN} .*

Assuming Claim 1 holds, we get that at time $t - 1$, for every $u \in S$ there must be at least one pebble on 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 time t gives $(c - 1) + c(d - 1)(h - 1) + 1$ pebbles. Hence, to complete the proof of Proposition 4.9.1, it just remains to prove Claim 1, and for that task we will benefit from a couple more simple definitions:

DEFINITION 4.10. *For $u \in G'$, the left-most (resp. right-most) path to u is the unique path from some leaf to u that is determined, starting at u , by moving to the smallest (resp. largest) child at every level.⁹*

DEFINITION 4.11. *For any path P from a leaf to a height l node u , for $l' \leq l$ let $P(l')$ be the height l' node on P .*

Recall the ordering on the nodes of G (which we extend to G') from Definition 4.9. For each $u \in S$ at height l , if u is less than (resp. greater than) $P_{\text{BN}}(l)$ then make P_u the left-most (resp. right-most) path to u . Intuitively, we are choosing P_u so that it moves away from P_{BN} as quickly as possible. Now we need to show that the paths $\{P_u\}_{u \in S} \cup \{P_{\text{BN}}\}$ are pairwise node-disjoint. The following fact is clear from the definition of G' .

⁸ S has $c - 1$ nodes at height h and $c(d - 1)$ nodes at heights $1, \dots, h - 1$. See Figure 5, where the pebbled nodes are the nodes in S plus one leaf node not in S .

⁹Equivalently: if u is at height l then the left-most (resp. right-most) path to u is the length l path u_1, \dots, u_l such that $u_l = u$ and u_i is the smallest (resp. largest) child of u_{i+1} for all $i \in \{1, \dots, l - 1\}$.

FACT 5. *For any $u, v \in G'$, if $u < v$ then the smallest child of u is not a child of v , and the largest child of v is not a child of u .*

First we show that P_u and P_{BN} are node-disjoint for every $u \in S$. The following lemma will help now and in the proof of Proposition 4.12.1.

LEMMA 4.11.1. *For $u, v \in G'$ 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 Fact 5 $P'_u(l-1) \neq P'_v(l-1)$, a contradiction. The proof for the second part of the lemma is similar. \square

That P_u and P_{BN} are disjoint follows from using Lemma 4.11.1 on u and the sibling of u in P_{BN} .

Next we show that for distinct $u, v \in S$, the paths P_u and P_v do not intersect. Let us first show that P_u does not contain v , and by symmetry we will have that P_v does not contain u . Suppose for the sake of contradiction that P_u contains v , and WLOG assume P_u is the left-most path to u (the other case is symmetric). Since $u \neq v$, there must be a height $l \leq \text{height}(u)$ such that $P_u(l-1) = v$ and $P_u(l)$ is a parent of v . From the definition of S , we know $P_{\text{BN}}(l)$ is also a parent of v . Since we assumed P_u is the left-most path to u , it must be that $P_u(l) < P_{\text{BN}}(l)$. But then Fact 5 tells us that v cannot be a child of $P_{\text{BN}}(l)$, a contradiction. So we have shown that P_v does not contain u , and by symmetry P_u does not contain v . Now suppose that P_u and P_v intersect at some node other than u or v . Then there is a height l , one greater than the height where they first intersect, such that $P_u(l) \neq P_v(l)$. Now, observe that P_u and P_v are both left-most paths or both right-most paths, since otherwise in order for them to intersect they would need to cross P_{BN} (which we showed does not happen). But then from Fact 5 $P_u(l-1) \neq P_v(l-1)$, a contradiction.

That completes the proof of Claim 1 and hence of Proposition 4.9.1. We now just need to prove Proposition 4.12.1 and then apply Klawe's Theorem 4.8.

DEFINITION 4.12. *A DAG H is nice if the following conditions hold:*

- (1) *If u_1 and u_2 are sibling nodes¹⁰ in H 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 siblings, then there is no path from u_1 to u_2 or from u_2 to u_1 .*
- (3) *If u, u_1, \dots, u_m are nodes none of which has a path to any of the others, then there are node-disjoint paths P_1, \dots, P_m such that P_i is a path from a leaf to u_i and there is no path between u and any node in P_i .*

PROPOSITION 4.12.1. *G' is nice.*

PROOF. Property 2 is obviously satisfied.

For property 1, the argument used to give the black pebbling lower bound of $(c-1) + c(d-1)(h-1) + 1$ can be used to give a lower bound of $c(d-1)(h'-1) + 1$ for the cost of black pebbling any node at height $h' \leq h$.¹¹ Moreover that bound is easily shown to be tight.

¹⁰i.e. they have a parent in common

¹¹The only change is in the size of the set of nodes S . For the root, which is at height $h+1$, S has size $(c-1) + c(d-1)(h-1)$, whereas for a height $h' \leq h$ node, S has size $c(d-1)(h'-1)$.

For property 3, we can choose P_i to be the left-most (resp. right-most) path from u_i if u_i is less than (resp. greater than) u . We then use Lemma 4.11.1 on each pair of nodes in $\{u, u_1, \dots, u_m\}$. \square

4.3. White sliding moves

In the definition of fractional pebbling (Definition 2.6) we allow black sliding moves but not white sliding moves. To allow white sliding moves we would add a clause

(4) *For every internal node i , decrease $w(i)$ to 0 and increase the white pebble value of each child of i so that each child has total pebble value 1.*

We did not include this move in the original definition because a nondeterministic k -way BP solving $FT_d^h(k)$ or $BT_d^h(k)$ does not naturally simulate it. The natural way to simulate such a move would be to verify the conjectured value of node i (conjectured when the white pebble was placed on i) by comparing it with $f_i(v_{j_1}, \dots, v_{j_d})$, where j_1, \dots, j_d are the children of i . But this would require the BP to remember a $(d + 1)$ -tuple of values, whereas potentially only d pebbles are involved.

White sliding moves definitely reduce the number of pebbles required to pebble some trees. For example the binary tree T_2^3 can easily be pebbled with 2 pebbles using white sliding moves, but requires 2.5 pebbles without (Theorem 4.4). The next result shows that $8/3$ pebbles suffice for pebbling T_2^4 with white sliding moves, whereas 3 pebbles are required without (Theorem 4.4).

THEOREM 4.13. *The binary tree of height 4 can be pebbled with $8/3$ pebbles using white sliding moves.*

PROOF. The height 3 binary tree can be pebbled with 2 pebbles. Use that sequence on node 2, but leave one third of a black pebble on node 2. That takes $7/3$ pebbles. Put black pebbles on nodes 12 and 13. Slide one third of a black pebble up to node 6. Remove the pebbles on nodes 12 and 13. Put black pebbles on nodes 14 and 15 – this is the first configuration with $8/3$ pebbles. Slide the pebble on node 14 up to node 7. Remove the pebble from 15. Put $2/3$ of a white pebble on node 6. Slide the black pebble on node 7 up to node 3. Remove one third of a black pebble from node 6. Put $2/3$ of a white pebble on node 2 – the resulting configuration has $8/3$ pebbles. Slide the black pebble on node 3 up to the root. Remove all black pebbles. At this point there is $2/3$ of a white pebble on both node 2 and node 6. Put a black pebble on node 12 and one third of a black pebble on node 13 – another bottleneck. Slide the $2/3$ white pebble on node 6 down to node 13. Remove the pebbles from nodes 12 and 13. Finally, use $8/3$ pebbles to remove the $2/3$ white pebble from node 2. \square

5. BRANCHING PROGRAM BOUNDS

In this section we prove tight bounds (up to a constant factor) for the number of states required for both deterministic and nondeterministic k -way branching programs to solve the Boolean problems $BT_d^h(k)$ for all trees of height $h = 2$ and $h = 3$. (The bound is obviously $\Theta(k^d)$ for trees of height 2, because there are $d + k^d$ input variables.) For every height $h \geq 2$ we prove upper bounds for deterministic *thrifty* programs which solve $FT_d^h(k)$ (Theorem 5.1, (15)), and show that these bounds are optimal for degree $d = 2$ even for the Boolean problem $BT_d^h(k)$ (Theorem 5.11). We prove upper bounds for nondeterministic thrifty programs solving $BT_d^h(k)$ in general, and show that these are optimal for binary trees of height 4 or less (Theorems 5.1 and 5.15).

