View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

A Note on Amortized Branching Program Complexity*

Aaron Potechin

Institute for Advanced Study, Princeton, NJ, USA
aaronpotechin@gmail.com

Abstract

In this paper, we show that while almost all functions require exponential size branching programs to compute, for all functions f there is a branching program computing a doubly exponential number of copies of f which has linear size per copy of f. This result disproves a conjecture about non-uniform catalytic computation, rules out a certain type of bottleneck argument for proving non-monotone space lower bounds, and can be thought of as a constructive analogue of Razborov's result that submodular complexity measures have maximum value $O(n)$.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases branching programs, space complexity, amortization

Digital Object Identifier 10.4230/LIPIcs.CCC.2017.4

1 Introduction

In amortized analysis, which appears throughout complexity theory and algorithm design, rather than considering the worst case cost of an operation, we consider the average cost of the operation when it is repeated many times. This is very useful in the situation where operations may have a high cost but if so, this reduces the cost of future operations. In this case, the worst-case rarely occurs and the average cost of the operation is much lower. A natural question we can ask is as follows. Does amortization only help for specific operations, or can any operation/function be amortized?

For boolean circuits, which are closely related to time complexity, Uhlig [12],[13] showed that for any function f, as long as m is $2^{o(\frac{n}{\log n})}$ there is a circuit of size $O(\frac{2^n}{n})$ computing f on m different inputs simultaneously. As shown by Shannon [11] and Lupanov [6], almost all functions require circuits of size $\Theta(\frac{2^n}{n})$ to compute, which means that for almost all functions f, the cost to compute many inputs of f is essentially the same as the cost to compute one input of f!

In this paper, we consider a similar question for branching programs, which are closely related to space complexity. In particular, what is the minimum size of a branching program which computes many copies of a function f on the same input? This question is highly non-trivial because branching programs are not allowed to copy bits, so we cannot just compute f once and then copy it. In this paper, we show that for $m = 2^{2^n - 1}$, there is a branching program computing m copies of f which has size $O(mn)$ and thus has size $O(n)$ per copy of f.

This work has connections to several other results in complexity theory. In catalytic computation, introduced by Buhrman, Cleve, Koucký, Loff, and Speelman [2], we have an

* This material is based upon work supported by the National Science Foundation under agreement No. CCF-1412958 and by the Simons Foundation.

© Aaron Potechin;
licensed under Creative Commons License CC-BY

additional tape of memory which is initially full of unknown contents. We are allowed to use this tape, but we must restore it to its original state at the end of our computation. As observed by Girard, Koucký, and McKenzie [5], the model of a branching program computing multiple instances of a function is a non-uniform analogue of catalytic computation and our result disproves Conjecture 25 of their paper. Our result also rules out certain approaches for proving general space lower bounds. In particular, any lower bound technique which would prove a lower bound on amortized branching program complexity as well as branching program size cannot prove non-trivial lower bounds. Finally, our result is closely related to Razborov's result [10] that submodular complexity measures have maximum size $O(n)$ and can be thought of as a constructive analogue of Razborov's argument.

1.1 Outline

In Section 2 we give some preliminary definitions. In section 3 we give our branching program construction, proving our main result. In section 4 we briefly describe the relationship between our work and catalytic computation. In section 5 we discuss which lower bound techniques for proving general space lower bounds are ruled out by our construction. Finally, in section 6 we describe how our work relates to Razborov's result [10] on submodular complexity measures.

2 Preliminaries

In this section, we define branching programs, branching programs computing multiple copies of a function, and the amortized branching program complexity of a function.

▶ **Definition 1.** We define a branching program to be a directed acyclic multi-graph B with labeled edges and distinguished start nodes, accept nodes, and reject nodes which satisfies the following conditions.

1. Every vertex of B has outdegree 0 or 2. For each vertex $v \in V(B)$ with outdegree 2, there exists an $i \in [1, n]$ such that one of the edges going out from v has label $x_i = 0$ and the other edge going out from v has label $x_i = 1$.
2. Every vertex with outdegree 0 is an accept node or a reject node.

Given a start node s of a branching program and an input $x \in \{0,1\}^n$, we start at s and do the following at each vertex v that we reach. If v is an accept or reject node then we accept or reject, respectively. Otherwise, for some i, one of the labels going out from v has label $x_i = 0$ and the other edge going out from v has label $x_i = 1$. If $x_i = 0$ then we take the edge with label $x_i = 0$ and if $x_i = 1$ then we take the edge with label $x_i = 1$. In other words, we follow the path starting at s whose edge labels match x until we reach an accept or reject node and accept or reject accordingly.

Given a branching program B and start node s, let $f_{B,s}(x) = 1$ if we reach an accept node when we start at s on input x and let $f_{B,s} = 0$ if we reach a reject node when we start at s on input x. We say that (B, s) computes $f_{B,s}$.

