Unitary Branching Programs: Learnability and Lower Bounds

Fidel Ernesto Díaz Andino [1] **Maria Kokkou** [2] **Mateus de Oliveira Oliveira** [3] **Farhad Vadiee** [3]

Abstract

Bounded width branching programs are a formalism that can be used to capture the notion of non-uniform constant-space computation. In this work, we study a generalized version of bounded width branching programs where instructions are defined by unitary matrices of bounded dimension. We introduce a new learning framework for these branching programs that leverages on a combination of local search techniques with gradient descent over Riemannian manifolds. We also show that gapped, read-once branching programs of bounded dimension can be learned with a polynomial number of queries in the presence of a teacher. Finally, we provide explicit near-quadratic size lower-bounds for bounded-dimension unitary branching programs, and exponential size lower-bounds for bounded-dimension read-once gapped unitary branching programs. The first lower bound is proven using a combination Nečiporuk's lower bound technique with classic results from algebraic geometry. The second lower bound is proven within the framework of communication complexity theory.

1. Introduction

Bounded width branching programs, also known as non-uniform finite automata, may be regarded as model of computation that generalizes the notion of constant space computation to the non-uniform setting (Nakanishi et al., 2000; Barrington, 1989; Ergün et al., 1995). A celebrated result due to Barrington states that any Boolean function $f : \{0,1\}^n \to \{0,1\}$ that can be computed by a Boolean circuit of depth d can also be computed by a width-5 branching program of length 4^d (Barrington, 1989). As a consequence, functions that can be computed by logarithmic depth Boolean circuits can be computed by width-5 branching programs of polynomial length.

The problem of constructing branching programs consistent with a given dataset has been studied under a large variety of paradigms (Bergadano et al., 1997; Mansour & McAllester, 2002; Ergün et al., 1995; Raghavan & Wilkins, 1993; Bshouty et al., 1998). In this work, we address the problem of constructing bounded-width branching programs consistent with a given input dataset from the perspective of continuous optimization theory. We formalize branching programs according to Barrington's M-program model, where branching programs are viewed as a sequence of tuples of elements of a monoid M (i.e., a semigroup with identity) (Bédard et al., 1993; Cleve, 1991; Caussinus, 1996; Barrington, 1989). This point of view allows us to consider branching programs over matrix groups, and in particular over the unitary group $U(k)$, i.e., the group of $k \times k$ unitary matrices with matrix multiplication as the operation.

For each fixed $k \in \mathbb{N}$, the unitary group $U(k)$ is a compact and connected topological group, that has has the structure of a Riemannian manifold, known as the *complex Stiefel manifold* (Adams & Walker, 1965). This allows us to view the problem of searching for a branching program consistent with a given dataset as the problem of minimizing function with unitary constraints (Abrudan et al., 2008; 2009; 2008). The caveat is that such problems can be reformulated as minimization problems in the Stiefel manifold, and therefore one can leverage on the machinery of Riemannian gradient descent to search for optimal solutions.

Still, when the number of variables become too large (meaning a few hundreds), state-of-the-art tools that implement Riemannian gradient descent algorithms become impractical. To mitigate this computational intractability, we combine Riemannian gradient descent with a simple local optimization technique inspired on the paradigm of waveform relaxation (Lelarasmee et al., 1982; Janssen & Vandewalle, 1997; Crow & Ilic, 1990). More precisely, we start by fixing the values of a large number of variables in the system. Subsequently, we compute an optimal solution with respect to the remaining variables. At this point, the values of the optimized variables are fixed, and a new small set of variables to be optimized is selected. The process is repeated until no improvement is observed. Interestingly, this algorithm works

[1]University of São Paulo, São Paulo, Brazil [2]Chalmers University of Technology, Gothenburg, Sweden [3]University of Bergen, Bergen, Norway. Correspondence to: Mateus de Oliveira Oliveira <mateus.oliveira@uib.no>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

surprisingly well for the task of learning unitary branching programs consistent with sparse unstructured datasets. More precisely, our heuristic was able to learn read-once unitary branching programs of dimension 3 consistent with datasets containing n positive and n negative binary strings, each of length n, with near 0% error. In our experiments, n was chosen from the set $\{16, 32, ..., 1024\}$ and the time for learning the branching programs varied from a few seconds (for $n = 16$) to about 72 hours (for $n = 1024$). It is worth noting that the time to learn these datasets with 10% error was substantially smaller (less than 4 hours).

Besides our learning framework, we also study connections between unitary branching programs and the theory of learning from a minimally adequate teacher. In particular, we show that gapped, read-once branching programs of bounded dimension can be learned with a polynomial number of queries in the presence of a teacher. Finally, we provide explicit near-quadratic size lower-bounds for bounded-dimension unitary branching programs, and exponential size lower-bounds for bounded-dimension read-once gapped unitary branching programs. The first lower bound is proven using a combination Nečiporuk's lower bound technique with classic results from algebraic geometry. The second lower bound is proven within the framework of communication complexity theory.

2. Unitary Branching Programs

We denote by $\mathbb{N}$, $\mathbb{R}$ and $\mathbb{C}$ the sets of natural numbers, real numbers and complex numbers respectively. The set of positive natural numbers is denoted by $\mathbb{N}_+$. For each $n \in \mathbb{N}_+$, we let $[n] \doteq \{1, \ldots, n\}$ and $[\![n]\!] = \{0, \ldots, n-1\}$. Given a vector V we may write $V[i]$ to denote the i-th entry of V.