For the nondeterministic case our best BP upper bounds for every $h \geq 2$ come from fractional pebbling algorithms via Theorem 3.4. For the deterministic case our best bounds for the function problem $FT_d^h(k)$ come from black pebbling via the same theo-

rem, although we can improve on them for the Boolean problem $BT_2^h(k)$ by a factor of $\log k$ (for $h \geq 3$).

THEOREM 5.1 (BP UPPER BOUNDS). *For all $h, d \geq 2$*

$$\#\text{detFstates}_d^h(k) = O(k^{(d-1)h-d+2}) \quad (15)$$

$$\#\text{detBstates}_d^h(k) = O(k^{(d-1)h-d+2}/\log k), \text{ for } h \geq 3 \quad (16)$$

$$\#\text{ndetBstates}_d^h(k) = O(k^{(d-1)(h/2)+1}) \quad (17)$$

The first and third bounds are realized by thrifty programs.

PROOF. The first and third bounds follow from Theorem 3.4 (which states that pebbling upper bounds give rise to upper bounds for the size of thrifty BPs) and from Theorems 4.1 and 4.4 (which give the required pebbling upper bounds).

To prove (16) we use a branching program which implements the algorithm below. Here we have a parameter m , and choosing $m = \lceil \log k^{d-1} - \log \log k^{d-1} \rceil$ suffices to show $\#\text{detBstates}_d^h(k) = O(k^{(d-1)(h-1)+1}/\log k^{d-1})$, from which (16) follows. We estimate the number of states required up to a constant factor.

1) Compute v_2 (the value of node 2 in the heap ordering), using the black pebbling algorithm for the principal left subtree. This requires $k^{(d-1)(h-2)+1}$ states. Divide the k possible values for v_2 into $\lceil k/m \rceil$ blocks of size m .

2) Remember the block number for v_2 , and compute v_3, \dots, v_{d+1} . This requires $k/m \times k^{d-2} \times k^{(d-1)(h-2)+1} = k^{(d-1)(h-1)+1}/m$ states.

3) Remember v_3, \dots, v_{d+1} and the block number for v_2 . Compute $f_1(a, v_3, \dots, v_{d+1})$ for each of the m possible values a for v_2 in its block number, and keep track of the set of a 's for which $f_1 = 1$. This requires $k^{d-1} \times k/m \times m \times 2^m = k^d 2^m$ states.

4) Remember just the set of possible a 's (within its block) from above (there are 2^m possibilities). Compute v_2 again and accept or reject depending on whether v_2 is in the subset. This requires $k^{(d-1)(h-2)+1} 2^m$ states.

The total number of states has order the maximum of $k^{(d-1)(h-1)+1}/m$ and $k^{(d-1)(h-2)+1} 2^m$, which is at most

$$k^{(d-1)(h-1)+1}/(\log k^{d-1} - \log \log k^{d-1})$$

for $m = \log k^{d-1} - \log \log k^{d-1}$. \square

We combine the above upper bounds with the Nečiporuk lower bounds in Subsection 5.1, Figure 6, to obtain the following.

COROLLARY 5.2 (TIGHT BOUNDS FOR HEIGHT 3 TREES). *For all $d \geq 2$*

$$\#\text{detFstates}_d^3(k) = \Theta(k^{2d-1})$$

$$\#\text{detBstates}_d^3(k) = \Theta(k^{2d-1}/\log k)$$

$$\#\text{ndetBstates}_d^3(k) = \Theta(k^{(3/2)d-1/2})$$

5.1. The Nečiporuk method

By applying the Nečiporuk method to a k -way branching program B computing a function $f : [k]^m \rightarrow R$, we mean the following well known steps [Nečiporuk 1966] (see [Wegener 2000]):

(1) Upper bound the number $N(s, v)$ of (syntactically) distinct branching programs of type B having s non-final states, each labelled by one of v variables.

Model	Lower bound for $FT_d^h(k)$	Lower bound for $BT_d^h(k)$
Deterministic k -way branching program	$\frac{d^{h-2}-1}{4(d-1)^2} \cdot k^{2d-1}$	$\frac{d^{h-2}-1}{3(d-1)^2} \cdot \frac{k^{2d-1}}{\log k}$
Deterministic binary branching program	$\frac{d^{h-2}-1}{5(d-1)^2} \cdot k^{2d} = \Omega(n^2/(\log n)^2)$	$\frac{d^{h-2}-1}{4d(d-1)} \cdot \frac{k^{2d}}{\log k} = \Omega(n^2/(\log n)^3)$
Nondeterministic k -way BP	$\frac{d^{h-2}-1}{2d-2} \cdot k^{\frac{3d}{2}-\frac{1}{2}} \sqrt{\log k}$	$\frac{d^{h-2}-1}{2d-2} \cdot k^{\frac{3d}{2}-\frac{1}{2}}$
Nondeterministic binary BP	$\frac{d^{h-2}-1}{2d-2} \cdot k^{\frac{3d}{2}} \sqrt{\log k} = \Omega(n^{3/2}/\log n)$	$\frac{d^{h-2}-1}{2d-2} \cdot k^{\frac{3d}{2}} = \Omega(n^{3/2}/(\log n)^{3/2})$

Fig. 6. Size bounds for k large enough, expressed in terms of $n = \Theta(k^d \log k)$ in the binary cases, obtained by applying the Nečiporuk method. Rectangles indicate optimality in k when $h = 3$ (Cor. 5.2). Improving any entry to $\Omega(k^{\text{unbounded } f(h)})$ would prove $\mathbf{L} \subsetneq \mathbf{P}$ (Cor. 3.3).

- (2) Pick a partition $\{V_1, \dots, V_p\}$ of $[m]$.
- (3) For $1 \leq i \leq p$, lower bound the number $r_{V_i}(f)$ of restrictions $f_{V_i} : [k]^{|V_i|} \rightarrow R$ of f obtainable by fixing values of the variables in $[m] \setminus V_i$.
- (4) Then $\text{size}(B) \geq |R| + \sum_{1 \leq i \leq p} s_i$, where $s_i = \min\{s : N(s, |V_i|) \geq r_{V_i}(f)\}$.

The Nečiporuk method still yields the strongest explicit binary branching program size lower bounds known today, namely $\Omega(\frac{n^2}{(\log n)^2})$ for the deterministic case [Nečiporuk 1966] and $\Omega(\frac{n^{3/2}}{\log n})$ for the nondeterministic case ([Pudlák 1987], see [Razborov 1991]). It is known that the above lower bounds are the best that can be obtained using the Nečiporuk method. For the deterministic case this is stated with proof hints in [Wegener 1987, P. 422]. An argument for the nondeterministic case is made in [Beame and McKenzie].

THEOREM 5.3. *Applying the Nečiporuk method yields Figure 6.*

Remark 5.4. Our $\Omega(n^{3/2}/(\log n)^{3/2})$ binary nondeterministic BP lower bound for the $BT_d^h(k)$ problem and in particular for $BT_2^3(k)$ applies to BP “state-size” defined here as the number of *states* in the BP. By comparison, Pudlák’s $\Omega(n^{3/2}/\log n)$ lower bound [Pudlák 1987; Razborov 1991] (for a different Boolean function) applies to the “edge-size” of the closely related switching and rectifier network model, where “edge-size” is defined as the number of (labelled) *edges* in the network. Because switching and rectifier networks can also use unlabelled edges, any k -way nondeterministic BP with state-size S can be simulated by a network of edge-size at most kS (regardless of the BP outdegree). Pudlák’s $\Omega(n^{3/2}/\log n)$ bound thus applies as well to the number of states in a binary nondeterministic BP computing his function, and his bound is the best that the Nečiporuk method can achieve [Beame and McKenzie].

PROOF OF THEOREM 5.3. We have $N_{\text{det}}^{k\text{-way}}(s, v) \leq v^s \cdot (s + |R|)^{sk}$ for the number of deterministic BPs and $N_{\text{ndet}}^{k\text{-way}}(s, v) \leq v^s \cdot (|R| + 1)^{sk} \cdot (2^s)^{sk}$ for nondeterministic BPs having s non-final states, each labelled with one of v variables. To see the latter bound, note that edges labelled $i \in [k]$ can connect a state S to zero or one state among the final states and can connect S independently to any number of states among the non-final states.

The only decision to make when applying the Nečiporuk method is the choice of the partition of the input variables. Here every entry in Figure 6 is obtained using

the same partition (with the understanding that a k -ary variable in the partition is replaced by $\log k$ binary variables when we treat 2-way branching programs).

