Electronic Colloquium on Computational Complexity, Report No. 138 (2015)

Nonuniform catalytic space and the direct sum for space

Vincent Girard*[1], Michal Koucký†[2], and Pierre McKenzie‡[3]

[1]Univ. de Montréal, <vin100_jrare@hotmail.com>
[2]Charles University, Prague, <koucky@iuuk.mff.cuni.cz>
[3]Univ. de Montréal and Chaire Digiteo E.N.S. Cachan - Polytechnique (France), <mckenzie@iro.umontreal.ca>

Monday 10th August, 2015

Abstract

This paper initiates the study of k-catalytic branching programs, a nonuniform model of computation that is the counterpart to the uniform catalytic space model of Buhrman, Cleve, Koucký, Loff and Speelman (STOC 2014). A k-catalytic branching program computes k boolean functions simultaneously on the same n-bit input. Each function has its own entry, accept and reject nodes in the branching program but internal nodes can otherwise be shared. The question of interest is to determine the conditions under which some form of a direct sum property for deterministic space can be shown to hold, or shown to fail.

We exhibit both cases. There are functions that are correlated in such a way that their direct sum fails: a significant amount of space can be saved by sharing internal computation among the k functions. By contrast, direct sum is shown to hold in some special cases, such as for the family of functions $(l_1 \circ (l_2 \circ (\cdots (l_{n-1} \circ l_n) \cdots)$ where each l_i is a literal on variable x_i, $1 \leq i \leq n$ and each $\circ$ stands individually for either $\wedge$ or $\vee$.

*Supported by a Nat. Sci. & Eng. Res. C. of Canada scholarship.

†The research leading to these results has received funding from the European Research Council under the European Unions Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 616787. Partially supported by a grant from Neuron Fund for Support of Science.

‡Supported by the Nat. Sci. & Eng. Res. C. of Canada and by the Fondation Digiteo.

ISSN 1433-8092

1 Introduction

Buhrman et al. [1] introduced catalytic computation, a space-bounded computation augmented by a large storage that can be used during the computation but has to be restored to its original state by the end of the computation. Our present paper studies the non-uniform version of this model, which we call *k-catalytic branching programs.*

This study is motivated by the aim of separating L from P and particularly understanding how space behaves under the composition of functions: if functions $f : \{0,1\}^n \to \{0,1\}^n$ and $g : \{0,1\}^n \to \{0,1\}$ require space $s(f)$ and $s(g)$ respectively, how much space does computing $g(f(\cdot))$ require? The natural answer is $s(g) + s(f) + O(\log n)$ space but is this amount optimal? This question is also closely related to the investigation of circuit depth for the composition of two functions, whose goal is to separate NC^1 from P. (See [3] for recent results on this topic.)

Buhrman et al. show a surprising result in which they can "recycle" space occupied by the function g to compute a particular function f in considerably less space that what is known without the auxiliary space. The space saved is shown proportional to the logarithm of the auxiliary space. Their goal is to compute the same function f regardless of the initial setting of the auxiliary space (which should also be the final setting).

We study this problem in the non-uniform setting of branching programs. Our model is a branching program with k entry points, each such point being associated with 2 dedicated exit points to indicate acceptance or rejection. A computation entering a particular entry point should finish in one of the associated final states corresponding to the value of a function f that is being computed on the current input. Such a k-catalytic branching program can trivially be obtained by taking k independent copies of a branching program for f. This gives a k-catalytic branching program that will be of size k times the size of a minimal branching program for f. *In terms of size, can we do better than that? For which functions can we not do better?* These are the main questions we are interested in.

A related scenario (also alluded to in [4]) is one in which, rather than computing the same function at each entry point, different functions are computed at these points. This is indeed a very natural setting as in the case of composing f and g we are interested in different output bits of f at different stages of the computation of g. The output bits of f correspond to different boolean functions.

If savings are possible when computing f in k-catalytic manner then one can save in constructing a branching program for $g(f(\cdot))$. The trivial branching program for $g(f(\cdot))$ is obtained by replacing each edge in a minimal branching program for g by a copy of a minimal branching program for f. k-catalytic savings would improve on this construction.

In this paper we show several situations in which functions $f_1, \dots, f_k$ can be computed by k-catalytic branching programs that are more efficient than just the plain sum of their complexities. We first show this unconditionally by crafting a special k-tuple of functions $f_1, \dots, f_k$ with particular correlations among them (Section 3). Then we show, conditioned on a widely accepted assumption, that the results of Buhrman et al. [1] imply other non-trivial savings. We discuss these in Section 4. In Section 5, we provide a basic analysis of k-catalytic branching programs and prove lower bounds on their sizes for some low complexity functions.

Our positive examples where saving is possible seem to rely either on specific correlations among functions or on some internal structure of a function f (implied by the fact that our example functions are in TC^1.) We show by a simple counting argument that for random k-tuple of functions $f_1, \ldots, f_k$ one cannot save when computing them catalytically. This leads to the intriguing question for random functions: *is it the case that no saving is possible when a random function f is computed k-catalytically?* So far we've been unable to settle this question, which we state as a conjecture in our concluding section.

We need to distinguish our setting from the usual setting of direct sum theorems. In the usual direct sum theorems one considers evaluating a function f on k independent (typically different) inputs in parallel. In our setting we are trying to evaluate k copies of the function f on the *same* input. This might sound ridiculous at first but it actually makes a perfect sense when properly defined for space bounded computation and branching programs. We elaborate on this point further after providing precise definitions in the next chapter. In the usual setting of computing f on k independent inputs it is known that in the case of a random function f one can obtain substantial savings in the circuit size when compared to the trivial construction ([6, 7], see also Wegener [9, Section 10.2]). However, those results seem unrelated to our setting.