A *semigroup* is a nonempty set G endowed with an associative binary operation of type $G \times G \to G$. If a and b are elements of G, then we denote by $a \cdot b$ the result of applying the semigroup operation to the ordered pair (a, b).

Branching Program. Let G be a semigroup and $s \in \mathbb{N}_+$. An (s, G)-*instruction* is a tuple $I = (A_0, \ldots, A_{s-1})$ of elements from G. An n-input branching program over G of length ℓ, arity s, and class size c is a tuple $B = (\mathcal{J}, \mathcal{I}, C)$ where $\mathcal{J} = (j_1, \ldots, j_\ell)$ is a sequence of *indices* from $[n]$, $\mathcal{I} = (I_1, \ldots, I_\ell)$ is a sequence of (s, G)-instructions, and $C = (C_0, \ldots, C_{c-1})$ is sequence of pairwise disjoint subsets of G.

Given a string $W \in [\![s]\!]^n$, the value of B on W is defined as follows.

$$value(B, W) = I_1[W[j_1]] \cdot I_2[W[j_2]] \cdot \ldots \cdot I_\ell[W[j_\ell]]. \quad (1)$$

Intuitively, the $value(B, W)$ is obtained by multiplying a sequence of ℓ elements from the semigroup G. For each $r \in [\ell]$, the r-th factor of this multiplication is the $W[j_r]$-th element of the instruction I_r.

We say that a branching program B computes a function $f : [\![s]\!]^n \to [\![c]\!]$ if for each string $W \in [s]^n$, the element $value(B, W)$ belongs to the set $C_{f(W)}$.

A celebrated complexity theoretic result due to Barrington, states that if an n-input Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a fan-in-2 Boolean circuit of depth d, then f can also be computed by an n-input branching program over the symmetric group $\mathbb{S}_5$ of length 4^d (Barrington, 1989). This implies that Boolean functions that can be computed by circuits of logarithmic depth can be computed by polynomial-size branching programs over the group $\mathbb{S}_5$. The importance of Barrington's theorem stems from the fact that many interesting Boolean functions can be computed by polynomial-size circuits of logarithmic depth, including arithmetic functions such as addition, multiplication, powering and division (Chiu et al., 2001; Beame et al., 1986), several cryptographic functions (Viola, 2009) and a large number of combinatorial problems (Elberfeld et al., 2012).

2.1. Unitary Branching Programs

Our main object of study is the notion of *unitary branching program*. Intuitively, unitary branching programs are precisely the branching programs defined above, except for the fact that we need to be more specific in the way we partition the unitary group $U(k)$ into classes. We choose a standard approach. Namely, if a branching program B has class size c, then we fix c representative unitary matrices $X_0, \ldots, X_{c-1}$ and say that an input string W belongs to class i if the unitary matrix $value(B, W)$ is closer to X_i than to any other representative matrix. In case of ties, we choose the class of smallest index.

Distance Between Matrices. Given complex vectors x and y, we let $\langle x, y \rangle = \sum_i x_i \cdot \overline{y}_i$ denote the *inner product* of x and y. Given unitary matrices U with columns $u_1, \ldots, u_k$, and W with columns $v_1, \ldots, v_k$, we define the *distance* between U and W as

$$\mathsf{MD}(U, W) = \sum_{i=1}^{k} (1 - \langle u_i, v_i \rangle)(1 - \overline{\langle u_i, v_i \rangle}).$$

Please note that $\mathsf{MD}(U, W)$ is always a non-negative real value. This notion of distance will be used for two purposes. First, to define a notion of distance for unitary branching programs. Second, to partition the space of unitary matrices into a finite number of regions.

Partitioning the Unitary Group $U(k)$. Given a tuple $\zeta = (X_0, \ldots, X_{c-1})$ of pairwise distinct $k \times k$ unitary

matrices, we partition the space $U(k)$ into c regions $C_\zeta = (C_0, \ldots, C_{c-1})$ as follows. For each matrix $Y \in U(k)$, we let Y belong to C_i if i is the minimum index in $[\![c]\!]$ with the property that $\mathsf{MD}(Y, X_i) \leq \mathsf{MD}(Y, X_j)$ for every $j \in [\![c]\!]$. Intuitively, a matrix Y belongs to C_i if X_i is the closest matrix to Y among the matrices in the sequence ζ. If this closest distance is achieved by more than one matrix in ζ, then Y is added to the cell of the partition corresponding to the matrix X_i of smallest index.

Unitary Branching Programs. A k-dimensional unitary branching program is a branching program $B = (\mathcal{J}, \mathcal{I}, C)$ over the group $U(k)$, where $C = (C_1, \ldots, C_c)$ is the partition of $U(k)$ induced by some tuple $\zeta = (X_0, \ldots, X_{c-1})$ of pairwise distinct unitary matrices. We say that ζ is the class sequence of B.