We will only partition the set V of k -ary $FT_d^h(k)$ or $BT_d^h(k)$ variables that pertain to internal tree nodes other than the root (we will neglect the root and leaf variables). Each internal tree node has $d - 1$ siblings and each sibling involves k^d variables. By a *litter* we will mean any set of d k -ary variables that pertain to precisely d such siblings. We obtain our partition by writing V as a union of

$$k^d \cdot \sum_{i=0}^{h-3} d^i = k^d \cdot \frac{d^{h-2} - 1}{d - 1}$$

litters. (Specifically, each litter can be defined as

$$\{f_i(j_1, j_2, \dots, j_d), f_{i+1}(j_1, j_2, \dots, j_d), \dots, f_{i+d-1}(j_1, j_2, \dots, j_d)\}$$

for some $1 \leq j_1, j_2, \dots, j_d \leq k$ and some d siblings $i, i + 1, \dots, i + d - 1$.)

Consider such a litter L . We claim that $|R|^{k^d}$ distinct functions $f_L : [k]^d \rightarrow R$ can be induced by setting the variables outside of L , where $|R| = k$ in the case of $FT_d^h(k)$ and $|R| = 2$ in the case of $BT_d^h(k)$. Indeed, to induce any such function, fix the “descendants of the litter L ” to make each variable in L relevant to the output; then, set the variables pertaining to the immediate ancestor node ν of the siblings forming L to the appropriate k^d values, as if those were the final output desired; finally, set all the remaining variables in a way such that the values in ν percolate from ν to the root.

It remains to do the calculations. We illustrate two cases. Similar calculations yield the other entries in Figure 6.

Nondeterministic k -way branching programs computing $FT_d^h(k)$. Here $|R| = k$. In a correct program, the number s of states querying one of the d litter L variables must satisfy

$$k^{k^d} \leq N_{\text{nondet}}^{k\text{-way}}(s, d) \leq d^s \cdot (k + 1)^{sk} \cdot (2^s)^{sk} \leq s^s \cdot k^{2sk} \cdot (2^s)^{sk}$$

since $d \leq s$ (because $FT_d^h(k)$ depends on all its variables), and thus

$$k^d \log k \leq s(\log s + 2k \log k) + s^2 k.$$

Suppose to the contrary that $s < (k^{\frac{d-1}{2}} \sqrt{\log k})/2$. Then

$$s(\log s + 2k \log k) + s^2 k < s\left(\frac{d-1}{2} \log k + \frac{\log \log k}{2} + 2k \log k\right) + s^2 k < s(sk) + s^2 k < k^d \log k$$

for large k and all $d \geq 2$, a contradiction. Hence $s \geq (k^{\frac{d-1}{2}} \sqrt{\log k})/2$. Since this holds for every litter, recalling step 4 in the Nečiporuk method as described prior to Theorem 5.3, the total number of states in the program is at least

$$k + k^d \cdot \frac{d^{h-2} - 1}{d - 1} \cdot (k^{\frac{d-1}{2}} \sqrt{\log k})/2 \geq \frac{d^{h-2} - 1}{2d - 2} \cdot k^{\frac{3d}{2} - \frac{1}{2}} \sqrt{\log k}.$$

Nondeterministic binary (i.e. 2-way) branching programs deciding $BT_d^h(k)$. Here $|R| = 2$. When the program is binary, the d variables in the litter L become $d \log k$ Boolean variables. The number s of states querying one of these $d \log k$ variables then satisfies

$$2^{k^d} \leq N_{\text{nondet}}^{2\text{-way}}(s, d \log k) \leq (d \log k)^s \cdot (2 + 1)^{2s} \cdot (2^s)^{2s} < (s \log k)^s \cdot 2^{4s+2s^2}$$

since $d \leq s$ and thus

$$k^d \leq s \log s + s \log \log k + 4s + 2s^2 \leq 3s^2 + 5s \log \log k.$$

It follows that $s \geq k^{\frac{d}{2}}/2$. Hence the total number of states in a binary nondeterministic program deciding $BT_d^h(k)$ is at least

$$k^d \cdot \frac{d^{h-2} - 1}{d-1} \cdot \frac{k^{d/2}}{2} \geq \frac{d^{h-2} - 1}{2(d-1)} \cdot k^{\frac{3d}{2}} = \frac{d^{h-2} - 1}{2(d-1)} \cdot \frac{(k^d \log k)^{3/2}}{(\log k)^{3/2}} = \Omega(n^{3/2}/(\log n)^{3/2})$$

where $n = \Theta(k^d \log k)$ is the length of the binary encoding of $BT_d^h(k)$. \square

The next two results together with Theorems 5.9 and 5.10 show limitations on the Nečiporuk method that are not necessarily present in the state sequence method. We include these to support our hope that the latter method and its generalizations have the potential to break the quadratic limitation in proving lower bounds using the Nečiporuk method.

Let $Children_d^h(k)$ have the same input as $FT_d^h(k)$ with the exception that the root function is deleted. The output is the tuple $(v_2, v_3, \dots, v_{d+1})$ of values for the children of the root. $Children_d^h(k)$ can be computed by a k -way deterministic BP with $O(k^{(d-1)h-d+2})$ states using the same black pebbling method which yields the bound (15) in Theorem 5.1.

THEOREM 5.5. *For any $d, h \geq 2$, the best k -way deterministic BP size lower bound attainable for $Children_d^h(k)$ by applying the Nečiporuk method is $\Omega(k^{2d-1})$.*

PROOF. The function $Children_d^h(k) : [k]^m \rightarrow R$ has $m = \Theta(k^d)$. Any partition $\{V_1, \dots, V_p\}$ of the set of k -ary input variables thus has $p = O(k^d)$. Claim: for each i , the best attainable lower bound on the number of states querying variables from V_i is $O(k^{d-1})$.

Consider such a set V_i , $|V_i| = v \geq 1$. Here $|R| = k^d$, so the number $N_{\det}^{k\text{-way}}(s, v)$ of distinct deterministic BPs having s non-final states querying variables from V_i satisfies

$$N_{\det}^{k\text{-way}}(s, v) \geq 1^s \cdot (s + |R|)^{sk} \geq (1 + k^d)^{sk} \geq k^{dsk}.$$

Hence the estimate used in the Nečiporuk method to upper bound $N_{\det}^{k\text{-way}}(s, v)$ will be at least k^{dsk} . On the other hand, the number of functions $f_{V_i} : [k]^v \rightarrow R$ obtained by fixing variables outside of V_i cannot exceed $k^{O(k^d)}$ since the number of variables outside V_i is $\Theta(k^d)$. Hence the best lower bound on the number of states querying variables from V_i obtained by applying the method will be no larger than the smallest s verifying $k^{ck^d} \leq k^{dsk}$ for some c depending on d and k . This proves our claim since then this number is at most $s = O(k^{d-1})$. \square

Let $SumMod_d^h(k)$ have the same input as $FT_d^h(k)$ with the exception that the root function is preset to the sum modulo k . In other words the output is $v_2 + v_3 + \dots + v_{d+1} \pmod k$.

THEOREM 5.6. *The best k -way deterministic BP size lower bound attainable for $SumMod_2^3(k)$ by applying the Nečiporuk method is $\Omega(k^2)$.*

PROOF. The function $SumMod_2^3(k) : [k]^m \rightarrow R$ has $m = \Theta(k^2)$. Consider a set V_i in any partition $\{V_1, \dots, V_p\}$ of the set of k -ary input variables, $|V_i| = v$. Here $|R| = k$, so the number $N_{\det}^{k\text{-way}}(s, v)$ of distinct deterministic BPs having s non-sink states querying variables from V_i satisfies

$$N_{\det}^{k\text{-way}}(s, v) \geq 1^s \cdot (s + |R|)^{sk} \geq (1 + k)^{sk} \geq k^{sk}.$$

If V_i contains a leaf variable, then perhaps the number of functions induced by setting variables complementary to V_i can reach the maximum k^{k^2} . Nečiporuk would conclude that k states querying the variables from such a V_i are necessary. Note that there are at most 4 sets V_i containing a leaf variable (hence a total of $4k$ states required to account for the variables in these 4 sets). Now suppose that V_i does not contain a leaf variable. Then setting the variables complementary to V_i can either induce a constant function (there are k of those), or the sum of a constant plus a variable (there are at most $k \cdot |V_i|$ of those) or the sum of two of the variables (there are at most $|V_i|^2$ of those). So the maximum number of induced functions is $|V_i|^2 = O(k^4)$. The number of states querying variables from V_i is found by Nečiporuk to be $s \geq 4/k$. In other words $s = 1$. So for any of the at least $p - 4$ sets in the partition not containing a leaf variable, the method gets one state. Since $p - 4 = O(k^2)$, the total number of states accounting for all the V_i is $O(k^2)$. \square

5.2. The state sequence method

Here we give alternative proofs for some of the lower bounds given in Section 5.1. These proofs are more intricate than the Nečiporuk proofs but they do not suffer a priori from a quadratic limitation. The method also yields stronger lower bounds for $Children_2^4(k)$ and $SumMod_2^3(k)$ (Theorems 5.9 and 5.10) than those obtained by applying Nečiporuk's method (Theorems 5.5 and 5.6).