We define the size of a branching program B to be $|V(B)|$, the number of vertices/nodes of B.

▶ Remark. We can consider branching programs whose start nodes all compute different functions, but for this note we will focus on branching programs whose nodes all compute the same function.

▶ **Definition 2.** We say that a branching program B computes f m times if B has m start nodes $s_1, \cdots, s_m$ and $f_{B,s_i} = f$ for all i.

▶ Remark. In the case where $m = 1$ we recover the usual definition of a branching program B computing a function f: if $f(x) = 1$ then B goes from s to an accept node on input x and if $f(x) = 0$ then B goes from s to a reject node on input x.

▶ **Definition 3.** We say that a branching program B is index-preserving if there is an m such that:
1. B has m, start nodes $s_1, \cdots, s_m$, m accept nodes $a_1, \cdots, a_m$, and m reject nodes $r_1, \cdots, r_m$,
2. for all i and all inputs x, if B starts at s_i on input x then it will either end on a_i or r_i.

▶ **Definition 4.** Given a function f.
1. We define $b_m(f)$ to be the minimal size of an index-preserving branching program which computes f m times.
2. We define the amortized branching program complexity $b_{avg}(f)$ of f to be

$$b_{avg}(f) = \lim_{m \to \infty} \frac{b_m(f)}{m}.$$

▶ **Proposition 5.** *For all functions f, $b_{avg}(f)$ is well-defined and is equal to* $\inf\{\frac{b_m(f)}{m} : m \geq 1\}$.

Proof. Note that for all $m_1, m_2 \geq 1$, $b_{m_1+m_2}(f) \leq b_{m_1}(f) + b_{m_2}(f)$ as if we are given a branching program computing f m_1 times and a branching program computing f m_2 times, we can take their disjoint union and this will be a branching program computing f $m_1 + m_2$ times. Thus for all $m_0 \geq 1$, $k \geq 1$, and $0 \leq r < m_0$, $b_{km_0+r}(f) \leq kb_{m_0}(f) + b_r(f)$. This implies that $\lim_{m \to \infty} \frac{b_m(f)}{m} \leq \frac{b_{m_0}(f)}{m_0}$ and the result follows. ◀

3 The Construction

In this section, we give our construction of a branching program computing doubly exponentially many copies of a function f which has linear size per copy of f, proving our main result.

▶ **Theorem 6.** *For all f, $b_{avg}(f) \leq 64n$. In particular, for all f, taking $m = 2^{2^n - 1}$, $b_m(f) \leq 32n2^{2^n}$.*

Proof. Our branching program has several parts. We first describe each of these parts and how we put them together and then we will describe how to construct each part. The first two parts are as follows:
1. A branching program which simultaneously identifies all functions $g : \{0,1\}^n \to \{0,1\}$ that have value 1 for a given x. More preceisely, it has start nodes $s_1, \cdots, s_m$ where $m = 2^{2^n - 1}$ and has one end node t_g for each possible function $g : \{0,1\}^n \to \{0,1\}$, with the guarantee that if $g(x) = 1$ for a given g and x then there exists an i such that that the branching program goes from s_i to t_g on input x.
2. A branching program which simultaneously evaluates all functions $g : \{0,1\}^n \to \{0,1\}$. More precisely, it has one start node s_g for each function g and has end nodes $a'_1, \cdots, a'_m$ and $r'_1, \cdots, r'_m$, with the guarantee that for a given g and x, if $g(x) = 1$ then the branching program goes from s_g to a'_i for some i and if $g(x) = 0$ then the branching program goes from s_g to r'_i for some i.

If f is the function which we actually want to compute, we combine these two parts as follows. The first part gives us paths from $\{s_i : i \in [1, m]\}$ to $\{t_g : g(x) = 1\}$. We now take each t_g from the first part and set it equal to $s_{(f \wedge g) \vee (\neg f \wedge \neg g)}$ in the second part. Once