2 Definitions and preliminaries

Branching programs. A *nondeterministic branching program* P is a directed acyclic graph in which each non-final node is labeled by some input variable and each edge is labeled by either 0 or 1. Two outdegree-zero *final* nodes are labeled **acc** and **rej** respectively. A computation on input x starts at a designated non-final *initial* node of indegree zero and continues along any edge labeled by the actual value of a variable that labels the current node. The computation ends when it reaches a node that has no outgoing edge labeled consistently with the input. The input x is accepted if there is a computation on input x that reaches **acc**. The input is rejected otherwise. A branching program is *deterministic* if each non-final node has exactly two outgoing edges, labeled 0 and 1 respectively. Unless stated otherwise, in this paper we consider deterministic branching programs.

Catalytic branching programs. A *k-catalytic branching program* is a branching program that has k distinct initial nodes $\mathbf{in}_1, \ldots, \mathbf{in}_k$ of indegree zero and $2k$ final nodes labeled $\mathbf{acc}_1, \mathbf{rej}_1, \mathbf{acc}_2, \mathbf{rej}_2, \ldots, \mathbf{acc}_k, \mathbf{rej}_k$. For boolean functions $f_1, f_2, \ldots, f_k : \{0,1\}^n \to \{0,1\}$, we say that the k-catalytic branching program P computes $f_1||f_2||\cdots||f_k$ if on every input $x \in \{0,1\}^n$, for each $j \in \{1, \ldots, k\}$, the computation of P starting at the initial node $\mathbf{in}_j$ on the input x reaches either $\mathbf{acc}_j$ or $\mathbf{rej}_j$, and it reaches $\mathbf{acc}_j$ iff $f_j(x) = 1$. We will use $f^{||k}$ to denote $f||f||\cdots||f$ where f is repeated k-times. A k-catalytic branching program P

is *read-once* if no path from an initial node to a final node in P queries the same variable twice.

The *size* of a branching program P, denoted $|P|$, refers to the number of nodes in P. For boolean functions $f, f_1, f_2, \ldots, f_k$ we let $\mathrm{SIZE}(f)$ to be the size of the smallest deterministic branching program computing f, and $\mathrm{SIZE}(f_1||f_2||\cdots||f_k)$ to be the size of the smallest k-catalytic branching program computing $f_1||f_2||\cdots||f_k$. Furthermore, $\mathsf{NSIZE}(\cdot)$ will be the size of smallest *nondeterministic* branching programs.

Proposition 1. *For any $f_1, f_2, \ldots, f_k : \{0,1\}^n \to \{0,1\}$:*

$$\mathrm{SIZE}(f_1||f_2||\cdots||f_k) \le \sum_{j=1}^{k} \mathrm{SIZE}(f_j).$$

In particular, $\mathrm{SIZE}(f^{||k}) \le k \cdot \mathrm{SIZE}(f)$*.*

It is natural to ask whether the bounds in Proposition 1 are tight. We will show that for some functions $f_1, f_2, \ldots, f_k$ these bounds cannot be improved, yet that for some other functions they can. This begs the following central question:

Question 2. *For which f does $\mathrm{SIZE}(f^{||k}) \ll k \cdot \mathrm{SIZE}(f)$ hold?*

We will give a special name to functions f that are at the extreme opposite end of the spectrum alluded to in Question 2:

Definition 3. *(Monolith) $f : \{0,1\}^n \to \{0,1\}$ is a* monolith *if* $\mathrm{SIZE}(f^{||k}) = k \cdot \mathrm{SIZE}(f)$*.*

Direct sum property. A k-catalytic branching program for $f^{||k}$ computes k repetitions of f, but on the same n-bit input. By contrast, a *direct sum property* is concerned with computing a function f on *different* inputs. A deterministic space direct sum property for f holds if computing f on k different inputs requires a branching program with as many nodes as a branching program performing k consecutive computations of f while remembering intermediate results (see [4, 5] for motivation and extensive pointers to the literature on such properties).

Write $f^{(k)} : \{0,1\}^{kn} \to \{0,1\}^k$ for the function that computes $f(x_1)f(x_2)\cdots f(x_k)$ from $x_1x_2\cdots x_k$ where each x_i is an n-bit input. A naive (and best known in general) branching program for computing $f^{(k)}$ consists of a full binary tree of subprograms for f. This requires a size in the order of $(2^k - 1) \cdot \mathrm{SIZE}(f)$ nodes. The following proposition is immediate as levels of the binary tree can be compressed using the 2^i-catalytic branching programs for f.

Proposition 4. *For any $f : \{0,1\}^n \to \{0,1\}$ and integer $k > 1$ it holds:*

$$\mathrm{SIZE}(f^{(k)}) \le \sum_{i=0}^{k-1} \mathrm{SIZE}(f^{||2^i}).$$

This implies that savings in k-catalytic computation for $f^{||k}$ translate into the failure of the deterministic space direct sum property. For example we can state the following consequence of the previous proposition which provides savings in computing $f^{(k)}$ when $k < \text{SIZE}(f)$.