3. Learning Unitary Branching Programs Using Gradient Descent

In this section, we introduce a new heuristic for learning unitary branching programs consistent with a given dataset. The idea is to view the search for a separating branching program as a continuous optimization problem and to leverage on the power of gradient descent algorithms. Unitary matrices enjoy several properties that are relevant in our context. First, the set of unitary matrices of a given dimension forms a group, and therefore, is closed under multiplication. Second, the determinant of an unitary matrix is a complex number of norm 1. This is important for stability reasons since computing with branching programs requires the multiplication of a long sequence of matrices. Third, the group of permutation matrices form a subgroup of unitary matrices, and therefore, the machinery from Barrington's theorem can be simulated by branching programs over the unitary group. Third, the unitary group $U(k)$ is endowed with the relative topology as a subset of $\mathcal{X}(k, \mathbb{C})$, the set of all $k \times k$ complex matrices. Notice that $\mathcal{X}(k, \mathbb{C})$ is homeomorphic to the euclidean space. As a topological space, $U(k)$ is both compact and connected. Additionally, $U(k)$ has the structure of a Riemannian manifold called the *complex Stiefel manifold*. Therefore, instead of considering the problem of finding an optimal branching program as an optimization problem with unitary constraints over the euclidean space, we will use Riemannian gradient descent algorithms to view the learning problem for branching programs as an unconstrained optimization problem over the complex Stiefel manifold. We note that our choice of the unitary group over the orthogonal group stems from the fact that the former is connected while the latter is not. In particular, for each $m \in \mathbb{N}$ the space formed by the Cartesian product of m copies of the unitary group still has a unique component, the product of m copies of the orthogonal group has 2^m connected components. Also note that the *special* orthogonal group, the subgroup of the orthogonal group containing matrices with determinant equal to 1 is indeed connected. Nevertheless, restricting instructions to use such matrices may restrict the power of the branching program, since for instance, permutation matrices corresponding to odd permutations do not belong to this group.

Still, given the high dimensionality of our data, simply formulating the problem as an optimization problem on Riemannian manifolds turns to be intractable to current state of the art optimization tools. To counter this drawback, we follow an approach similar to the one used in the context of waveform relaxation algorithms. We start by guessing random solution, and by using this random solution to fix the value of all but a small fraction of the variables. Subsequently, we apply the Riemannian optimization algorithm only on this small fraction of variables. Once this optimum has been achieved, we select a new small fraction of variables to optimize. We keep repeating this process until time is up or until we have reached a solution with a good enough accuracy. In our case, the variables chosen to be optimized at each time step are either those corresponding to a small set of instructions of the unitary branching program, or those corresponding to the class matrices.

Datasets. An n-input dataset of arity s and class-size c is a tuple $D = (D_0, \ldots, D_{c-1})$ where for each $i \in [\![c]\!]$, D_i is a subset of $[\![s]\!]^n$ and for each $i, j \in [\![c]\!]$, $D_i \cap D_j = \emptyset$. We say that an n-input branching program B of arity s and class size c is *consistent* with D if for each $i \in [\![c]\!]$, and each string $W \in D_i$, the matrix $value(B, W)$ belongs to C_i.

Discrete Distance. It will be convenient to define a measure of how well a branching program approximates a dataset D. We define the discrete distance between B and D as the value

$$\mathsf{DD}(B, D) = \max_{i \in [\![c]\!]} \frac{|\{W \in D_i \ : \ value(B, W) \notin C_i\}|}{|D_i|}.$$

In other words, for each class i we consider the ratio between the number of strings in D_i that were incorrectly classified by the branching program and the size of D_i. The distance between B and D is the maximum value of these ratios. Note that if the branching program is consistent with the branching program then the distance is zero, while if the branching program makes mistakes in every string belonging to some class, then the distance is 1.

Continuous Distance. For purposes of optimization, we also define a continuous real-valued distance function between unitary branching programs and datasets. This will be the function to be optimized by our Riemannian gradient

descent algorithm.

$$\mathsf{CD}(B, D) = \sum_{i=0}^{c-1} \sum_{W \in D_i} \frac{\mathsf{MD}(value(B, W), X_i)}{|D_i|}. \quad (2)$$

Learning Unitary Branching Programs The problem of learning a k-dimensional unitary branching program B compatible with a given n-input dataset D can be formulated as a continuous optimization problem with unitary constraints.

More precisely, given a sequence $\mathcal{J} = (j_1, \ldots, j_l)$ of indices from $[n]$, and an n-input dataset D of arity s and class size c, the goal is to find sequences[1]

$$\mathcal{I} = (A_0^0, \ldots, A_{s-1}^0, \ldots, A_0^l, \ldots, A_{s-1}^l)$$

and

$$\zeta = (X_0, \ldots, X_{c-1})$$

of k-dimensional unitary matrices such that the branching program $B = (\mathcal{J}, \mathcal{I}, C_\zeta)$ minimizes the continuous distance function $\mathsf{CD}(B, D)$.

For each k, the unitary group $U(k)$ has the structure of a manifold, known as the $k \times k$ complex Stiefel manifold. This allows one to reformulate the problem of solving an optimization problem with a single $k \times k$ unitary constraint as an optimization problem over the $k \times k$ complex Stiefel manifold comprising of k^2 complex variables (Absil et al., 2009). More generally, a problem with several $k \times k$ unitary constraints can be reformulated as an optimization problem over a Cartesian product of Stiefel manifolds. In our case, this amounts to solving an optimization problem with $k^2 \cdot (l \cdot s + c)$ complex variables over the manifold $U(k)^{l \cdot s + c}$, since we have $l \cdot s + c$ constraints over $U(k)$. More precisely, this corresponds to optimizing over the unitary matrices $A_0^1, \ldots, A_{s-1}^l$ defining the instructions of the branching programs and the matrices $X_0, \ldots, X_{c-1}$ defining representatives for the partition of the space $U(k)$ used by the branching program.