THEOREM 5.7. $\#ndetBstates_2^3(k) \geq k^{2.5}$ for sufficiently large k .

PROOF. Consider an input I to $BT_2^3(k)$. We number the nodes in T_2^3 as in Figure 1, and let v_j^I denote the value of node j under input I . We say that a state in a computation on input I learns v_j^I if that state queries $f_j^I(v_{2j}^I, v_{2j+1}^I)$ (recall $2j, 2j+1$ are the children of node j).

Definition [Learning Interval] Let B be a k -way nondeterministic BP that solves $BT_2^3(k)$. Let $C = \gamma_0, \gamma_1, \dots, \gamma_T$ be a computation of B on input I . We say that a state γ_i in the computation is critical if one or more of the following holds:

- (1) $i = 0$ or $i = T$
- (2) γ_i learns v_2^I and there is an earlier state which learns v_3^I with no intervening state that learns v_2^I .
- (3) γ_i learns v_3^I and no earlier state learns v_3^I unless an intervening state learns v_2^I .

We say that a subsequence $\gamma_i, \gamma_{i+1}, \dots, \gamma_j$ is a learning interval if γ_i and γ_j are consecutive critical states. The interval is type 3 if γ_i learns v_3^I , and otherwise the interval is type 2.

The reason for the asymmetry in the above definition is that the initial state γ_0 of B may learn neither v_2^I nor v_3^I , in which case the initial learning interval is type 2. Since the tree T_2^3 has a symmetry which interchanges nodes 2 and 3, we may assume w.l.o.g. that γ_0 does not query the function f_3 and hence it does not learn v_3^I no matter what the input I . Thus a type 2 learning interval begins with γ_0 and/or a state which learns v_2^I , and never learns v_3^I until the last state. A type 3 learning interval begins with a state which learns v_3^I and never learns v_2^I until the last state.

Now let B be as above, and for $j \in \{2, 3\}$ let Γ_j be the set of all states of B which make a query of the form $f_j(x, y)$ for some $x, y \in [k]$. We will prove the theorem by showing that for large k

$$|\Gamma_2| + |\Gamma_3| > k^2 \sqrt{k}. \quad (18)$$

For $r, s \in [k]$ let $F_{yes}^{r,s}$ be the set of inputs I to B whose four leaves are labelled r, s, r, s respectively, whose middle node functions f_2^I and f_3^I are identically 1 except

$f_2^I(r, s) = v_2^I$ and $f_3^I(r, s) = v_3^I$, and $f_1^I(v_2^I, v_3^I) = 1$ (so $v_1^I = 1$). Thus each such I is a ‘YES input’, and should be accepted by B .

Note that for fixed r, s , each member I of $F_{yes}^{r,s}$ is uniquely specified by a triple

$$(v_2^I, v_3^I, f_1^I) \text{ where } f_1^I(v_2^I, v_3^I) = 1 \quad (19)$$

and we assume $f_1^I : [k] \times [k] \rightarrow \{0, 1\}$, so $F_{yes}^{r,s}$ has exactly $k^2(2^{k^2-1})$ members.

For $j \in \{2, 3\}$ and $r, s \in [k]$ let $\Gamma_j^{r,s}$ be the subset of Γ_j consisting of those states which query $f_j(r, s)$. Then Γ_j is the disjoint union of $\Gamma_j^{r,s}$ over all pairs (r, s) in $[k] \times [k]$. Hence to prove (18) it suffices to show

$$|\Gamma_2^{r,s}| + |\Gamma_3^{r,s}| > \sqrt{k} \quad (20)$$

for large k and all r, s in $[k]$. We will show this by showing

$$(|\Gamma_2^{r,s}| + 1)(|\Gamma_3^{r,s}| + 1) \geq k/2 \quad (21)$$

for all $k \geq 2$. (Note that given the product, the sum is minimized when the summands are equal.)

For each input I in $F_{yes}^{r,s}$ we associate a fixed accepting computation $\mathcal{C}(I)$ of B on input I .

Now fix $r, s \in [k]$. For $a, b \in [k]$ and $f : [k] \times [k] \rightarrow \{0, 1\}$ with $f(a, b) = 1$ we use (a, b, f) to denote the input I in $F_{yes}^{r,s}$ it represents as in (19).

To prove (21), the idea is that if it is false, then as I varies through all inputs (a, b, f) in $F_{yes}^{r,s}$ there are too few states learning $v_2^I = a$ and $v_3^I = b$ to verify that $f(a, b) = 1$. Specifically, we can find a, b, f, g such that $f(a, b) = 1$ and $g(a, b) = 0$, and by cutting and pasting the accepting computation $\mathcal{C}(a, b, f)$ with accepting computations of the form $\mathcal{C}(a, b', g)$ and $\mathcal{C}(a', b, g)$ we can construct an accepting computation of the ‘NO input’ (a, b, g) .

We may assume that the branching program B has a unique initial state γ_0 and a unique accepting state δ_{ACC} .

For $j \in \{2, 3\}$, $a, b \in [k]$ and $f : [k] \times [k] \rightarrow \{0, 1\}$ with $f(a, b) = 1$ define $\varphi_j(a, b, f)$ to be the set of all state pairs (γ, δ) such that there is a type j learning interval in $\mathcal{C}(a, b, f)$ which begins with γ and ends with δ . Note that if $j = 2$ then $\gamma \in (\Gamma_2^{r,s} \cup \{\gamma_0\})$ and $\delta \in (\Gamma_3^{r,s} \cup \{\delta_{ACC}\})$, and if $j = 3$ then $\gamma \in \Gamma_3^{r,s}$ and $\delta \in (\Gamma_2^{r,s} \cup \{\delta_{ACC}\})$.

To complete the definition, define $\varphi_j(a, b, f) = \emptyset$ if $f(a, b) = 0$.

For $j \in \{2, 3\}$ and $f : [k] \times [k] \rightarrow \{0, 1\}$ we define a function $\varphi_j[f]$ from $[k]$ to sets of state pairs as follows:

$$\varphi_2[f](a) = \bigcup_{b \in [k]} \varphi_2(a, b, f) \subseteq S_2$$

$$\varphi_3[f](b) = \bigcup_{a \in [k]} \varphi_3(a, b, f) \subseteq S_3$$

where $S_2 = (\Gamma_2^{r,s} \cup \{\gamma_0\}) \times (\Gamma_3^{r,s} \cup \{\delta_{ACC}\})$ and $S_3 = \Gamma_3^{r,s} \times (\Gamma_2^{r,s} \cup \{\delta_{ACC}\})$.

For each f the function $\varphi_j[f]$ can be specified by listing a k -tuple of subsets of S_j , and hence there are at most $2^{k|S_j|}$ distinct such functions as f ranges over the 2^{k^2} Boolean functions on $[k] \times [k]$, and hence there are at most $2^{k(|S_2|+|S_3|)}$ pairs of functions $(\varphi_2[f], \varphi_3[f])$. If we assume that (21) is false, we have $|S_2| + |S_3| < k$. Hence by the pigeonhole principle there must exist distinct Boolean functions f, g such that $\varphi_2[f] = \varphi_2[g]$ and $\varphi_3[f] \neq \varphi_3[g]$.