Proposition 5. *If for every $k > 1$, $\text{SIZE}(f^{||2^k}) = O(2^k(1 + \frac{\text{SIZE}(f)}{k}))$ then for every $k > 1$, $\text{SIZE}(f^{(k)}) = O(2^k(1 + \frac{\log k}{k} \cdot \text{SIZE}(f)))$.*

Proof. A standard branching program can compute $f^{(k-\log k)}$ on the first $(k - \log k)$ n-bit arguments of $f^{(k)}$. This program uses $O(\frac{2^k}{k}\,\text{SIZE}(f))$ nodes and has $2^k/k$ exit points. Then the computation is completed by running the catalytic computation of $f^{||2^{k-i}}$, for $i = \log k, \ldots, 1$, on the last $\log k$ n-bit arguments of $f^{(k)}$. By hypothesis, this last step can be done using $O(2^k + \log k \cdot 2^k \cdot \text{SIZE}(f)/k)$ nodes, for a total size of $O(2^k(1 + \frac{\log k}{k} \cdot \text{SIZE}(f)))$. □

At the end of Section 4 we give examples of functions where the deterministic space direct sum property fails.

3 General functions $f_1, \ldots, f_k$

In this section we consider the case of computing catalytically k different functions $f_1, \ldots, f_k$. We start by noting there are functions for which the catalytic branching program cannot achieve any savings.

Proposition 6. *Let $n, k > 0$ be integers. Let $f_1, f_2, \ldots, f_k : \{0,1\}^n \to \{0,1\}$ be functions chosen uniformly at random. Then with high probability*

$$\text{SIZE}(f_1||\cdots||f_k) \geq \frac{1}{3} \cdot k \cdot \frac{2^n}{n + \log k}.$$

In particular, for $k \leq 2^n$, $\text{SIZE}(f_1||\cdots||f_k) \geq \frac{1}{6} \cdot k \cdot \frac{2^n}{n}$, with high probability.

The proof is just a standard counting argument, we give it only for completeness.

Proof. There are at most $S^{2S}n^S \leq S^{3S}$ non-isomorphic k-catalytic branching programs of size $S \geq n$. For $S = \frac{1}{3} \cdot k \cdot \frac{2^n}{n+\log k}$, the upper bound becomes

$$\left(\frac{1}{3} \cdot k \cdot \frac{2^n}{n + \log k}\right)^{k \cdot \frac{2^n}{n+\log k}} \ll 2^{k \cdot 2^n}.$$

But there are $2^{k \cdot 2^n}$ possible choices for the k-tuple of functions f_i so the claim follows. □

It is well known [10] that for any function $f : \{0,1\}^n \to \{0,1\}$, $\mathrm{SIZE}(f) \leq O(2^n/n)$. Hence by Proposition 1, for any k-tuple of functions $f_1, \dots, f_k$, $\mathrm{SIZE}(f_1 || \cdots || f_k) \leq O(k \cdot 2^n/n)$. Thus for random functions $f_1, \dots, f_k$, Proposition 1 gives an essentially optimal bound. We will show now that one can construct functions where Proposition 1 provides rather weak bound on the size of k-catalytic branching programs.

Let $k = 2^\ell$ and $1 \leq m < n, \ell$ be integers. Consider an arbitrary function $g : \{0,1\}^{n-m} \to \{0,1\}$. For each $\alpha \in \{0,1\}^{\ell-m}$ and $\beta \in \{0,1\}^m$ define a function $f_{\alpha\beta}\{0,1\}^n \to \{0,1\}$ by

$$f_{\alpha\beta}(vw) = \begin{cases} g(w), & \text{if } v = \beta \\ 0, & \text{otherwise,} \end{cases}$$

where $v \in \{0,1\}^m$ and $w \in \{0,1\}^{n-m}$.

Lemma 7. *For the functions f defined above,*

$$\mathrm{SIZE}(f_{00\cdots 0} || \cdots || f_{11\cdots 1}) \leq O(km + \frac{k}{2^m} \cdot \mathrm{SIZE}(g)).$$

Proof. Let the input variables to the branching program be labeled v, w, where v represents the first m variables and w the latter $n-m$. For each $\alpha \in \{0,1\}^{\ell-m}$ let P_α be an independent copy of a minimal branching program for g which takes as its input variables w. The catalytic branching program will compute in three phases: *mixing* phase, g-phase and *de-mixing* phase. For $\alpha \in \{0,1\}^{\ell-m}$ and $\beta \in \{0,1\}^m$, the computation starting at $\mathbf{in}_{\alpha\beta}$ first tests whether $v = \beta$, that is the mixing phase. If $v \neq \beta$ then the branching program goes into the state $\mathbf{rej}_{\alpha\beta}$ otherwise it enters the initial state of P_α. The computation continues through P_α, which constitutes the g-phase. From each of the final states of P_α, we enter the de-mixing phase in which we read the values of variables v and we go into states $\mathbf{acc}_{\alpha v}$ or $\mathbf{rej}_{\alpha v}$ depending on whether the final state of P_α is accepting or rejecting. All the accepting states of P_α share the same de-mixing states, and all the rejecting states of P_α share the same de-mixing states.

We claim that the resulting branching program is catalytic and that it computes $f_{00\cdots 0} || \cdots || f_{11\cdots 1}$. Indeed, on an actual input vw, the branching program starting from $\mathbf{in}_{\alpha\beta}$ enters P_α if $v = \beta$ or goes directly to $\mathbf{rej}_{\alpha\beta}$. Hence, it reaches P_α iff $v = \beta$. From P_α it leaves either through $\mathbf{acc}_\alpha$ or $\mathbf{rej}_\alpha$ and it continues to $\mathbf{acc}_{\alpha v} = \mathbf{acc}_{\alpha\beta}$ or $\mathbf{rej}_{\alpha\beta}$. Hence it behaves catalytically. It is also clear that it computes exactly $f_{\alpha\beta}(vw)$.