Optimization problems over the complex Stiefel manifolds, and many other well studied manifolds, can be easily specified using open source tool boxes such as Manopt (Boumal et al., 2014) or ROPTLIB (Huang et al., 2018). In particular, our optimization problem can be easily specified in using Matlab in conjunction with Manopt or in C++ in conjunction with ROPTLIB. The problem is that a direct specification in this setting does not scale well with the length of the strings in the dataset. In particular, the task of learning read-once unitary branching programs over the group $U(2)$ compatible with randomly generated datasets consisting of 64 positive strings and 64 negative strings, each with 64 bits, turned out to be impractical using this direct setting. Note that this corresponds to solving an optimization problem with 520 complex variables[2] over the product manifold $U(2)^{\times 66}$.

To circumvent this difficulty, we perform two crucial improvements. First, instead of applying the optimization directly into the whole product manifold $U(k)^{l \cdot s + c}$ we combine a local optimization paradigm together with sliding window protocol to optimize the problem in small parts. Second, we use pre-computation to reduce significantly the time required to evaluate the function to be optimized at each step. We will describe both steps below.

3.1. Local Optimization.

As mentioned above, we address the global optimization problem of finding a branching program that minimizes the objective function $\mathsf{CD}(B, D)$ into a succession of local optimization problems over manifolds of much smaller dimension. This process is split into two parts, an *instruction optimization* part and a *class representative optimization*. We describe these two parts in more detail below.

Window of Instructions Optimization. In the first part, given a fixed window size parameter w, an initial position i, and an initial branching program B, we construct a branching program B' by keeping all matrices fixed, except for the matrices

$$A_0^i, \ldots, A_{s-1}^i, \ldots, A_0^{i+w-1}, \ldots, A_s^{i+w-1}$$

belonging to instructions $I_i, \ldots, I_{i+w-1}$. In this sense, at each step, we solve an unconstrained optimization problem with $k^2 \cdot s \cdot w$ variables over the manifold $U(k)^{s \cdot w}$. Note that here, the window size is supposed to be small, and can even be set to 1, meaning that a single instruction (containing s unitary matrices) is optimized at a time.

Class Representative Optimization. In the second part, given an initial branching program B, we construct a branching program B' by keeping all matrices fixed, except for the matrices

$$\zeta = (X_0, \ldots, X_{c-1})$$

which define the class representatives of the branching program. The intuition is that fixing a sequence of class representatives a priory may affect the ability of a branching program learning a dataset. Since this effect is very hard to predict, the best option is to also leave the task of learning the class representatives to the learning algorithm.

[1] For each $i \in \{1, \ldots, l\}$, $(A_0^i, \ldots, A_{s-1}^i)$ are the matrices corresponding to the i-th instruction of the branching program.

[2] In this case, $k = 2$, $l = 64$, $s = 2$ and $c = 2$. The corresponding optimization problem to be solved has $k^2 \cdot (l \cdot s + c) = 520$ variables.

Combining Both Optimization Procedures. In essence, the algorithm works as follows. We start by sampling all matrices of the branching program, including the instruction matrices, and class representative matrices at random. Subsequently, we make a round of instruction optimizations. This round consists of $l - w + 1$ steps, where at step i, then we optimize the matrices corresponding to instructions $I_i, \ldots, I_{i+w-1}$ as described above. When this round is finished, we perform the class representative optimization. We iterate this process until the target accuracy has been achieved, or until time is up. The problem with applying this approach without further modification is at each call of the objective function $\mathsf{CD}(B, D)$, one needs to evaluate the branching program on each string of the dataset. Each such evaluation takes time $O(k^3 \cdot n \cdot m)$ where m is the number of strings in the dataset, n is the length of each string, and k is the dimension of the unitary matrices in the branching program.

3.2. Pre-computation

We can significantly reduce the time spent per call of the continuous distance function by using pre-computation.

Given a dataset $D = (D_0, \ldots, D_{c-1})$. A pre-computation for D is a tuple $(f_0, \ldots, f_c)$ of functions, where for each $a \in [\![c]\!]$, f_a is a function from $[|D_a|]$ to $U(k)$.

Given a dataset $D = (D_0, \ldots, D_{c-1})$ a k-dimensional unitary branching program B with index sequence $\mathcal{J} = (j_1, \ldots, j_\ell)$, and a non-negative integer $p \in \{0, \ldots, l\}$ we let

$$\mathrm{Left}_D(B, p) = (f_0, \ldots, f_{c-1})$$

where for each $a \in [\![c]\!]$, $f_a : [|D_a|] \to U(k)$ is the function defined as follows for each $z \in [|D_a|]$,

$$f_a(z) = \begin{cases} \mathbf{J}_k & \text{if } p = 0, \\ I_1[D_a[z][j_1]] \cdot \ldots \cdot I_p[D_a[z][j_p]] & \text{otherwise.} \end{cases}$$

In other words, $f_a(z)$ is the unitary in $U(k)$ obtained by applying instructions $I_1, \ldots, I_p$ to the z-string of D_a. In case $p = 0$, then $f_a(z)$ is the identity matrix $\mathbf{J}_k$, which intuitively corresponds to applying no instruction at all.