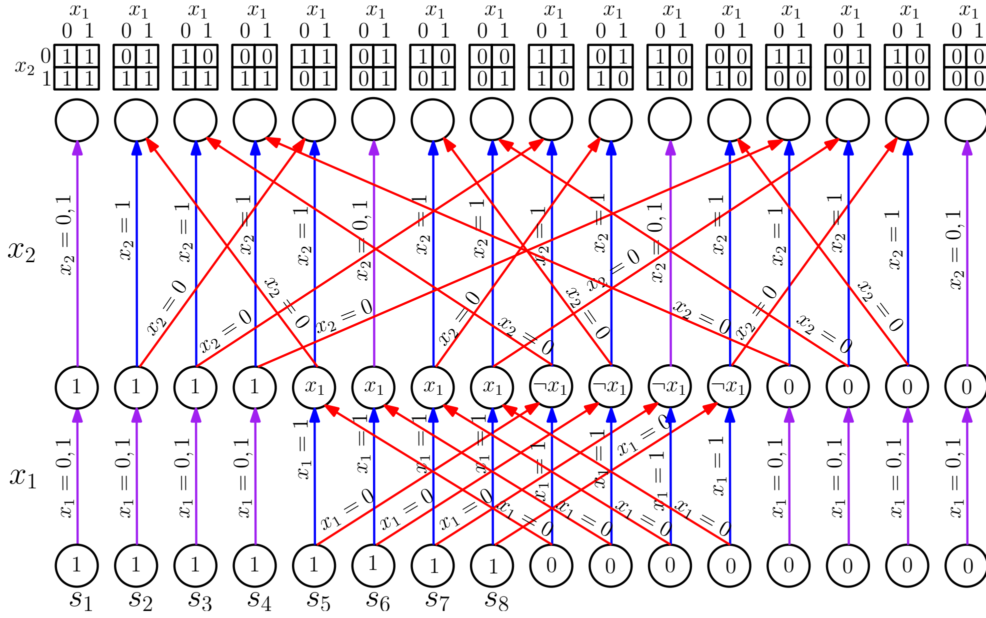

Figure 1 This figure illustrates part 1 of our construction for $n = 2$. The functions for the top vertices are given by the truth tables at the top and each other vertex corresponds to the function inside it. Blue edges can be taken when the corresponding variable has value 1, red edges can be taken when the corresponding variable has value 0, and purple edges represent both a red edge and a blue edge (which are drawn as one edge to make the diagram cleaner). Note that for all inputs x, there are paths from the start nodes to the functions which have value 1 on input x at each level.

we do this, if $f(x) = 1$ then for all g, $g(x) = 1 \iff (f \wedge g) \vee (\neg f \wedge \neg g) = 1$ so we will have paths from $\{t_g : g(x) = 1\} = \{s_{(f\wedge g)\vee(\neg f \wedge \neg g)} : g(x) = 1\}$ to $\{a_i : i \in [1, m]\}$. If $f(x) = 0$ then for all g, $g(x) = 1 \iff (f \wedge g) \vee (\neg f \wedge \neg g) = 0$, so we will have paths from $\{t_g : g(x) = 1\} = \{s_{(f\wedge g)\vee(\neg f \wedge \neg g)} : g(x) = 1\}$ to $\{r_i : i \in [1, m]\}$. Putting everything together, when $f(x) = 1$ we will have paths from $\{s_i : i \in [1, m]\}$ to $\{a_i : i \in [1, m]\}$ and when $f(x) = 0$ we will have paths from $\{s_i : i \in [1, m]\}$ to $\{r_i : i \in [1, m]\}$.

Combining these two parts gives us a branching program B which computes f m times. However, these paths do not have to map s_i to a'_i or r'_i, they can permute the final destinations. In other words, B will not be index-preserving. To fix this, we construct a final part which runs the branching program we have so far in reverse. If we added this final part to B, this would fix the permutation issue but would get us right back where we started! To avoid this, we instead have two copies of the final part, one applied to $\{a'_i : i \in [1, m]\}$ and one applied to $\{r'_i : i \in [1, m]\}$. This separates the case when $f(x) = 1$ and the case $f(x) = 0$, giving us our final branching program.

We now describe how to construct each part. For the first part, which simultaneously identifies the functions which have value 1 on input x, we have a layered branching program with $n+1$ levels going from 0 to n. At level j, for each function $g : \{0,1\}^j \to \{0,1\}$, we have $2^{2^n - 2^j}$ nodes corresponding to g. For all $j \in [1, n]$ we draw the arrows from level $j-1$ to level j as follows. For a node corresponding to a function $g : \{0,1\}^{j-1} \to \{0,1\}$, we draw an arrow with label $x_j = 1$ from it to a node corresponding to a function $g' : \{0,1\}^j \to \{0,1\}$ such that $g'(x_1, \cdots, x_{j-1}, 1) = g(x_1, \cdots, x_{j-1})$. Similarly, we draw an arrow with label $x_j = 0$ from it to a node corresponding to a function $g' : \{0,1\}^j \to \{0,1\}$ such that $g'(x_1, \cdots, x_{j-1}, 0) = g(x_1, \cdots, x_{j-1})$. We make these choices arbitrarily but make sure that no two arrows with the same label have the same destination.