For each $\alpha\beta$, the mixing phase requires m auxiliary states to read the variable of v one by one and check that their correspond to β. For each α, the de-mixing phase requires a branching program (decision tree) that reads all the m variables of v one by one and branches according to their values. Thus it requires $2^m - 1$ auxiliary states for each accepting or rejecting state of P_α. In total the branching program will be of size at most $2^\ell m + 2^{\ell-m} \cdot \mathrm{SIZE}(g) + 2^{\ell-m} \cdot 2 \cdot (2^m - 1) \leq O(km + \frac{k}{2^m} \cdot \mathrm{SIZE}(g))$ □

If $\mathrm{SIZE}(g) \geq \Omega(m2^m)$, the bound in the previous lemma becomes $O(\frac{k}{2^m} \cdot \mathrm{SIZE}(g))$. This will be the case when we chose $m = \frac{n}{2} - \log n$ and g will be a function of branching program complexity $\Omega(2^{n/2})$. A random function g will have such a high complexity. In that case we get $O(\frac{k}{2^m} \cdot \mathrm{SIZE}(g))$ for the k-catalytic branching program computing $f_{00\cdots 0}|| \cdots ||f_{11\cdots 1}$ instead of $\theta(k \cdot \mathrm{SIZE}(g))$. Hence we achieve substantially better bound than provided by Proposition 1. For example when $k = O(2^m)$ we have $O(\mathrm{SIZE}(g))$ so the increase in the size of the catalytic program will be only constant factor. One could argue that the savings above were to be expected because the function f used is often zero. However, we can generalize the above example to functions h of arbitrary complexity. This contrasts with Proposition 6.

Let $k = 2^\ell$ and $1 \leq m < n, \ell$ be integers. Consider an arbitrary function $g : \{0,1\}^n \to \{0,1\}$. For each $\alpha \in \{0,1\}^{\ell-m}$ and $\beta \in \{0,1\}^m$ define a function $h_{\alpha\beta}\{0,1\}^n \to \{0,1\}$ by

$$h_{\alpha\beta}(vw) = g((\beta \oplus v) \cdot w).$$

where $v \in \{0,1\}^m$, $w \in \{0,1\}^{n-m}$, and $\oplus$ is the bit-wise xor.

Lemma 8. *For the functions h defined above,*

$$\mathrm{SIZE}(h_{00\cdots 0}|| \cdots ||h_{11\cdots 1}) \leq O(km + \frac{k}{2^m} \cdot \sum_{v \in \{0,1\}^m} \mathrm{SIZE}(g(v\cdot)).$$

Proof. The proof closely mimics the previous one. Again, the computation proceeds in three phases: *mixing* phase, g-phase and *de-mixing* phase. For each α and β, the catalytic branching program contains an independent copy $P_{\alpha\beta}$ of a minimal branching program computing the function $g(\beta\cdot)$ on m variables w. In the mixing phase, starting from the initial state $\mathbf{in}_{\alpha\beta}$, the branching program queries v and enters the initial state of $P_{\alpha\cdot(\beta\oplus v)}$. In the g-phase the computation passes through one of the $P_{\alpha\beta}$'s. In the de-mixing phase, the computation goes from an accepting state of $P_{\alpha\beta}$ into the accepting state $\mathbf{acc}_{\alpha\cdot(\beta\oplus v)}$. Similarly for rejecting states.

Similarly to the previous proof one can easily verify that the program catalytically computes $h_{00\cdots 0}|| \cdots ||h_{11\cdots 1}$.

The mixing phase starting at each $\mathbf{in}_{\alpha\beta}$ is implemented by a decision tree which reads all m variables of v, and reaches the appropriate $P_{\alpha\cdot(\beta\oplus v)}$ accordingly. The de-mixing phase is similarly implemented by another copy of the decision tree. Thus the total size of the program is bounded by $3 \cdot 2^\ell \cdot 2^m$ size for the mixing and de-mixing, plus $O(2^{\ell-m} \sum_{v\in\{0,1\}^m} \mathrm{SIZE}(g(v\cdot))$ for the g-phase. The size of the mixing and de-mixing can be further reduced to $3 \cdot 2^\ell m$ by removing redundancy. (The query process proceeds bit by bit, and we flip the next bit in the label of the current auxiliary state based on the value of the next bit of v.) This gives the total size of the branching program as required. $\square$

If g is taken to be a random function then its complexity will be $\Omega(2^n/n)$ with high probability. This will also be the complexity of all the functions h. By setting $m = \frac{n}{2} - \log n$, the upper bounds in the previous lemma will become $O(\frac{k}{2^m} \cdot 2^m \cdot \frac{2^{n-m}}{n-m}) = O(\frac{k}{2^m} \cdot \frac{2^n}{n}) = O(\frac{k}{2^m} \cdot \mathrm{SIZE}(g))$. Again, we obtain non-trivial savings.

4 Uniform catalytic computation

Buhrman et al. [1] study the uniform version of our problem. Their setup is as follows: they consider the space-bounded computation on the usual Turing machines where the input is on a read-only input tape and the computation can use space s on its work tape. In addition to that the machine has an auxiliary read/write tape of size ℓ that can also be used during the computation provided that at the end of the computation the content of this auxiliary tape is restored to content it had at the beginning of the computation.