Analogously, given a dataset D and a branching program B as above and a non-negative integer $p \in \{1, \ldots, l + 1\}$, we define

$$\mathrm{Right}_D(B, p) = (g_0, \ldots, g_{c-1})$$

where for each $a \in [\![c]\!]$, $g_a : [|D_a|] \to U(k)$ is the function defined as follows for each string $W \in D_a$,

$$g_a(z) = \begin{cases} \mathbf{J}_k & \text{if } p = l + 1, \\ I_p[D_a[z][j_p]] \cdot \ldots \cdot I_l[D_a[z][j_l]] & \text{otherwise.} \end{cases}$$

In other words, $g_a(z)$ is the unitary in $U(k)$ obtained by applying instructions $I_p, \ldots, I_l$ to the z-th string of D_a. In case $p = l + 1$, then $g_a(z)$ is the identity matrix $\mathbf{J}_k$, which intuitively corresponds to applying no instruction at all.

Finally, given a dataset D and a branching program B as above and non-negative integers $p, p' \in \{1, \ldots, l\}$ with $p < p'$, we define

$$\mathrm{Middle}_D(B, p, p') = (h_0, \ldots, h_{c-1})$$

where for each $a \in [\![c]\!]$, $h_a : [|D_a|] \to U(k)$ is the function defined as follows for each $z \in [|D_a|]$.

$$h_a(W) = I_p[D_a[z][j_p]] \cdot \ldots \cdot I_{p'}[D_a[z][j_{p'}]].$$

In other words, $h_a(z)$ is the unitary in $U(k)$ obtained by applying instructions $I_p, \ldots, I_{p'}$ to the z-th string in D_a.

With the notation above, we have the following observation.

Observation 1. *Let $D = (D_0, \ldots, D_c)$ be a dataset, B be a branching program of length l, and $p < p'$ be positions in $[l]$. Let*

- $\mathrm{Left}_D(B, p - 1) = (f_0, \ldots, f_{c-1})$,
- $\mathrm{Middle}_D(B, p, p') = (h_0, \ldots, h_{c-1})$, *and*
- $\mathrm{Right}_D(B, p' + 1) = (g_0, \ldots, g_{c-1})$.

Then, for each $a \in [\![c]\!]$ and each $z \in [|D_a|]$, we have that $value(B, D_a[z]) = f_a(z) \cdot h_a(z) \cdot g_a(z)$.

Let $D = (D_0, \ldots, D_c)$ be a dataset, and B be a k-dimensional branching program of length l. Let $L = (f_0, \ldots, f_c)$ and $R = (g_0, \ldots, g_c)$ be pre-computations for D, and p and w be positive integers. Let $\mathrm{FD}(B, D, L, R, p, w)$ be the real number obtained from the following sum:

$$\sum_{a \in [\![c]\!]} \sum_{z \in [|D_a|]} \frac{\mathsf{MD}(f_a(z) \cdot h_a(z) \cdot g_a(z), X_a)}{|D_a|}$$

where $(h_0, \ldots, h_a) = \mathrm{Middle}_D(B, p, p + w - 1)$.

Then, from Observation 1, we have the following proposition stating that $\mathrm{FD}(B, D, L, R, p, w)$ can be used to compute the continuous distance between a branching program and a dataset.

Proposition 2. *Let D be a dataset, B be a branching program of length l, and p and w be positive integers such that $p + w - 1 \leq l$. Let $L = \mathrm{Left}_D(B, p - 1)$ and $R = \mathrm{Right}_D(B, p + w)$ then $\mathrm{FD}(B, D, L, R, p, w) = \mathsf{CD}(B, D)$.*

The reason why we will be using $\mathrm{FD}(B, D, L, R, p, w)$ as a way of computing the continuous distance $\mathsf{CD}(B, D)$ between a branching program and a dataset is because the former is much more efficient than the later. More precisely, computing $\mathsf{CD}(B, D)$ takes time $O(k^3 \cdot n \cdot m)$, where m is the number of strings in the dataset, n is the length of each string, and k is the dimension of the unitary matrices in the branching program. On the other hand, computing this value using $\mathrm{FD}(B, D, L, R, p, w)$ takes time $O(k^3 \cdot w \cdot m)$, since k and w will be very small in our applications, this computation takes essentially linear time on the number of strings of the dataset, provided that the pre-computations L and R have already been performed. It turns out that in the process of optimizing the matrices belonging to instructions in a given window, the Riemannian gradient descent algorithm will need to call the function $\mathrm{FD}(B, D, L, R, p, w)$ many times until it stabilizes, while the pre-computations L and R will be kept fixed during the whole process. This represents a huge save of time, and allows us to deal with datasets containing strings with thousands of bits.

In Algorithm 1, we depict a pseudo-code of our algorithm. In the pseudo-code, for given positive integers k and r, we let $\mathrm{Sample}(U(k)^r)$ be the function that samples a tuple containing r $k \times k$ unitary matrices at random. We let $\mathrm{OptWindow}(B, D, L, R, p, w)$ be the function that applies the Riemannian gradient descent method to optimize the matrices corresponding to instructions $I_p, \ldots, I_{p+w-1}$. Internally, this function makes calls to the function $\mathrm{FD}(B, D, L, R, p, w)$, and the pre-computations L and R are obtained before calling this function. Finally, $\mathrm{OptClasses}(B, D, M)$ is the function that optimizes the class representatives. Intuitively, M is the pre-computation corresponding to applying all instructions of the branching program. Internally, this optimization function calls the pre-computed function $\hat{\mathrm{FD}}(B, D, M)$ which when given a branching program B, a dataset D and a pre-computation $M = (h_0, \ldots, h_{c-1})$, outputs the value

$$\sum_{a \in [\![c]\!]} \sum_{z \in [|D_a|]} \frac{\mathsf{MD}(h_a(z), X_a)}{|D_a|}.$$

This function can also be used to compute the continuous distance function between a branching program and a dataset, as stated in the following proposition.