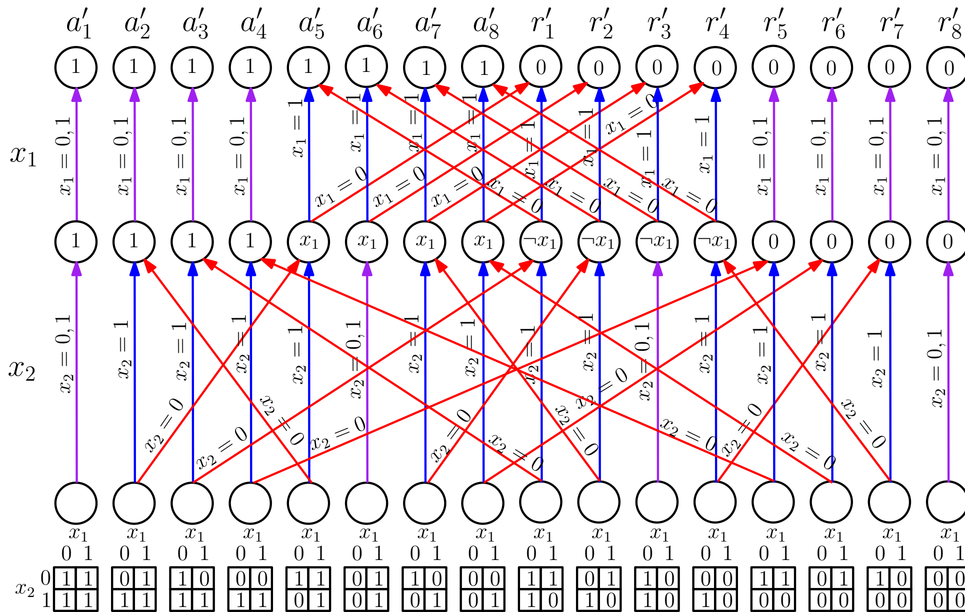

Figure 2 This figure illustrates part 2 of our construction for $n = 2$. The functions for the bottom vertices are given by the truth tables at the bottom and each other vertex corresponds to the function inside it. Note that for all inputs x, the paths go between the functions which evaluate to 1 on input x and the accept nodes and between the functions which evaluate to 0 on input x and the reject nodes.

For the second part, which simultaneously evaluates each function, we have a layered branching program with $n + 1$ levels going from 0 to n. At level $n - j$, for each function $g : \{0,1\}^j \to \{0,1\}$, we have $2^{2^n - 2^j}$ nodes corresponding to g. For all $j \in [1, n]$ we draw the arrows from level $n - j$ to level $n - j + 1$ as follows. For a node corresponding to a function $g : \{0,1\}^j \to \{0,1\}$, we draw an arrow with label $x_j = 1$ from it to a node corresponding to the function $g(x_1, \cdots, x_{j-1}, 1)$ and draw an arrow with label $x_j = 0$ from it to a node corresponding to the function $g(x_1, \cdots, x_{j-1}, 0)$. Again, we make these choices arbitrarily but make sure that no two edges with the same label have the same destination.

For the final part, note that because we made sure not to have any two edges with the same label have the same destination and each level has the same number of nodes, our construction so far must have the following properties:

1. Every vertex has indegree 0 or 2. For the vertices v with indegree 2, there is a j such that one edge going into v has label $x_j = 1$ and the other edge going into v has label $x_j = 0$.
2. The vertices which have indegree 0 are precisely the vertices in the bottom level.

These conditions imply that if we reverse the direction of each edge in the branching program we have so far, this gives us a branching program which runs our branching program in reverse. As described before, we now take two copies of this reverse program. For one copy, we take its start nodes to be $a'_1, \cdots, a'_m$ and relabel its copies of $s_1, \cdots, s_m$ as $a_1, \cdots, a_m$. For the other copy, we take its start nodes to be $r'_1, \cdots, r'_m$ and relabel its copies of $s_1, \cdots, s_m$ as $r_1, \cdots, r_m$. ◀

4 Relationship to catalytic computation

In catalytic computation, we have additional memory which we may use but this memory starts with unknown contents and we must restore this memory to its original state at the

end. Our result is related to catalytic computation through Proposition 9 of [5], which says the following

▶ **Proposition 7.** *Let f be a function which can be computed in space $s(n)$ using catalytic tape of size $l(n) \leq 2^{s(n)}$. Then $b_{2^{l(n)}}(f)$ is $2^{l(n)} \cdot 2^{O(s(n))}$.*

For convenience, we give a proof sketch of this result here.

Proof Sketch. This can be proved using the same reduction that is used to reduce a Turing machine using space $s(n)$ to a branching program of size $2^{O(s(n))}$, with the following differences. There are $2^{l(n)}$ possibilities for what is in the catalytic tape at any given time, so the resulting branching program is a factor of $2^{l(n)}$ times larger. The requirement that the catalytic tape is restored to its original state at the end implies that there must be $2^{l(n)}$ disjoint copies of the start, accept, and reject nodes, one for each possibility for what is in the catalytic tape originally. This means that the branching program computes f $2^{l(n)}$ times. Finally, the condition that $l(n) \leq 2^{s(n)}$ is necessary because otherwise the branching program would have to be larger in order to keep track of where the pointer to the catalytic tape is pointing! ◀

Girard, Koucký, and McKenzie [5] conjectured that for a random function f, for all $m \geq 1$, $b_m(f)$ is $\Omega(mb_1(f))$. If true, this conjecture would imply (aside from issues of non-uniformity) that a catalytic tape does not significantly reduce the space required for computing most functions. However, our construction disproves this conjecture.