Buhrman et al. focus on $s = O(\log n)$ and $\ell = 2^s$, and call the class of functions computable in this setting catalytic log-space (CL). The same function f must be computed regardless of the starting content of the auxiliary tape, so the tape plays the role of a catalyst in a chemical reaction which facilitates the reaction but is recovered at the end. Buhrman et al. show that catalytic log-space contains various functions that are not known to be in log-space, i.e., computable without the auxiliary tape. An example of such a function is the problem of multiplying n integer matrices or any function in (uniform) TC^1 (which includes nondeterministic log-space NL and problems log-space reducible to context-free languages LOGCFL, see [8]).

Our catalytic branching programs capture the non-uniform version of this model. The k entry points of the catalytic branching program correspond to the 2^ℓ possible initial contents of the auxiliary tape, and the requirement to leave in the corresponding final state corresponds to the requirement to restore the contents of the auxiliary tape. If one takes the usual configuration graph of the Turing machine with the auxiliary (*catalytic*) tape, one obtains the k-catalytic branching program provided the Turing machine meets the requirement on preserving the content of the tape. The size of the branching program will be bounded by the number of possible configurations of the Turing machine which is $O(2^\ell \cdot 2^s \cdot \ell \cdot s \cdot n)$, where 2^ℓ counts the different contents of the catalytic tape, 2^s counts the contents of the work tape, and ℓ, s, n count the possible head positions. For catalytic log-space this size becomes $2^\ell \cdot n^{O(1)} = k \cdot n^{O(1)}$ for $k = 2^{n^{O(1)}}$. We get the following claim.

Proposition 9. *Let f be a function computed catalytically in space $s(n)$ using catalytic tape of size $\ell(n) \leq 2^{s(n)}$. Then* $\mathrm{SIZE}(f^{||2^{\ell(n)}}) \leq n \cdot 2^{\ell(n)+O(s(n))}$.

The reverse direction also holds as formulated in the following statement.

Proposition 10. *Let $\{f_n\}_{n>0}$ be a sequence of boolean functions and $\{k_n\}_{n>0}$ be a sequence of integers. There is a sequence of advice strings $\{d_n\}_{n>0}$ such that* $|d_n| \leq O(\mathrm{SIZE}(f_n^{||k_n}) \cdot$

$\log \mathrm{SIZE}(f_n^{||k_n})$) *and a Turing machine M with an auxiliary (catalytic) tape such that for any input $x \in \{0,1\}^n$ and any string $w \in \{0,1\}^{\lceil \log k_n \rceil}$, the machine M on input $x\#d_n$ written on its input tape and w written on its catalytic tape computes $f(x)$ while using at most $O(\log \frac{\mathrm{SIZE}(f_n^{||k_n})}{k_n} + \log n)$ work space and finishing the computation with w written on its catalytic tape.*

Proof. We provide only a sketch of the proof. The advice d_n contains a suitable description of the k_n-catalytic branching program, and the machine M interprets this program on its input x. We may assume that each node of the branching program is labeled by a string of $\log \mathrm{SIZE}(f_n^{||k_n})$ bits, where we store the first $\log k_n$ bits on the catalytic tape of M and the remaining bits on the work tape. We may assume WLOG that the k_n initial nodes of the branching program are the ones which have only zeros in the "work tape" part, the corresponding final nodes agree with them on the "catalytic tape" part and are distinguished by a particular setting of the "work tape" part. Simulation then proceeds by using the catalytic tape together with a part of the work tape as a counter to advance the input head to a particular position in d_n where one can read off the next configuration of the simulated branching program. By the properties of the catalytic branching program when the computation finishes, the catalytic tape contains again the same string as at the beginning which is the part of the label of the final node. $\square$

The technique of Buhrman et al. [1] gives the following theorems.

Theorem 11 (Buhrman, Cleve, Koucký, Loff, Speelman)**.** *It holds:*

1. *Let $\mathsf{IMM}_{n,a\times a} : \{0,1\}^{n\cdot a^2} \to \{0,1\}$ be the function giving the $(1,1)$ entry of the product of n $a \times a$ matrices over $GF[2]$. Let $a \le n$ and $k \ge 2^{3a^2}$. Then $\mathrm{SIZE}(\mathsf{IMM}_{n,a\times a}^{||k}) \le k \cdot n^{O(1)}$.*

2. *For any $d, b \ge 1$ there is a constant $c > 0$ such that if $f : \{0,1\}^n \to \{0,1\}$ is a function computable by TC^1 circuits of depth $d \log n$ consisting of at most n^b MAJ-gates and NOT-gates then for $k \ge 2^{n^c}$ we have $\mathrm{SIZE}(f^{||k}) \le k \cdot n^c$.*

In Part 1 of the previous theorem, there is nothing special about the ring $GF[2]$, and one can obtain a similar claim for multiplying matrices over say the ring of integers $\mathbb{Z}_{2^n}$ and testing whether the $(1,1)$ entry is zero to make the function Boolean. For such matrix multiplication one will require $k \ge 2^{12a^2 \log n}$ to obtain the similar bound $k \cdot n^{O(1)}$ on the size of the k-catalytic branching program.

We do not know how to compute the product of n $a \times a$ matrices in space less than $O(\log a \cdot \log n)$. In particular, we do not know whether such matrix multiplication can be done by branching programs of size smaller than $n^{O(\log a)}$, so for $a \in \omega(1)$ whether it can be done by polynomial size branching programs. If nondeterministic log-space is not contained in non-uniform deterministic log-space ($\mathsf{NL} \not\subseteq \mathsf{L}/poly$) then $\mathsf{IMM}_{n,n\times n}$ requires super-polynomial branching programs. One can state:

Proposition 12. *If* $\mathsf{NL} \not\subseteq \mathsf{L}/poly$ *then* $\mathsf{IMM}_{n,n\times n} \notin \mathrm{SIZE}(n^{O(1)})$.