Proposition 3. *Let D be a dataset and B be a branching program of length l. Let $M = \mathrm{Middle}_D(B, 1, l)$ then $\hat{\mathrm{FD}}(B, D, M) = \mathsf{CD}(B, D)$.*

4. A Near Quadratic Size Lower Bound for Unitary Branching Programs

The task of proving superlinear lower bounds on the size of Boolean circuits for explicit families of functions (i.e functions in NP), is still one of the major unsolved open problems in computational complexity theory (Find et al., 2015; Iwama & Morizumi, 2002; Lachish & Raz, 2001). This task is even more tantalizing when considering non-Boolean models of computation. Therefore, developing techniques to prove super-linear lower bounds distinct models of computation has been an active line of research (Nečiporuk, 1966; Turán & Vatan, 1997; Roychowdhury & Vatan, 2001). In particular, near-quadratic lower bounds have been obtained for Boolean formulas (Nečiporuk, 1966), quantum formulas (Roychowdhury & Vatan, 2001), and more generaly for circuits of bounded treewidth (de Oliveira Oliveira, 2016). These lower bounds rely on a framework called the Nečiporuk's method. In this section, we combine Nečiporuk's method with techniques from algebraic geometry to prove a near-quadratic lower bound on the number of instructions in a unitary branching programs computing the element distictness function (Theorem 4). We note that our lower bound assumes no restriction at all on the number of bits necessary to represent the entries of the unitary matrices belonging to the branching program.

Algorithm 1: Sliding Window Unitary Learning.

Data: An n-input dataset D of arity s and class-size c; an index sequence $\mathcal{J}$, a dimension parameter k, a window-size parameter w; an accuracy parameter α; a maximum number of iterations $\mathbf{Max}$

Result: A branching program B with $\mathsf{DD}(B, D) \leq \alpha$ or the branching program obtained after executing iteration $\mathbf{Max}$.

$\mathcal{I} \leftarrow \mathrm{Sample}(U(k)^{l \cdot s})$.
$\zeta \leftarrow \mathrm{Sample}(U(k)^c)$.
$B \leftarrow (\mathcal{J}, \mathcal{I}, C_\zeta)$.
$t \leftarrow 0$; *Total number of iterations.*
$p \leftarrow 1$; *Initial position of the window.*
$\mathrm{MinDiscDist} \leftarrow \mathsf{DD}(B, D)$; *smallest discrete distance obtained so far.*
while $\mathrm{MinDiscDist} > \alpha$ *and* $t < \mathbf{Max}$ **do**
  $L \leftarrow \mathrm{Left}_D(B, p-1)$
  $R \leftarrow \mathrm{Right}_D(B, p+w)$
  $B \leftarrow \mathrm{OptWindow}(B, L, R, p, w)$
  $M \rightarrow \mathrm{Middle}_D(B, 1, l)$
  $B \leftarrow \mathrm{OptClasses}(B, M)$
  if $\mathsf{DD}(B, D) < \mathrm{MinDiscDist}$ **then**
    $\mathrm{MinDiscDist} \leftarrow \mathsf{DD}(B, D)$
  end
  if $p + w \leq l$ **then**
    $p \leftarrow p + 1$;
  else
    $p \leftarrow 1$;
  end
end

Let X be a set of variables. A *Boolean assignment* for X is a function of the form $\alpha : X \to \{0,1\}$. We denote by $\{0,1\}^X$ the set of all Boolean assignments for X. A Boolean function over X is a function $f : \{0,1\}^X \to \{0,1\}$.

Let $X = \{x_1, ..., x_n\}$ be a set of $n = 2q \log q$ distinct variables partitioned into q blocks $Y_1, Y_2, ..., Y_q$, where each block Y_i has $2 \log q$ variables. The *element distinctness* function $\delta_n : \{0,1\}^X \to \{0,1\}$ is defined as follows for each assignment $s_1, s_2, ..., s_q$ of the blocks $Y_1, Y_2, ..., Y_q$ respectively.

$$\delta_n(s_1, s_2, ..., s_q) = \begin{cases} 1 & \text{if } s_i \neq s_j \text{ for } i \neq j, \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

Theorem 4. *Let X be a set with n Boolean variables, and let $\delta_n : \{0,1\}^X \to \{0,1\}$ be the n-bit element distinctness function. Let B be a unitary branching program of dimension k and length l computing δ_n. Then $l = \Omega(\frac{n^2}{k^2 \log n})$.*

5. Gapped Read-Once Unitary Branching Programs

Let $B = (\mathcal{J}, \mathcal{I}, C)$ be a unitary branching program with class sequence $\zeta = (X_0, \ldots, X_{c-1})$. We say that B is *read-once* if the indices in $\mathcal{J}$ are pairwise distinct. In other words, each symbol of the input string is read at most once. Let δ be a positive real number between 0 and 1. We say that B is *δ-gapped* if for each i, j in $\{0, \ldots, c-1\}$, $|\mathsf{MD}(X_i, value(B, W)) - \mathsf{MD}(X_j, value(B, W))| > \delta$ for each $W \in [\![c]\!]^n$.