That said, our construction requires m to be doubly exponential in n. It is quite possible that $\log(\frac{b_m(f)}{m})$ is $\Omega(\log(b_1(f)))$ for much smaller m, which would still imply (aside from issues of non-uniformity) that a catalytic tape does not reduce the space required for computing most functions by more than a constant factor.

5 Barrier for input-based bottleneck arguments

As noted in the introduction, our result rules out any general lower bound approach which would prove lower bounds on amortized branching program complexity as well as branching program size. In this section, we discuss one such class of techniques.

One way we could try to show lower bounds on branching programs is as follows. We could argue that for the given function f and a given branching program B computing f, for every YES input x the path that B takes on input x contains a vertex giving a lot of information about x and thus G must be large to accomodate all of the possible inputs. We observe that this kind of argument would show lower bounds on amortized branching program complexity as well as on branching program size and thus cannot show nontrivial general lower bounds. We assume for the remainder of the section that we are trying to compute some function $f : \{0,1\}^n \to \{0,1\}$ with a branching program of a given type.

▶ **Definition 8.** We define a function description h to be a mapping which takes a branching program B and assigns a funtion $h_v : \{0,1\}^n \to \Omega$ to every vertex v of B. Here Ω can be an arbitrary set but we will focus on the case when $\Omega = \{0,1\}$.

▶ **Example 9.** A particularly useful function description is the reachability function description which sets $h_v(x) = 1$ if it is possible to reach v from a start node on input x and sets $h_v(x) = 0$ otherwise.

▶ **Definition 10.** We define a bottleneck criterion c to be a mapping which takes a function $g : \{0,1\}^n \to \Omega$ and an $x \in \{0,1\}^n$ and outputs 0 or 1. The idea is that $c(g,x) = 1$ if g gives a lot of information about x and $c(g,x) = 0$ otherwise.

▶ **Definition 11.** We say that a function description and bottleneck criterion (h, c) are valid for a given type of branching program if for any branching program B of this type, any YES input x, and any path P in B from a start node to an accept node on input x there is a vertex $v \in V(P)$ such that $c(h_v, x) = 1$.

▶ **Definition 12.** We say that a bottleneck criterion c has selectivity S_c if there is a set of YES inputs I such that for all g, there are at most $\frac{|I|}{S_c}$ inputs $x \in I$ such that $c(g, x) = 1$

We note that bottleneck criteria with high selectivity imply large lower size bounds on the given type of branching program.

▶ **Proposition 13.** *If there exists a valid function description and bottleneck criterion (h, c) for a given type of branching program and c has selectivity S_c then any branching program of the given type computing f must have size at least S_c.*

We now observe that bottleneck criteria with high selectivity also imply large lower size bounds on amortized branching programs.

▶ **Lemma 14.** *If there exists a valid function description and bottleneck criterion (h, c) for a given type of branching program and c has selectivity S_c then any branching program of the given type computing f m times has size at least mS_c*

Proof. Let B be a branching program computing f m times. Let N be the total number of times that we have a vertex $v \in V(B)$ and an input $x \in I$ such that $c(h_v, x) = 1$. On the one hand, $N \geq m|I|$ as for each $x \in I$ there are m disjoint paths in B from a start node to an accept node, each of which must contain a vertex v such that $c(h_v, x) = 1$ (as otherwise (h, c) wouldn't be valid). On the other hand, since c has selectivity S_c, for each $v \in V(B)$ there are at most $\frac{|I|}{S_c}$ $x \in I$ such that $c(h_v, x) = 1$. Thus, $N \leq \frac{|V(B)| \cdot |I|}{S_c}$. Putting these two bounds together, we have that $\frac{|V(B)| \cdot |I|}{S_c} \geq m|I|$ which implies that $|V(B)| \geq mS_c$, as needed. ◀

▶ **Corollary 15.** *Given a valid function description and bottleneck criterion (h, c) for general branching programs, $S_c \leq 64n$*

Proof. By Lemma 14, if (h, c) is a valid function description and bottleneck criterion and c has selectivity S_c then for all m, $b_m(f) \geq mS_c$. However, Theorem 6 says that for $m = 2^{2^n - 1}$, $b_m(f) \leq 64mn$. Thus, we must have that $S_c \leq 64n$, as needed. ◀

▶ Remark. To prove such an upper bound on S_c it is sufficient to find a B which computes f m times. B does not have to be index-preserving. As noted in the proof of Theorem 6, the first two parts of the construction in Theorem 6 are sufficient to construct such a B. Thus, we have the same upper bound on S_c even for oblivious read-twice branching programs (a branching program is oblivious if it reads the input bits in the same order regardless of the input)!