Since f and g are distinct we may assume that there exist a, b such that $f(a, b) = 1$ and $g(a, b) = 0$. Since $\varphi_2[f](a) = \varphi_2[g](a)$, if (γ, δ) are the endpoints of a type 2 learning

interval in $\mathcal{C}(a, b, f)$ there exists b' such that (γ, δ) are the endpoints of a type 2 learning interval in $\mathcal{C}(a, b', g)$ (and hence $g(a, b') = 1$). Similarly, if (γ, δ) are endpoints of a type 3 learning interval in $\mathcal{C}(a, b, f)$ there exists a' such that (γ, δ) are the endpoints of a type 3 learning interval in $\mathcal{C}(a', b, g)$.

Now we can construct an accepting computation for the ‘NO input’ (a, b, g) from $\mathcal{C}(a, b, f)$ by replacing each learning interval beginning with some γ and ending with some δ by the corresponding learning interval in $\mathcal{C}(a, b', g)$ or $\mathcal{C}(a', b, g)$. (The new accepting computation has the same sequence of critical states as $\mathcal{C}(a, b, f)$.) This works because a type 2 learning interval never queries v_3 and a type 3 learning interval never queries v_2 .

This completes the proof of (21) and the theorem. \square

THEOREM 5.8. *Every deterministic branching program that solves $BT_2^3(k)$ has at least $k^3 / \log k$ states for sufficiently large k .*

PROOF. We modify the proof of Theorem 5.7. Let B be a deterministic BP which solves $BT_2^3(k)$, and for $j \in \{2, 3\}$ let Γ_j be the set of states in B which query f_j (as before). It suffices to show that for sufficiently large k

$$|\Gamma_2| + |\Gamma_3| \geq k^3 / \log k. \quad (22)$$

For $r, s \in [k]$ we define the set $F^{r,s}$ to be the same as $F_{yes}^{r,s}$ except that we remove the restriction on f_1^I . Hence there are exactly $k^2 2^{k^2}$ inputs in $F^{r,s}$.

As before, for $j \in \{2, 3\}$, Γ_j is the disjoint union of $\Gamma_j^{r,s}$ for $r, s \in [k]$. Thus to prove (22) it suffices to show that for sufficiently large k and all r, s in $[k]$

$$|\Gamma_2^{r,s}| + |\Gamma_3^{r,s}| \geq k / \log k. \quad (23)$$

We may assume there are unique start, accepting, and rejecting states $\gamma_0, \delta_{ACC}, \delta_{REJ}$. Fix $r, s \in [k]$.

For each root function $f : [k] \times [k] \rightarrow \{0, 1\}$ we define the functions

$$\begin{aligned} \psi_2[f] : [k] \times (\Gamma_2^{r,s} \cup \{\gamma_0\}) &\rightarrow (\Gamma_3^{r,s} \cup \{\delta_{ACC}, \delta_{REJ}\}) \\ \psi_3[f] : [k] \times \Gamma_3^{r,s} &\rightarrow (\Gamma_2^{r,s} \cup \{\delta_{ACC}, \delta_{REJ}\}) \end{aligned}$$

by $\psi_2[f](a, \gamma) = \delta$ if δ is the next critical state after γ in a computation with input (a, b, f) (this is independent of b), or $\delta = \delta_{REJ}$ if there is no such critical state. Similarly $\psi_3[f](b, \delta) = \gamma$ if γ is the next critical state after δ in a computation with input (a, b, f) (this is independent of a), or $\delta = \delta_{REJ}$ if there is no such critical state.

CLAIM: The pair of functions $(\psi_2[f], \psi_3[f])$ is distinct for distinct f .

For suppose otherwise. Then there are f, g such that $\psi_2[f] = \psi_2[g]$ and $\psi_3[f] = \psi_3[g]$ but $f(a, b) \neq g(a, b)$ for some a, b . But then the sequences of critical states in the two computations $\mathcal{C}(a, b, f)$ and $\mathcal{C}(a, b, g)$ must be the same, and hence the computations either accept both (a, b, f) and (a, b, g) or reject both. So the computations cannot both be correct.

Finally we prove (23) from the CLAIM. Let $s_2 = |\Gamma_2^{r,s}|$ and let $s_3 = |\Gamma_3^{r,s}|$, and let $s = s_2 + s_3$. Then the number of distinct pairs (ψ_2, ψ_3) is at most

$$(s_3 + 2)^{k(s_2+1)} (s_2 + 2)^{k s_3} \leq (s + 2)^{k(s+1)}$$

and since there are 2^{k^2} functions f we have

$$2^{k^2} \leq (s + 2)^{k(s+1)}$$

so taking logs, $k^2 \leq k(s + 1) \log(s + 2)$ so $k / \log(s + 2) \leq s + 1$, and (23) follows. \square

Recall from Theorem 5.5 that applying the Nečiporuk method to $Children_2^4(k)$ yields a non-optimal $\Omega(k^3)$ size lower bound and from Theorem 5.6 that applying it to $SumMod_2^3(k)$ yields a non-optimal $\Omega(k^2)$ lower bound. The next two results improve on these bounds using the state sequence method. The new lower bounds match the upper bounds given by the pebbling method used to prove (15) in Theorem 5.1.

THEOREM 5.9. *Any deterministic k -way BP for $Children_2^4(k)$ has at least $k^4/2$ states.*

PROOF. Let E_{4true} be the set of all inputs I to $Children_2^4(k)$ such that $f_2^I = f_3^I = +k$ (addition mod k), and for $i \in \{4, 5, 6, 7\}$ f_i^I is identically 0 except for $f_i^I(v_{2i}^I, v_{2i+1}^I)$.

Let B be a branching program as in the theorem. For each $I \in E_{4true}$ let $\mathcal{C}(I)$ be the computation of B on input I .

For $r, s \in [k]$ let $E_{4true}^{r,s}$ be the set of inputs I in E_{4true} such that for $i \in \{4, 5, 6, 7\}$, $v_{2i}^I = r$ and $v_{2i+1}^I = s$. Then for each pair r, s each input I in $E_{4true}^{r,s}$ is completely specified by the quadruple $v_4^I, v_5^I, v_6^I, v_7^I$, so $|E_{4true}^{r,s}| = k^4$.

For $r, s \in [k]$ and $i \in \{4, 5, 6, 7\}$ let $\Gamma_i^{r,s}$ be the set of states of B that query $f_i(r, s)$, and let

$$\Gamma^{r,s} = \Gamma_4^{r,s} \cup \Gamma_5^{r,s} \cup \Gamma_6^{r,s} \cup \Gamma_7^{r,s} \quad (24)$$

The theorem follows from the following Claim.

CLAIM 1: $|\Gamma^{r,s}| \geq k^2/2$ for all $r, s \in [k]$.

To prove CLAIM 1, suppose to the contrary for some r, s

$$|\Gamma^{r,s}| < k^2/2 \quad (25)$$

We associate a pair

$$T(I) = (\gamma^I, v_i^I)$$

with I as follows: γ^I is the last state in the computation $\mathcal{C}(I)$ that is in $\Gamma^{r,s}$ (such a state clearly exists), and $i \in \{4, 5, 6, 7\}$ is the node queried by γ^I . (Here v_i^I is the value of node i).

We also associate a second triple $U(I)$ with each input I in $E_{4true}^{r,s}$ as follows:

$$U(I) = \begin{cases} (v_4^I, v_5^I, v_3^I) & \text{if } \gamma^I \text{ queries node 4 or 5} \\ (v_6^I, v_7^I, v_2^I) & \text{otherwise.} \end{cases}$$

CLAIM 2: As I ranges over $E_{4true}^{r,s}$, $U(I)$ ranges over at least $k^3/2$ triples in $[k]^3$.

To prove CLAIM 2, consider the subset E' of inputs in $E_{4true}^{r,s}$ whose values for nodes 4,5,6,7 have the form a, b, a, c for arbitrary $a, b, c \in [k]$. For each such I in E' an adversary trying to minimize the number of triples $U(I)$ must choose one of the two triples $(a, b, a+k c)$ or $(a, c, a+k b)$. There are a total of k^3 distinct triples of each of the two forms, and the adversary must choose at least half the triples from one of the two forms, so there must be at least $k^3/2$ distinct triples of the form $U(I)$. This proves CLAIM 2.