An n-input 2-*classes dataset* is a dataset $D = (D_0, D_1)$ with two classes $D_0, D_1 \subseteq [\![s]\!]^n$ for some s. The next theorem states that for each fixed $k \in \mathbb{N}_+$, any δ-gapped read-once unitary branching program consistent with a given n-input 2-classes dataset D can be transformed into a deterministic finite automaton over the alphabet $[\![s]\!]$ consistent with D whose number of states is polynomial in n and in $1/\delta$.

Theorem 5. *Let B be a gapped, read-once unitary branching program of dimension k separating an n-input 2-classes dataset $D = (D_0, D_1)$. One can construct a deterministic finite automaton $\mathcal{F}(B)$ over $[\![s]\!]$ with $\left(\frac{n}{\delta}\right)^{O(k^2)}$ states such that $D_1 \subseteq \mathcal{L}(\mathcal{F}(B))$ and $D_0 \cap \mathcal{L}(\mathcal{F}(B)) = \emptyset$.*

In the classic framework of learning with membership and equivalence queries, the goal is to learn a regular language L over a given alphabet $[\![s]\!]$ in the setting where the learner has access to an oracle (a minimally adequate teacher) that is able to answer two types of queries: *membership queries*, where the learner selects a word in $W \in [\![s]\!]^*$ and the teacher answers whether or not $W \in L$; and *equivalence queries* where the learner selects an hypothesis automaton $\mathcal{H}$ and the teacher answers whether or not L is the language of $\mathcal{H}$. If this is not the case, the teacher gives to the student a counter-example word $W \in \Sigma^* \backslash L$. A celebrated result from Angluin (Angluin, 1987) states that any regular language can be exactly learned with a number of queries that is polynomial in the number of states of a minimum deterministic automaton representing the target language, and the size of the largest counter-example returned by the teacher. If L is the characteristic language of a function $f : [\![s]\!]^n \to \{0,1\}$, meaning that $L(f) = \{W \in [\![s]\!]^n \ : \ f(W) = 1\}$, then Angluin's result implies that one can learn an automaton accepting L in time polynomial in n and in the size of the minimum deterministic finite automaton accepting L. Theorem 5 can be used to transfer this classic result from the context of functions computable by DFAs to functions computable by gapped read-once unitary branching programs.

Theorem 6. *For any fixed k and any $\delta \in \in \mathbb{R}$ with $0 < \delta < 1$, the representation class of n-input δ-gapped read-once unitary branching programs is exactly learnable using the representation class of deterministic finite automata with a total number of queries that is polynomial in n and in $1/\delta$.*

Another interesting consequence of Theorem 5 is that it allows one to use techniques from best-partition communication complexity theory to prove explicit lower bounds for the dimension necessary to compute certain functions using gapped read-once branching programs. In the setting of best-partition communication complexity theory, two players, Alice and Bob wish to compute the value of a function $f : \{0,1\}^n \to \{0,1\}$, known in advance by both players, on a specific input W. The caveat is that the positions in $\{1, ..., n\}$ are partitioned into two sets A and B in such a way that Alice only has access to bits of W at positions in A and Bob only has access to bits of W at positions in B. In order to compute the value $f(W)$, Alice and Bob exchange bits using a communication protocol, where in the end of the process, the last bit of the protocol is the value $f(W)$. Both the protocol and the partition of the input bits are agreed by Alice and Bob before the bits of the input string are distributed to them. And the protocol should give the correct answer on every input string. The deterministic communication complexity of f is the minimum number of bits in a deterministic protocol computing f, while the non-deterministic complexity of f is the minimum number of bits in a non-deterministic protocol.

Lemma 7. *Let $f : 0,1^n \to \{0,1\}$ be a function such that $L(f)$ is the language of some non-deterministic finite automaton with m states. Then the non-deterministic communication complexity of f is $O(\log m)$.*

Therefore, by combining Lemma 7 with Theorem 5, we have the following.

Lemma 8. *Let $f : \{0,1\}^n \to \{0,1\}$ be a function computable by a δ-gapped read-once unitary branching pro-*

gram of dimension k. Then the communication complexity of f is $O(k^2 \cdot \log n/\delta)$.

An example of function with best-partition non-deterministic communication complexity $\Omega(n)$ is the *triangle-freeness* function $\Delta_n : \{0,1\}^n \to \{0,1\}$ (Jukna & Schnitger, 2002). This function takes as input an array $x = (x_{ij})_{1 \le i < j \le m}$ consisting of $n = \binom{m}{2}$ Boolean variables representing an undirected graph $G(x)$ on m vertices $\{1, ..., m\}$. The graph $G(x)$ has an edge connecting vertices i and j, with $i < j$, if and only if $x_{ij} = 1$. The triangle-freeness function Δ_n returns 1 on an input x if and only if the graph $G(x)$ does not contain a triangle. From Lemma 8 and the $\Omega(n)$ lower bound on the best-partition non-deterministic communication complexity of Δ_n, we have the following theorem.