5.1 Examples of input-based bottleneck arguments

In this subsection, we briefly discuss which lower bound approaches can be viewed as input-based bottleneck arguments. In particular, we note that the current framework of Potechin and Chan [3] for analyzing monotone switching networks can be seen as an input-based bottleneck argument, as can many lower bounds on read-once branching programs.

▶ **Example 16** (Fourier analysis on monotone switching networks)**.** At a high level, the current framework of Potechin and Chan [3] for analyzing monotone switching networks works as follows:

1. Take I to be a large set of minimal YES inputs which are almost disjoint from each other.
2. Use the reachability function description, focusing on the maximal NO instances (the cuts).
3. Carefully construct a set of functions g_{xi} for each $x \in I$ and use the criterion $c(h_v, x) = 1$ if $|\langle g_{xi}, h_v \rangle| \geq \frac{a}{l}$ for some i and is 0 otherwise, where $a > 0$ is a constant and l is the maximum length of an accepting path in the switching network. The intuition is that the functions g_{xi} pick out high-degree information about the input x which much be processed to accept x, so any accepting path for x must contain a vertex v where $|\langle g_{xi}, h_v \rangle|$ is large for some i.
4. Use Fourier analysis to argue that c has high selectivity.

▶ **Example 17** (k-clique)**.** Wegener [14] and Zak [15] independently proved exponential lower bounds on the size of read-once branching programs solving the problem of whether a graph G has a clique of linear size. To prove their lower bounds, they argue that near the start node, the branching program must branch off like a tree or else it cannot be completely accurate. This kind of argument is not captured with an input-based bottleneck argument, as it uses the structure of the given braching program. That said, we can prove a $\frac{\binom{n}{k}}{n-k+1}$ lower size bound on read-once branching programs for k-clique with the following input-based bottleneck argument:

1. We take c to be the following bottleneck criterion. We take I to be the set of minimal YES instances, i.e. graphs which have a clique of size k and no other edges. Since we are considering a read-once branching program, for each node v we have a partition of the input bits based on whether they are examined before or after reaching v (input bits which are never examined on any computation path containing v can be put in either side). Given an $x \in I$, this induces a partition (E_1, E_2) of the edges of the k-clique in x. We take $c(v, x) = 1$ if there is a path from a start node to an accept node on input x which passes through v and we have that E_1 contains $k - 2$ of the edges incident to some vertex u in the k-clique but there is no vertex u in the k-clique such that E_1 contains all $k - 1$ edges incident with u.
2. We argue that c has high selectivity as follows. If $c(v, x) = c(v, x') = 1$ for some v, x, x' then consider the corresponding partitions (E_1, E_2) and (E'_1, E'_2). $E_1 \cup E'_2$ and $E'_1 \cup E_2$ must form k-cliques so we must have that $|E_1| = |E'_1|$ and $|E_2| = |E'_2|$ and $E_1 \cup E'_2, E'_1 \cup E_2$ contain no extra edges.
   Now let A be the set of vertices w of the k-clique in x such that both E_1 and E_2 contain some but not all of the edges incident to w in x. We observe that the k-clique in x' contains A as otherwise the edges in $E_1 \cup E'_2$ and $E'_1 \cup E_2$ incident to w are wasted which is impossible as $E_1 \cup E'_2$ and $E'_1 \cup E_2$ can have no extra edges. Thus, we can only have $c(v, x') = 1$ for the x' such that the k-clique contains A.
   From the definition of c, A must include some vertex u in the k-clique of x and $k - 2$ of its neighbors, so $|A| \geq k - 1$. This implies that $c(v, x') = 1$ for at most $n - k + 1$ distinct x'. The total number of $x' \in I$ is $\binom{n}{k}$ so our lower bound is $\frac{\binom{n}{k}}{n-k+1}$.

▶ **Example 18** (Majority)**.** In his master's thesis introducing branching programs, Masek [7] proved a quadratic lower bound on the size of read-once branching programs computing the majority function. With an input-based bottleneck argument, we obtain a lower bound of $\Omega(n^{\frac{3}{2}})$, which is lower, but there is a reason for this. We can prove our lower bound as follows. Here we assume that $n \geq 3$ is odd.

1. We take I to be the set of inputs with exactly $\frac{n+1}{2}$ ones.
2. We note that in order for the branching program to be read-once and be correct on all inputs from all starting nodes, we must have that for each node v of the branching program, there is a partition (A, B) of the inputs bits such that on all paths from a start node to v, only input bits in A are examined and on all paths from v to an accept node or reject node, only inputs in B are examined. Moreover, if $|A| < \frac{n}{2}$ then every path from a start node to v must examine all of the bits in A and must have the same number of these bits equal to one.
3. We choose an $m < \frac{n}{2}$ and take $c(v, x) = 1$ if $|A| = m$ and there is a path from a start node to v on input x and we take $c(v, x) = 0$ otherwise. Note that for any vertex v, all of the x such that $c(v, x) = 1$ have the same number of ones in A so there can be at most $O(\frac{|I|}{\sqrt{m}})$ such x and c has selectivity $\Omega(\sqrt{m})$.
4. We sum this over all $m < \frac{n}{2}$ obtaining our final lower bound of $\Omega(n^{\frac{3}{2}})$