Under this assumption or the weaker assumption $\mathsf{LOGCFL} \not\subseteq \mathsf{L}/poly$ we obtain super-polynomial savings for the k-catalytic branching program:

Corollary 13. *If* $\mathsf{LOGCFL} \not\subseteq \mathsf{L}/poly$ *then* $\mathrm{SIZE}(\mathsf{IMM}_{n,n\times n}^{||k}) \leq \frac{k}{n^{\omega(1)}} \cdot \mathrm{SIZE}(\mathsf{IMM}_{n,n\times n})$ *for* $k \geq 2^{3n^2}$.

Under the same assumption as above one can show using the technique of Proposition 5 that $\mathrm{SIZE}(\mathsf{IMM}_{n,n\times n}^{(\ell)}) \leq \frac{2^\ell}{n^{\omega(1)}} \cdot \mathrm{SIZE}(\mathsf{IMM}_{n,n\times n})$ for $\ell \geq n^3$. The deterministic space direct sum property of $\mathsf{IMM}_{n,n\times n}$ will not hold in this case.

5 When a catalytic space direct sum property holds

Recall that a monolith is a function f for which k-catalytic computation of $f^{||k}$ provides no advantage whatsoever over k separate computations of f. This section is mostly devoted to proving the following:

Theorem 14. *Any* $f(x_1, \cdots, x_n)$ *expressible as* $p_1 \ \Delta_1 \ (\cdots \ p_{m-1} \ \Delta_{m-1} \ p_m)\cdots)$, *where* $m \leq n$, *for all* $i \in \{1, \cdots, m\}$, $p_i \in \{x_i, \neg x_i\}$ *and for all* $j \in \{1, \cdots, m-1\}$, $\Delta_j \in \{\vee, \wedge\}$, *is a monolith.*

It follows in particular that the AND and the OR functions are monoliths. We will also observe that read-once branching programs cannot benefit from catalytic computation.

Note that we already know monoliths: any f that depends on at most one bit is a monolith. This is because then $\mathrm{SIZE}(f) = 3$ and by definition for any g, $\mathrm{SIZE}(g^{||k}) \geq 3k$.

We will use the following definitions in this section. A *path* in a k-catalytic branching program P refers to a graph-theoretic path. A node v in P is *co-accessible* if a path connects v to a final node. If every node in P is co-accessible, then P itself is declared co-accessible. For $1 \leq i \leq k$, an *i-path* will refer to a path starting at $\mathbf{in}_i$. For a fixed i, the *ith slice* of P is the subprogram formed by all the nodes and edges in P that can be reached from $\mathbf{in}_i$. A path (resp. a set of paths) in P is *consistent* if some setting of the input variables $x_1, \ldots, x_n$ allows traversing the path (resp. allows traversing every path in the set). A path π and a path σ are said to be *twins* if π and σ are consistent and every variable that is queried along both paths gets the same answer along each.

Proposition 15. *If every initial-to-final path in a k-catalytic branching program P computing $f^{||k}$ is consistent, then* $|P| \geq k \cdot \mathrm{SIZE}(f)$.

Proof. We can assume that P is co-accessible. This is because every $\mathbf{in}_i$ node in a k-catalytic branching program is co-accessible by definition, and any node v that is not co-accessible can be removed from P, reducing the size of P with no effect on the function computed.

Suppose to the contrary that some non-final node v is shared by the ith and the jth slices of P, $i \neq j$. Then an i-path π_i and a j-path π_j meet at v. Since v is co-accessible, some path π connects v to an output node in P. The initial-to-final paths $\pi_i\pi$ and $\pi_j\pi$ are consistent by hypothesis, i.e., there is an input $x \in \{0,1\}^n$ traversing $\pi_i\pi$ and an input $y \in \{0,1\}^n$ traversing $\pi_j\pi$. Regardless of $f(x)$ or $f(y)$, this is a contradiction since $\pi_i\pi$ and $\pi_j\pi$ have the same endpoint. We conclude that every non-final node in P belongs to a unique slice. Since each slice requires $\mathrm{SIZE}(f)$ nodes in order to compute f, the total number of nodes in P is at least $k \cdot \mathrm{SIZE}(f)$. □

Corollary 16. *If a read-once k-catalytic branching program P computes $f^{||k}$, then $|P| \geq k \cdot \mathrm{SIZE}(f)$.*

Proof. In a read-once branching program, no initial-to-final path is inconsistent. So Proposition 15 applies. □

The proof of Proposition 15 actually exploits the following:

Lemma 17. *If $1 \leq i < j \leq k$ and an i-path π_i and a j-path π_j meet at v in a k-catalytic branching program P, then every consistent path from v to a final node in P extends either π_i or π_j inconsistently.*

Remark 18. *Paths π and σ are twins iff the set $\{\pi, \sigma\}$ is consistent.*

Lemma 19. *Consider an i-path π_i and a j-path π_j, $i \neq j$, in a co-accessible k-catalytic branching program computing some $f^{||k}$. If π_i and π_j are twins then π_i and π_j are disjoint.*

Proof. Suppose to the contrary that the twin paths π_i and π_j meet at some node v. Since $\{\pi_i, \pi_j\}$ is a consistent set of paths, there exists an input x that can traverse both π_i and π_j. Starting from $\mathbf{in}_i$ or from $\mathbf{in}_j$, such an x reaches v and follows a unique path from v on to some unique final node in P. This is a contradiction. □

Proposition 20. *Let $f : \{0,1\}^n \longrightarrow \{0,1\}$ and $g : \{0,1\}^{n-1} \longrightarrow \{0,1\}$ be functions where $f(x_1, x_2, \cdots, x_n) = x_1 \wedge g(x_2, \cdots, x_n)$. Then f is a monolith if and only if g is a monolith.*

Proof. First we can observe that if g rejects every input, then so does f. We then have $\mathrm{SIZE}(f) = \mathrm{SIZE}(f^{||k}) = \mathrm{SIZE}(g) = \mathrm{SIZE}(g^{||k}) = 0$. So both functions are monoliths, which proves the proposition for this particular case.