On the other hand by (25) there are fewer than $k^3/2$ possible values for $T(I)$. Hence there exist inputs $I, J \in E_{4true}^{r,s}$ such that $U(I) \neq U(J)$ but $T(I) = T(J)$. Since $U(I) \neq U(J)$ but $v_i^I = v_i^J$ (where i is the node queried by $\gamma^I = \gamma^J$) it follows that either $v_2^I \neq v_2^J$ or $v_3^I \neq v_3^J$, so I and J give different values to the function $Children_2^4(k)$. But since $T(I) = T(J)$ it follows that the two computations $\mathcal{C}(I)$ and $\mathcal{C}(J)$ are in the same state $\gamma^I = \gamma^J$ the last time any of the nodes $\{4, 5, 6, 7\}$ is queried, and the answers $v_i^I = v_i^J$ to the queries are the same, so both computations give identical outputs. Hence one of them is wrong. \square

THEOREM 5.10. *Any deterministic k -way BP for $\text{SumMod}_2^3(k)$ requires at least k^3 states.*

PROOF. We adapt the previous proof. Now $E^{r,s}$ is the set of inputs I to $\text{SumMod}_2^3(k)$ such that for $i \in \{2, 3\}$, f_i^I is identically one except possibly for $f_i^I(r, s)$, and $v_4^I = v_6^I = r$ and $v_5^I = v_7^I = s$. Note that an input to $E^{r,s}$ can be specified by the pair (v_2^I, v_3^I) , so $E^{r,s}$ has exactly k^2 elements. Define

$$\Gamma^{r,s} = \Gamma_2^{r,s} \cup \Gamma_3^{r,s}$$

Now we claim that an input I in $E^{r,s}$ can be specified by the pair (γ^I, v_i^I) , where γ^I is the last state in the computation $\mathcal{C}(I)$ that is in $\Gamma^{r,s}$, and $i \in \{2, 3\}$ is the node queried by γ^I .

The claim holds because (γ^I, v_i^I) determines the output of the computation, which in turn (together with v_i^I) determines v_j^I , where j is the sibling of i .

From the claim it follows that $|\Gamma^{r,s}| \geq k$ for all $r, s \in [k]$, and hence there must be at least k^3 states in total. \square

5.3. Thrifty lower bounds

Recall (Definition 2.4) that a thrifty branching program can only query $f_i(\vec{x})$ if \vec{x} is the correct vector of values for the children of node i .

Theorem 5.11 below shows that the upper bound given in Theorem 5.1 (15) is optimal for deterministic thrifty programs solving the function problem $FT_d^h(k)$ for $d = 2$ and all $h \geq 2$. Theorem 5.15 shows that the upper bound of k^3 given in Theorem 5.1 (17) is optimal for nondeterministic thrifty programs solving the Boolean problem $BT_d^h(k)$ for $d = 2$ and $h = 4$ (it is optimal for $h \leq 3$ by Theorem 5.2). We have not been able to extend this last result to $h > 4$.

THEOREM 5.11. *For any h, k , every deterministic thrifty branching program solving $BT_2^h(k)$ has at least k^h states.*

Fix a deterministic thrifty BP B that solves $BT_2^h(k)$. Let E be the inputs to B . Let Vars be the set of k -valued input variables (so $|E| = k^{|\text{Vars}|}$). 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. For $q \in Q$ let $\text{var}(q)$ 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 $\text{node}(\text{var}(q))$. When it is clear from the context that q is on the computation path of an input 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. If q is a state on P we say that I visits q . Let n be the number of nodes in the tree. 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 except¹² $f_1^J(v_2^I, v_3^I) := 1 - f_1^I(v_2^I, v_3^I)$; 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 5.12. *For any input 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 .*

¹²We assume the root function $f_1 : [k] \times [k] \rightarrow \{0, 1\}$

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. \square

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. $2i + 1$).

We say that a collection of nodes is a *minimal cut* of the tree if every path from root to leaf contains exactly one of the nodes. Now we assign a black 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. (Here we allow the removal of the pebbles on the children of a node i as part of the move that places a pebble on i .) 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 6. *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).

Now consider the last critical state in the computation path P^I that queries a height 2 node (i.e. a parent of leaves). We use r^I to denote this state and call it the **supercritical state** of I . The pebbling configuration associated with r^I is called the **bottleneck configuration**, and its pebbled nodes are called **bottleneck nodes**. The two children of $\text{node}(r^I)$ must be bottleneck nodes, and the bottleneck nodes form a minimal cut of the tree. The path from the root to $\text{node}(r^I)$ is the **bottleneck path**, and by Fact 6 it cannot contain any bottleneck nodes. Since the bottleneck nodes form a minimal cut, each of the $h - 1$ nodes on the bottleneck path has one or more distinct bottleneck nodes as descendants, and $\text{node}(r^I)$ has two such descendants, namely its two children. Hence there must be at least h bottleneck nodes.

Here is the main property of the pebbling sequences that we need:

FACT 7. *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 5.13. *For every $r \in R$, there is an surjective function from $[k]^{|\text{Vars}|-h}$ to E_r .*

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

PROOF. Fix $r \in R$ and let $D := E_r$. Let $i_{sc} := \text{node}(r)$. Since r is thrifty for every I in D , there are values $v_{2i_{sc}}^D$ and $v_{2i_{sc}+1}^D$ such that $v_{2i_{sc}}^I = v_{2i_{sc}}^D$ and $v_{2i_{sc}+1}^I = v_{2i_{sc}+1}^D$ for every I

in D . The surjective function of the lemma is computed by 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 $\text{adv}^I \in [k]^{|\text{Vars}|-h}$ such that $\text{INTERADV}(\text{adv}^I) \downarrow = I$ (i.e. the procedure terminates and returns I).

The idea is that the procedure INTERADV traces the computation path P starting from state r , using the advice string adv^I when necessary to answer queries made by each state q along the path. By the thrifty property, the procedure can ‘learn’ the values a, b of the children of $i = \text{node}(q)$ (if i is an internal node) from the query $f_i(a, b)$ of q . Each such child that has not been queried earlier in the trace saves one advice value for the future. By Fact 7 the parent of each of the h bottleneck nodes will be queried before the node itself, making a total savings of at least h values in the advice string. After the trace is completed, the remaining advice values complete the specification of the input $I \in E_r$.

In more detail, for each input I in D we consider the execution of the procedure using the advice string adv^I tailored for I . We maintain a current state q , a partial function v^* from nodes to $[k]$, and a set of nodes U_L (the L stands for ‘learned’). 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 the same computation path that input I follows starting from r . So initially $q = r$. In general, if $v^*(i) = a \downarrow$ (i.e. $v^*(i)$ is defined and has value a) for some a then we have determined this either from reading some element of adv^I or by querying the parent of i and using the thrifty property. 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, when given the complete advice adv^I for I there will be at least h nodes U_L^I that we ‘learn’ without directly using the advice. Such an opportunity 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 . So initially U_L is empty. There is a loop in the procedure INTERADV that iterates until $|U_L| = h$. 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 $i \in \text{Dom}(v^*)$.

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: $q := r, U_L := \emptyset, v^* := \text{undefined everywhere.}$
- 2: **Loop Invariant:** If N elements of \vec{a} have been used, then $|\text{Dom}(v^*)| = N + |U_L|$.
- 3: **while** $|U_L| < h$ **do**
- 4: $i := \text{node}(q)$
- 5: **if** i is an internal node and $2i \notin \text{Dom}(v^*)$ or $2i + 1 \notin \text{Dom}(v^*)$ **then**
- 6: let b_1, b_2 be such that $\text{var}(q) = f_i(b_1, b_2)$.
- 7: **if** $2i \notin \text{Dom}(v^*)$ **then**
- 8: $v^*(2i) := b_1$ and $U_L := U_L + 2i$.
- 9: **end if**
- 10: **if** $2i + 1 \notin \text{Dom}(v^*)$ and $|U_L| < h$ **then**
- 11: $v^*(2i + 1) := b_2$ and $U_L := U_L + (2i + 1)$.
- 12: **end if**
- 13: **end if**
- 14: **if** $i \notin \text{Dom}(v^*)$ **then**

15: let a be the next unused element of \vec{a} .
 16: $v^*(i) := a$.
 17: **end if**
 18: $q :=$ the state reached by taking the edge out of q labeled $v^*(i)$.
 19: **end while**
 20: let \vec{b} be the next $|\text{Vars}| - |\text{Dom}(v^*)|$ unused elements of \vec{a} .
 21: let $I_1, \dots, I_{|v^*(D)|}$ 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 lexicographical ordering of $[k]^{|\text{Vars}| - |\text{Dom}(v^*)|}$, and $t \leq |v^*(D)|$, then return I_t .¹³

Note that the algorithm may hang at line 18 if q is a terminal state. This can only happen if the advice string \vec{a} does not correspond to any input in D .