▶ Remark. With a more complicated argument, it can be shown that this lower bound applies even if we allow the branching program to reject a small portion of the YES inputs (while still requiring that it rejects all NO inputs). This is a reason why we only obtain a lower bound of $\Omega(n^{\frac{3}{2}})$ rather than $\Omega(n^2)$; we can probabilistically choose a branching program of size $O(n^{\frac{3}{2}}log(n))$ for majority which rejects all NO inputs and accepts any given YES input with very high probability.

▶ Remark. If we assume that our branching program is oblivious as well as read-once (in which case we can assume without loss of generality that the input bits are read in order) then we can prove an $\Omega(n^2)$ lower bound using an input-based bottleneck argument. The idea is that we can take a different set of inputs I in order to increase the selectivity of our bottleneck criterion c. In particular, for each m we can take a set of inputs I_m such that I_m contains $m + 1$ minimal YES inputs, each with a different number of ones in the first m bits. This c now has selectivity m and summing over all $m < \frac{n}{2}$ gives a lower bound of $\Omega(n^2)$

As these examples show, input-based bottleneck arguments are effective for proving lower bounds on read-once branching programs. Thus, Theorem 6 and Lemma 14, which together rule out input-based bottleneck arguments even for oblivious read-twice branching programs, provide considerable insight into the spike in difficulty between proving lower bounds for read-once branching programs and read-twice branching programs which can be seen in Razborov's survey [9] on branching programs and related models. That said, Theorem 6 and Lemma 14 do not say anything about lower bounds based on counting functions such as Nechiporuk's quadratic lower bound [8] or lower bounds based on communication complexity arguments such as Babai, Nisan, and Szegedy's result [1] proving an exponential lower bound on oblivious read-k branching programs for arbitrary k. We also note that Cook, Edmonds, Medabalimi, and Pitassi [4] give another explanation for the failure of bottleneck arguments past read-once branching programs.

6 Linear upper bound on complexity measures

Another way we could try to lower bound branching program size is through a complexity measure on functions. However, Razborov [10] showed that submodular complexity measures cannot have superlinear values. In this section we show that this is also true for a similar class of complexity measures, branching complexity measures, which correspond more closely to branching programs. We then show that all submodular complexity measures are also branching complexity measures, so Theorem 6 is a constructive analogue and a slight generalization of Razborov's result [10].

▶ **Definition 19.** We define a branching complexity measure μ_b to be a measure on functions which satisfies the following properties:

1. $\forall i, \mu_b(x_i) = \mu_b(\neg x_i) = 1$,
2. $\forall f, \mu_b(f) \geq 0$,
3. $\forall f, i, \mu_b(f \wedge x_i) + \mu_b(f \wedge \neg x_i) \leq \mu_b(f) + 2$,
4. $\forall f, g, \mu_b(f \vee g) \leq \mu_b(f) + \mu_b(g)$.

▶ **Definition 20.** Given a node v in a branching program, define $f_v(x)$ to be the function such that $f_v(x)$ is 1 if there is a path from some start node to v on input x and 0 otherwise. Note that for any start node s, $f_s = 1$.

▶ **Lemma 21.** *If μ_b is a branching complexity measure then for any branching program, the number of non-end nodes which it contains is at least*

$$\frac{1}{2}\left(\sum_{t:t \text{ is an end node}} \mu_b(f_t) - \sum_{s:s \text{ is a start node}} \mu_b(f_s)\right).$$

Proof. Consider what happens to $\sum_{t:t \text{ is an end node}} \mu_b(f_t) - \sum_{s:s \text{ is a start node}} \mu_b(f_s)$ as we construct the branching program. At the start, when we only have the start nodes and these are also our end nodes, this expression has value 0. Each time we merge end nodes together, this can only decrease this expression. Each time we branch off from an end node, making the current node a non-end node and creating two new end nodes, this expression increases by at most 2. Thus, the final value of this expression is at most twice the number of non-end nodes in the final branching program, as needed. ◀

▶ **Corollary 22.** *For any branching complexity measure μ_b and any function f, $\mu_b(f) \leq 130n$*

Proof. By Lemma 21 we have that for all $m \geq 1$, $\frac{m\mu_b(f) - m\mu_b(1)}{2} \leq m \cdot b_m(f)$. Using Theorem 6 and noting that $\mu_b(1) \leq 2$ we obtain that $\mu_b(f) \leq 130n$. ◀

Finally, we note that every submodular complexity measure μ_s is a branching complexity measure, so Corollary 22 is a slight generalization of Razborov's result [10] (though with a worse constant).