Now we can assume that g accepts at least one input. For this case we will demonstrate a property: for every $k \in \mathbb{N}$ (including $k = 1$), we have $\mathrm{SIZE}(f^{||k}) = k + \mathrm{SIZE}(g^{||k})$.

$\leq$: We just have to give a branching program of size $k + \mathrm{SIZE}(g^{||k})$ that calculates $f^{||k}$. To do so, we take a branching program of size $\mathrm{SIZE}(g^{||k})$ that calculates $g^{||k}$. We then add a node denoted x_1 before every initial node. The outgoing edge denoted

1 sends us to the associated initial node, while the one denoted 0 sends us to the "reject" final node of the associated initial node.

We then have a branching program that rejects the input if $x_1 = 0$ and returns the value of $g(x_2, \cdots, x_n)$ if $x_1 = 1$ which is exactly the function f.

$\geq$: This inequality is a bit more tricky. We take a branching program of size $\text{SIZE}(f^{||k})$ that calculates $f^{||k}$. Since this program is minimal, we can assume it is co-accessible. Now since g accepts at least one input, so does f. If we fix that input, we get k paths that each start at a different initial node and ends at the corresponding accepting final node. By Lemma 19, these paths must be disjoint. We can observe that each path must have at least one node denoted x_1, or else the path would still accept the input if we changed the value of x_1 to 0. So there are at least k nodes denoted x_1 in the program.

Now let's do exactly as we did in the proof of the other inequality: we add a node denoted x_1 for each initial node to obtain a branching program of size $k + \text{SIZE}(f^{||k})$ calculating $(x_1 \wedge f(x_1, \cdots, x_n))^{||k}$ (which is equivalent to $f(x_1, \cdots, x_n)^{||k}$). Doing this assures us that every path in the program starts in one of these k nodes. Now for every other node denoted x_1, every path passing through this node must have already answered 1 to the value of x_1 at the first node, so we can just remove that node and reroute every edge pointing to it to where the edge $x_1 = 1$ was pointing. By doing so, we know we removed at least k nodes, so the new branching program (calculating $f^{||k}$) is of size at most $\text{SIZE}(f^{||k})$.

Finally, we observe that the subprogram without the k (initial) nodes denoted x_1 contains exactly $\text{SIZE}(f^{||k}) - k$ nodes, none of which are denoted x_1. This branching program must output $(g(x_2, \cdots, x_n))^{||k}$ and thus be at least of size $\text{SIZE}(g^{||k})$.

Now that we have proven our equality, we can finish the proof of our proposition:

$$\begin{aligned} f \text{ is a monolith } &\Leftrightarrow \forall k \in \mathbb{N},\ \text{SIZE}(f^{||k}) = k\,\text{SIZE}(f) \\ &\Leftrightarrow \forall k \in \mathbb{N},\ k + \text{SIZE}(g^{||k}) = k(1 + \text{SIZE}(g)) \\ &\Leftrightarrow \forall k \in \mathbb{N},\ \text{SIZE}(g^{||k}) = k\,\text{SIZE}(g) \\ &\Leftrightarrow g \text{ is a monolith} \end{aligned}$$

□

Corollary 21. *Let $f : \{0,1\}^n \longrightarrow \{0,1\}$ and $g : \{0,1\}^{n-1} \longrightarrow \{0,1\}$ be functions where $f(x_1, x_2, \cdots, x_n) = x_1 \vee g(x_2, \cdots, x_n)$. Then f is a monolith if and only if g is a monolith.*

Proof. We know that replacing a variable of the function by its negation or adding a negation in front of said function doesn't change the size of its minimal branching program.

(In the first case, you switch the outgoing edges of the associated nodes and in the second case, you switch the final nodes.)

So $f(x_1, x_2, \cdots, x_n)$ is a monolith if and only if $\neg f(\neg x_1, x_2, \cdots, x_n) = \neg((\neg x_1) \vee g(x_2, \cdots, x_n)) = x_1 \wedge \neg g(x_2, \cdots, x_n)$ is a monolith which, by our proposition, is a monolith if and only if $\neg g(x_2, \cdots, x_n)$ is a monolith. But like we said, $\neg g(x_2, \cdots, x_n)$ is a monolith if and only if $g(x_2, \cdots, x_n)$ is a monolith which completes the proof. □

Corollary 22. *Let $f : \{0,1\}^n \longrightarrow \{0,1\}$ and $g : \{0,1\}^{n-1} \longrightarrow \{0,1\}$ be functions where $f(x_1, x_2, \cdots, x_n) = p \; \Delta \; g(x_2, \cdots, x_n)$ (where $p \in \{x_1, \neg x_1\}$ and $\Delta \in \{\vee, \wedge\}$). Then f is a monolith if and only if g is a monolith.*

Proof. This results directly from what we have proven so far and the fact that replacing a variable of the function by its negation doesn't change the size of its minimal branching program. □