If the loop finishes, then there are at most $|E|/|k^{\text{Dom}(v^*)}| = k^{|\text{Vars}| - |\text{Dom}(v^*)|}$ 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 $\text{adv}^I \in [k]^{|\text{Vars}| - h}$ such that $\text{INTERADV}(\vec{a}) \downarrow = I$. 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 $|\text{Dom}(v^*)| - h$ 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 $|\text{Vars}| - h$ elements of advice. To remove that assumption, first observe that for each I , we can set the advice to some adv^I so that $I \in v^*(D)$ is maintained 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_{\perp}^I (the h nodes U_{\perp} 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. \square

COROLLARY 5.14. *For any h, k , every deterministic thrifty branching program solving $BT_2^h(k)$ has at least $\sum_{2 \leq l \leq h} k^l$ 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 $l = 2, \dots, h$. \square

THEOREM 5.15. *Every nondeterministic thrifty branching program solving $BT_2^4(k)$ has $\Omega(k^3)$ states.*

PROOF. As in the proof of the Theorem 5.7 we restrict attention to inputs I in which the function f_i associated with each internal node i satisfies $f_i(x, y) = 1$ except possibly when x, y are the values of its children. For $r, s \in [k]$ let $E^{r,s}$ be the set of all such inputs I such that for all $j \in \{4, 5, 6, 7\}$, $v_{2j}^I = r$ and $v_{2j+1}^I = s$ (i.e. each pair of sibling leaves have values r, s), and f_1 is identically 1 (so I is a YES instance). Thus I is determined by the values of its 6 middle nodes $\{2, 3, 4, 5, 6, 7\}$, so

$$|E^{r,s}| = k^6$$

Let B be a nondeterministic thrifty branching program that solves $BT_2^4(k)$, and let Γ be the set of states of B which query one of the nodes 4, 5, 6, 7. We will show $|\Gamma| = \Omega(k^3)$.

¹³See after this code for argument that $|v^*(D)| \leq k^{|\text{Vars}| - |\text{Dom}(v^*)|}$.

For $r, s \in [k]$ let $\Gamma^{r,s}$ be the set of states of Γ that query $f_j(r, s)$ for some $j \in \{4, 5, 6, 7\}$. We will show

$$|\Gamma^{r,s}| + 1 \geq k/\sqrt{3} \quad (26)$$

Since Γ is the disjoint union of $\Gamma^{r,s}$ for all $r, s \in [k]$, it will follow that $|\Gamma| = \Omega(k^3)$ as required.

For each $I \in E^{r,s}$ let $\mathcal{C}(I)$ be an accepting computation of B on input I . Let t_1^I be the first time during $\mathcal{C}(I)$ that the root f_1 is queried. Let γ^I be the last state in $\Gamma^{r,s}$ before t_1^I in $\mathcal{C}(I)$ (or the initial state γ_0 if there is no such state) and let δ^I be the first state in $\Gamma^{r,s}$ after t_1^I (or the ACCEPT state δ_{acc} if there is no such state).

We associate with each $I \in E^{r,s}$ a tuple

$$U(I) = (u, \gamma^I, \delta^I, x_1, x_2, x_3, x_4)$$

where $u \in \{1, 2, 3\}$ is a tag, and x_1, x_2, x_3, x_4 are in $[k]$ and are chosen so that $U(I)$ uniquely determines I (by determining the values of all 6 middle nodes). Specifically, $x_1 = v_i^I$, where i is the node queried by γ^I (or $i = 4$ if $\gamma^I = \gamma_0$).

Let $\mathcal{S}(I)$ denote the segment of the computation $\mathcal{C}(I)$ between γ^I and δ^I (not counting the action of the last state δ^I). This segment always queries the root $f_1(v_2, v_3)$, but does not query any of the nodes 4, 5, 6, 7 except γ^I may query node i .

Below we partition $E^{r,s}$ into three sets $E_1^{r,s}, E_2^{r,s}, E_3^{r,s}$ according to which of the nodes v_2, v_3 that $\mathcal{S}(I)$ queries. (The tag u tells us that I lies in set $E_u^{r,s}$.)

Let node $j \in \{2, 3\}$ be the parent of node i (where i is defined above) and let $j' \in \{2, 3\}$ be the sibling of j .

- $E_1^{r,s}$ consists of those inputs I for which $\mathcal{S}(I)$ queries neither v_2 nor v_3 .
- $E_2^{r,s}$ consists of those inputs I for which $\mathcal{S}(I)$ queries $v_{j'}$.
- $E_3^{r,s}$ consists of those inputs I for which $\mathcal{S}(I)$ queries v_j but not $v_{j'}$.

To complete the definition of $U(I)$ we need only specify the meaning of x_2, x_3, x_4 . The idea is that the segment $\mathcal{S}(I)$ will determine (using the definition of *thrifty*) the values of (at least) two of the six middle nodes, and x_1, x_2, x_3, x_4 will specify the remaining four values. We require that x_1, x_2, x_3, x_4 must specify the value of any node (except the root) that is queried during the segment, but the state that queries the node determines the values of its children.

In case the tag $u = 1$, the computation queries $f_1(v_2, v_3)$, and hence determines v_2, v_3 , so x_1, x_2, x_3, x_4 specify the four values v_4, v_5, v_6, v_7 .

In case $u = 2$, the computation queries $f_{j'}$ at the values of its children, so x_1, x_2, x_3, x_4 do not specify the values of these children, but instead specify v_2, v_3 .

In case $u = 3$, x_1, x_2, x_3, x_4 do not specify the value of the sibling of node i and do not specify $v_{j'}$, but do specify v_j and the values of the other level 2 nodes.

Claim: If $I, J \in E^{r,s}$ and $U(I) = U(J)$, then $I = J$.

Inequality (26) (and hence the theorem) follows from the Claim, because if $|\Gamma^{r,s}| + 1 < k/\sqrt{3}$ then there would be fewer than k^6 choices for $U(I)$ as I ranges over the k^6 inputs in $E^{r,s}$.

To prove the Claim, suppose $U(I) = U(J)$ but $I \neq J$. Then we can define an accepting computation of input I which violates the definition of *thrifty*. Namely follow the computation $\mathcal{C}(I)$ up to γ^I . Now follow the segment of $\mathcal{C}(J)$ between γ^I and δ^I , and complete the computation by following $\mathcal{C}(I)$. Notice that the segment of $\mathcal{C}(J)$ never queries any of the nodes 4, 5, 6, 7 except for v_i , and $U(I) = U(J)$ (together with the definition of $E^{r,s}$) specifies the values of the other nodes that it queries. However, since $I \neq J$, this segment of $\mathcal{C}(J)$ with input I will violate the definition of *thrifty* while querying at least one of the three nodes v_1, v_2, v_3 . \square

6. CONCLUSION

The Thrifty Hypothesis (page 4) states that thrifty branching programs are optimal among k -way BPs solving $FT_d^h(k)$ (the Tree Evaluation Problem for balanced degree d trees of height h). This implies that the black pebbling method is optimal for the deterministic case. Proving this would separate L from P (Corollary 3.3). Even disproving the hypothesis would be interesting, since it would show that one can improve upon this obvious application of pebbling.

Open Problem 1 Prove or disprove the Thrifty Hypothesis.

Corollary 5.2 gives tight lower bounds for $FT_d^h(k)$ for trees of height 3, thus proving the Thrifty Hypothesis for this case. The next important step is to extend these bounds to height 4 trees. The upper bound given in Theorem 5.1 (15) for the height 4 function problem $FT_d^4(k)$ for deterministic BPs is $O(k^{3d-2})$. If we could match this with a similar lower bound when $d = 4$ (e.g. by using a variation of the state sequence method in Section 5.2) this would yield $\Omega(k^{10})$ states for the function problem and hence (by Lemma 2.3) $\Omega(k^9)$ states for the Boolean problem $BT_4^4(k)$. Since the input length $n = O(k^4 \log k)$, this would break the Nečiporuk $\Omega(n^2)$ barrier for branching programs (see Section 5.1).

Open Problem 2 Establish the complexity of deterministic branching programs solving the Tree Evaluation Problem for height 4 trees.