▶ **Definition 23.** A submodular complexity measure μ_s is a measure on functions which satisfies the following properties:

1. $\forall i, \mu(x_i) = \mu(\neg x_i) = 1$,
2. $\forall f, \mu(f) \geq 0$,
3. $\forall f, g, \mu_s(f \vee g) + \mu_s(f \wedge g) \leq \mu_s(f) + \mu_s(g)$.

▶ **Lemma 24.** *Every submodular complexity measure μ_s is a branching complexity measure.*

Proof. Note that

$$\mu_s(f \vee x_i) + \mu_s(f \wedge x_i) \leq \mu_s(f) + \mu_s(x_i)$$

and

$$\mu_s((f \vee x_i) \wedge \neg x_i) + \mu_s((f \vee x_i) \vee \neg x_i) = \mu_s(f \wedge \neg x_i) + \mu_s(1) \leq \mu_s(f \vee x_i) + \mu_s(\neg x_i).$$

Combining these two inequalities we obtain that

$$\mu_s(f \wedge \neg x_i) + \mu_s(1) + \mu_s(f \wedge x_i) \leq \mu_s(f) + \mu_s(x_i) + \mu_s(\neg x_i)$$

which implies that $\mu_s(f \wedge \neg x_i) + \mu_s(f \wedge x_i) \leq \mu_s(f) + 2 - \mu_s(1) \leq \mu_s(f) + 2$, as needed. ◀

7 Conclusion

In this paper, we showed that for any function f, there is a branching program computing a doubly exponential number of copies of f which has linear size per copy of f. This result shows that in the branching program model, any operation/function can be amortized with sufficiently many copies. This result also disproves a conjecture about nonuniform catalytic computation, rules out certain approaches for proving general lower space bounds, and gives a constructive analogue of Razborov's result [10] on submodular complexity measures.

However, the number of copies required in our construction is extremely large. A remaining open problem is to determine whether having a doubly exponential number of copies is necessary or there a construction with a smaller number of copies. Less ambitiously, if we believe but cannot prove that a doubly exponential number of copies is necessary, can we show that a construction with fewer copies would have surprising implications?

Acknowledgments. The author would like to thank Avi Wigderson and Venkatesh Medabalimi for helpful conversations.

References

1 Laszlo Babai, Noam Nisan, and Mario Szegedy. Multiparty Protocols, Pseudorandom Generators for Logspace, and Time-Space trade-offs. *Journal of Computer and System Sciences*, 45:204–232, 1992.

2 Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *46th Annual Symposium on the Theory of Computing (STOC 2014)*, pages 857–866, 2014. doi:10.1145/2591796.2591874.

3 Siu Man Chan and Aaron Potechin. Tight Bounds for Monotone Switching Networks via Fourier Analysis. *Theory of Computing*, 10:389–419, 2014. doi:10.1145/2213977.2214024.

4 Stephen Cook, Jeff Edmonds, Venkatesh Medabalimi, and Toniann Pitassi. Lower Bounds for Nondeterministic Semantic Read-Once Branching Programs. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *LIPIcs*, pages 36:1–36:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.36.

5 Vincent Girard, Michal Koucký, and Pierre McKenzie. Nonuniform catalytic space and the direct sum for space, 2015. URL: https://eccc.weizmann.ac.il/report/2015/138/.

6 Oleg Lupanov. The synthesis of contact circuits. *Doklady Akademii Nauk SSSR*, 119:23–26, 1958.

7 William Masek. A fast algorithm for the string editing problem and decision graph complexity. Master's thesis, MIT, 1976.

8 E. I. Nechiporuk. On a Boolean function. *Soviet Mathematics Doklady*, 7(4):999–1000, 1966.

9 Alexander Razborov. Lower bounds for deterministic and nondeterministic branching programs. In *Proceedings of the 8th International Symposium on Fundamentals of Computation Theory (FCT)*, volume 529 of *LNCS*, pages 47–60, 1991.

10 Alexander Razborov. On submodular complexity measures. In *Proceedings of the London Mathematical Society symposium on Boolean function complexity*, pages 76–83, 1992.

11 Claude Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.

12 Dietmar Uhlig. On the synthesis of self-correcting schemes for functional elements with a small number of reliable elements. In *Mathematical Notes of the Academy of Sciences of the USSR*, volume 16, pages 558–562, 1974.
13 Dietmar Uhlig. Networks computing boolean functions for multiple input values. In *Proceedings of the London Mathematical Society Symposium on Boolean Function Complexity*, pages 165–173, 1992.
14 Ingo Wegener. On the complexity of branching programs and decision trees for clique functions. *Journal of the Association for Computing Machinery (JACM)*, 35(2):389–419, 1988. doi:10.1145/42282.46161.
15 Stanislav Zak. An exponential lower bound for one-time-only branching programs. In *Proceedings of the Mathematical Foundations of Computer Science (MFCF)*, pages 562–566, 1984.