Corollary 23. *Let $f : \{0,1\}^n \longrightarrow \{0,1\}$ and $g : \{0,1\}^{n-1} \longrightarrow \{0,1\}$ be functions where $f(x_1, \cdots, x_n) = p_1 \; \Delta_1 \; (p_2 \; \Delta_2 \; (\cdots \; p_m \; \Delta_m \; g(x_{m+1}, \cdots, x_n) \cdots))$ (where for all $i \in \{1, \cdots, m\}$, we have $p_i \in \{x_i, \neg x_i\}$ and $\Delta_i \in \{\vee, \wedge\}$). Then f is a monolith if and only if g is a monolith.*

Proof. This is simply proven by applying induction to our last corollary. □

Proof of Theorem 14. First if $n = m$, we only need to prove that the function $g(x) = x$ is a monolith. But this function depends on a single variable and we already observed that such functions are monoliths. Then we can write $f(x_1, \cdots, x_n) = p_1 \; \Delta_1 \; (\cdots \; p_{m-1} \; \Delta_{m-1} \; g(p_m)) \cdots)$ which is a monolith if and only if $g(p_m)$ is a monolith which is true if and only if $g(x_m)$ is a monolith.

Now if $m < n$ we define $g : \{0,1\}^{n-m} \longrightarrow \{0,1\}$ as the function which rejects every input. Like we said earlier, this function has the property that $\text{SIZE}(g^{||k}) = 0$ for all $k \in \mathbb{N}$ so it is a monolith.

Now we only have to observe that we can rewrite f as:

$$f(x_1, \cdots, x_n) = p_1 \; \Delta_1 \; (\cdots \; p_{m-1} \; \Delta_{m-1} \; (p_m \vee g(x_{m+1}, \cdots, x_n)) \cdots)$$

which by our last corollary must be a monolith since g is a monolith. □

Corollary 24. *(to Theorem 14) Any $f : \{0,1\}^n \to \{0,1\}$ such that $|f^{-1}(1)| = 1$ or $|f^{-1}(0)| = 1$, such as $x_1 \wedge \cdots \wedge x_n$ and $x_1 \vee \cdots \vee x_n$, is a monolith.*

6 Conclusion

In this paper we have exhibited functions for which a form of direct sum property for k-catalytic space computation fails, i.e, for which significant savings over the obvious upper bound are possible when computing k different functions on the same input. We have also exhibited examples where such a property holds, such as when computing k random functions, or when computing the function $x_1 \vee (\neg x_2 \vee (x_3 \wedge (x_4 \cdots (x_{n-1} \wedge x_n) \cdots)))$ k times.

We have observed that failure of the direct sum property for k-catalytic space with all functions equal to the same f implies failure of the direct sum property for deterministic space (involving the computation of f on k different inputs).

Of the many open questions that remain, we single out two. The first is whether the following, which we frame as a conjecture, holds:

Conjecture 25. *For a random f, $\mathrm{SIZE}(f^{||k}) \geq \Omega(k \cdot \mathrm{SIZE}(f))$ with high probability.*

Here one should contrast our conjecture with the work of Uhlig [6, 7] who shows that the direct sum property is provably false in the model of Boolean circuits. Can a similar argument be made for the branching program size measure?

The second question is whether a characterization of low complexity monoliths can be developed. We cannot expect a full characterization of monoliths because declaring f a monolith requires knowledge of the exact optimal size of a branching program for f. We have shown here that all Boolean functions with singleton support are monoliths. Are all Boolean functions $f(x_1, \ldots, x_n)$ with $|f^{-1}(1)| = 2$ monoliths? In particular, is the "all-bits-equal" function, for which we can prove an exact size bound of $\lceil 3n/2 \rceil$, a monolith?

Acknowledgements

The question on the size of what we call catalytic branching programs was first asked by Steve Cook and Yuval Filmus in the context of their project of proving strong lower bounds for functions not known to be in L, namely the Tree Evaluation Function (see e.g. [2]). We thank Valentine Kabanets for pointers to [9, 4].

References

[1] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *Symposium on Theory of Computing, STOC 2014*, pages 857–866, 2014.

[2] Stephen A. Cook, Pierre McKenzie, Dustin Wehr, Mark Braverman, and Rahul Santhanam. Pebbles and branching programs for tree evaluation. *TOCT*, 3(2):4, 2012.

[3] Dmitry Gavinsky, Or Meir, Omri Weinstein, and Avi Wigderson. Toward better formula lower bounds: an information complexity approach to the krw composition conjecture. In *STOC*, pages 213–222, 2014.

[4] Ronen Shaltiel. Towards proving strong direct product theorems. *Computational Complexity*, 12(1-2):1–22, 2003.

[5] Alexander A. Sherstov. Strong direct product theorems for quantum communication and query complexity. *SIAM J. Comput.*, 41(5):1122–1165, 2012.

[6] Dietmar Uhlig. On the synthesis of self-correcting schemes for functional elements with a small number of reliable elements. *Math. Notes. Acad. Sci. USSR*, 16:558–562, 1974.

[7] Dietmar Uhlig. Networks computing boolean functions for multiple input values. In *Poceedings of the London Mathematical Society Symposium on Boolean Function Complexity*, pages 165–173, New York, NY, USA, 1992. Cambridge University Press.

[8] H. Vollmer. *Introduction to Circuit Complexity – A Uniform Approach.* Texts in Theoretical Computer Science. Springer Verlag, 1999.

[9] I. Wegener. *The Complexity of Boolean Functions.* Wiley-Teubner series in computer science. B. G. Teubner & John Wiley, Stuttgart, 1987.

[10] I. Wegener. *Branching Programs and Binary Decision Diagrams.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de