For nondeterministic BPs, the upper bound given by Theorem 5.1 for the Boolean problem for height 4 trees is $O(k^{2d-1})$. This comes from the upper bound on fractional pebbling given in Theorem 4.4, which we suspect is optimal. For $h = 4$ and degree $d = 3$, the corresponding lower bound for nondeterministic BPs for $BT_3^4(k)$ would be $\Omega(k^5)$. Since the input length $n = O(k^3 \log k)$, a proof would break the Nečiporuk $\Omega(n^{3/2})$ barrier for nondeterministic BPs.

Open Problem 3 Establish the complexity of nondeterministic branching programs solving the Tree Evaluation Problem for height 4 trees.

The next two problems seem to be more accessible than the first three.

Open Problem 4 Improve Theorem 4.4 to get exact bounds on the number of pebbles required to fractionally pebble T_d^h , preferably with a direct proof.

The above problem is important since we conjecture that fractional pebbling algorithms yield optimal nondeterministic thrifty algorithms for tree evaluation (and indeed optimal with ‘thrifty’ omitted).

Open Problem 5 Generalize Theorem 5.15 to get good lower bounds for nondeterministic thrifty BPs solving $BT_2^h(k)$ for $h > 4$.

The proof of Theorem 5.11, which states that deterministic thrifty BPs require at least k^h states to solve $BT_2^h(k)$, is taken from [Wehr 2010]. That paper also proves the same lower bound for the more general class of ‘less-thrifty’ BPs, which are allowed to query $f_i(a, b)$ provided that either (a, b) correctly specify the values of both children of i , or neither a nor b is correct.

[Wehr 2010] also calculates $(k + 1)^h$ as the exact number of states required to solve $FT_2^h(k)$ using the black pebbling method, and proves that this number cannot be beaten by any k -way deterministic BP when $h = 2$. In fact, we have not been able to beat this BP upper bound by even one state, for any h and any k using any method.

Open Problem 6 Prove or disprove the hypothesis that for all $h \geq 2$ and for all sufficiently large k , every deterministic BP solving $FT_2^h(k)$ requires at least $(k+1)^h$ states.¹⁴

In [Wehr 2011], Wehr generalizes the Tree Evaluation Problem to the DAG Evaluation problem DE_G^k for each rooted DAG G and $k \geq 2$, and proves that every thrifty deterministic BP solving DE_G^k has at least k^p states, where p is the black pebble cost of G (i.e. the minimum number of pebbles required to black-pebble the root of G). This generalizes our Theorem 5.11, which applies to the case that G is a balanced binary tree. (Wehr uses his result to prove an exponential lower bound on the size of semantic incremental branching programs solving GEN, answering an open question in [Gál et al. 2008].)

This suggests generalizing the thrifty hypothesis to rooted DAGs. This would imply that for each rooted DAG G with black pebble cost p , every deterministic k -way BP solving DE_G^k has $\Omega(k^p)$ states, where the constant implied by Ω depends on G . The obvious deterministic k -way BPs coming from the black pebbling algorithm for G that uses the minimum number of pebbles p have $\Theta(k^p)$ states. We do not know how to do better than this for any G .

Open Problem 7 Find a rooted DAG G and a family $\langle B_k \rangle$ of deterministic k -way BPs, where B_k solves DE_G^k and has $o(k^p)$ states.

ACKNOWLEDGMENT

James Cook played a helpful role in the early parts of this research.

REFERENCES

- AUF DER HEIDE, F. M. 1979. *A comparison between two variations of a pebble game on graphs*. Master's thesis, Universität Bielefeld, Fakultät für Mathematik.
- BEAME, P. AND MCKENZIE, P. A note on Nečiporuk's method for nondeterministic branching programs. In preparation.
- BEAME, P., SAKS, M., SUN, X., AND VEE, E. 2003. Time-space tradeoff lower bounds for randomized computation of decision problems. *Journal of the ACM* 50, 154–195.
- BORODIN, A. AND COOK, S. 1982. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.* 11, 2, 287–297.
- BORODIN, A., RAZBOROV, A., AND SMOLENSKY, R. 1993. On lower bounds for read- k -times branching programs. *Computational Complexity* 3, 1–18.
- BRAVERMAN, M., COOK, S., MCKENZIE, P., SANTHANAM, R., AND WEHR, D. 2009a. Branching programs for tree evaluation. In *34th International Symposium, Mathematical Foundations of Computer Science*. LNCS Series, vol. 5734. Springer, 175–186.
- BRAVERMAN, M., COOK, S., MCKENZIE, P., SANTHANAM, R., AND WEHR, D. 2009b. Fractional pebbling and thrifty branching programs. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*. LIPIcs, 109–120.
- COOK, S. 1974. An observation on time-storage trade off. *J. Comput. Syst. Sci.* 9, 3, 308–316.
- COOK, S. AND SETHI, R. 1976. Storage requirements for deterministic polynomial time recognizable languages. *J. Comput. Syst. Sci.* 13, 1, 25–37.
- EDMONDS, J., IMPAGLIAZZO, R., RUDICH, S., AND SGALL, J. 2001. Communication complexity towards lower bounds on circuit depth. *Computational Complexity* 10, 3, 210–246. An abstract appeared in the *32nd IEEE FOCS (1991)*.
- GÁL, A., KOUČKÝ, M., AND MCKENZIE, P. 2008. Incremental branching programs. *Theory Comput. Syst.* 43, 2, 159–184.
- GOLDREICH, O. 2008. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press.
- HÅSTAD, J. AND WIGDERSON, A. 1993. Composition of the universal relation. *Advances in Computational Complexity Theory, AMS-DIMACS 13*, 119–134.

¹⁴Wehr [unpublished note] has shown that $(k+1)^h$ is exactly optimal among deterministic thrifty read-once BPs solving $FT_2^h(k)$. The upper bound comes from black pebbling.

- KARCHMER, M., RAZ, R., AND WIGDERSON, A. 1995. Super-logarithmic depth lower bounds via direct sum in communication complexity. *Computational Complexity* 5, 191–204. An abstract appeared in the *6th Structure in Complexity Theory Conference (1991)*.
- KLAWE, M. 1985. A tight bound for black and white pebbles on the pyramid. *J. ACM* 32, 1, 218–228.
- LENGAUER, T. AND TARJAN, R. 1980. The space complexity of pebble games on trees. *Inf. Process. Lett.* 10, 4/5, 184–188.
- LOUI, M. 1979. The space complexity of two pebble games on trees. Tech. Rep. LCS 133, MIT, Cambridge, Massachusetts.
- MAHAJAN, M. February 2007. Polynomial size log depth circuits: between NC^1 and AC^1 . *Bulletin of the EATCS* 91, 30–42.
- NEČIPORUK, È. 1966. On a boolean function. *Doklady of the Academy of the USSR* 169, 4, 765–766. English translation in *Soviet Mathematics Doklady* 7:4, pp. 999–1000.
- NORDSTRÖM, J. 2009. 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/>.
- PATERSON, M. AND HEWITT, C. 1970. Comparative schematology. In *Record of Project MAC Conference on Concurrent Systems and Parallel Computations*. 119–128. (June 1970) ACM, New Jersey.
- PUDLÁK, P. 1987. The hierarchy of boolean circuits. *Computers and artificial intelligence* 6, 5, 449–468.
- RAZBOROV, A. 1991. Lower bounds for deterministic and nondeterministic branching programs. In *8th Internat. Symp. on Fundamentals of Computation Theory*. 47–60.
- SUDBOROUGH, H. 1978. On the tape complexity of deterministic context-free languages. *J. ACM* 25, 3, 405–414.
- TAITSLIN, M. 2005. An example of a problem from PTIME and not in NLogSpace. In *Proceedings of Tver State University*. Applied Mathematics, issue 2, Tver State University, Tver Series, vol. 6(12). 5–22.
- WEGENER, I. 1987. *The Complexity of Boolean Functions*. Wiley-Teubner series in computer science. B. G. Teubner & John Wiley, Stuttgart.
- WEGENER, I. 2000. *Branching Programs and Binary Decision Diagrams*. SIAM Monographs on Discrete Mathematics and Applications. Soc. for Industrial and Applied Mathematics, Philadelphia.
- WEHR, D. 2010. Pebbling and branching programs solving the tree evaluation problem. MSc research paper, Department of Computer Science, University of Toronto, Feb, 2010, arXiv:1002.4676.
- WEHR, D. 2011. Lower bound for deterministic semantic-incremental branching programs solving GEN. (14 Jan, 2010), arXiv:1101.2705.