Theorem 9. *Let B be a δ-gapped read-once branching program computing $\Delta_n : \{0,1\}^n \to \{0,1\}$. Then $k = \Omega(\sqrt{n}/(\log n/\delta))$.*

6. Experimental Evaluation

In this section, we describe experiments designed to evaluate the performance of our algorithm in the task of learning unitary branching programs consistent with a given dataset. For the purpose of these experiments, we focus on read-once branching programs, where the bits of the input are read from left to right. More formally, for datasets containing strings with n bits, the sequence of indices of the branching program is $\mathcal{J} = (1, 2, \ldots, n)$.

We have implemented our algorithm in a tool called *LUBP - Learning Unitary Branching Programs*, which can be downloaded at https://github.com/AutoProving/LUBP. Our tool was implemented in C++ using ROPTLIB, a library for optimization on Riemannian manifolds (Huang et al., 2018). Each instance was executed in a machine with CPU of type *Intel Xeon-Gold 6138 2.0 GHz*. We set the maximum amount of RAM memory to 4GB per tested instance. Our implementation is sequential, in the sense that it does not resort to the use of parallel programming techniques.

In our experiments we evaluated the ability of our algorithm to learn branching programs consistent with randomly generated sparse datasets. More precisely, for each n in the set $\{16, 32, 64, 128, 256, 512, 1024\}$, we generated 10 datasets at random, each containing two classes, and each class containing n strings from $\{0,1\}^n$. We call these datasets *n-datasets*. Each such n-dataset was generated in two stages. In the first stage, we sampled, uniformly at random, one n-bit string at a time and added it to a set S_0 until it had n strings. Subsequently, we generated one n-bit string, uniformly at random, at a time and added it to a set S_1, provided it was not already present in the first set S_0, until it had n strings.

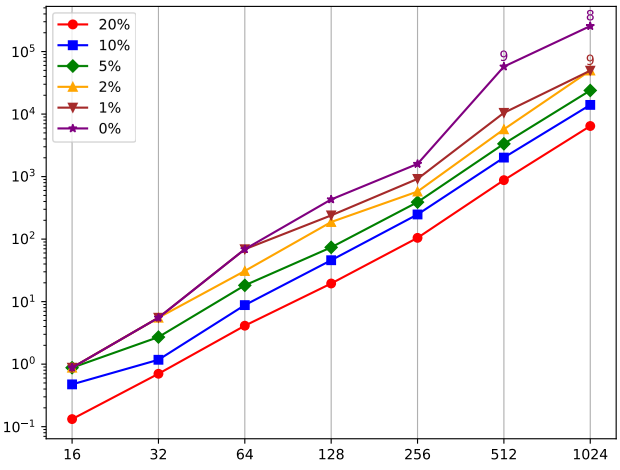

Figure 1. Average time in seconds needed to learn a dimension-3 unitary branching program consistent with an n-dataset up to a given accuracy. For instance, the red line depicts the time necessary to learn such a branching program with at most 20% error. The considered values of n are represented in the horizontal axis, while the time in seconds is depicted in the vertical axis. A number r close to point (n, t) of a given color indicates that the accuracy represented by the color was achieved only for r out of the 10 randomly generated instances of n-datasets. In this case, the average is taken with respect to these r instances. No number close to such a point (n, t) indicates that the corresponding accuracy was obtained on all 10 n-datasets. For instance, for all but two of the 1024-datasets our program was able to learn a completely consistent (0% error) dimension-3 unitary branching program.

We run our algorithm on each dataset and recorded for each of these datasets the number of seconds necessary to achieve an error of at most x, for x belonging to the set $\{20\%, 10\%, 5\%, 2\%, 1\%, 0\%\}$. We note that for the vast majority of the generated datasets (except for one instance of a 512-dataset and two instances of a 1024-dataset), our algorithm was able to learn a completely consistent unitary branching program of dimension-3 (0% error). In Figure 1 we depict a graph where the horizontal axis corresponds to the length of strings in a given dataset, and the vertical axis corresponds to the time necessary to learn the dataset with a given accuracy. For instance, the red line in Figure 1 depicts the time necessary to learn a dataset with at most 20% error in function of n, while the purple line represents the time to learn the datasets with 0% error.

7. Conclusion

In this work, we have studied the notion of unitary branching program (UBP) a model of computation that lifts the classical notion of programs over monoids to the continuous setting. The main insight is that such UBPs may be viewed as points in a complex Stiefel manifold, and therefore, the

process of learning a UBP consistent with a given dataset can be approached using techniques based on Riemannian gradient descent.

We have proved explicit near-quadratic lower bounds on the length of unitary branching programs of bounded dimension. This result imposes no restriction on the number of times each symbol of the input is read and on the number of bits necessary to represent the complex numbers in the matrices defining the UBPs. On the other hand, we proved an explicit polynomial lower bound on the dimension of read-once polynomially-gapped UBPs. We leave open the question of proving similar (or stronger) lower bounds for general read-once UBPs (i.e., without imposing a restriction on the gap).

Our empirical results indicate that read-once UBPs of small dimension are able to represent n-datasets with 0% error. It would be very interesting to have an analytic proof or disproof for this empirical observation. More specifically, is it the case that read-once UBPs of dimension 3 can represent arbitrary n-datasets? In the opposite direction, can one construct a family of n-datasets which cannot be represented by read-once UBPs of constant dimension?

Acknowledgements

This work was supported by the Research Council of Norway in the context of the project Automated Theorem Proving from the Mindset of Parameterized Complexity Theory (Grant no 288761). We also acknowledge support from the Sigma2 Network (Proj. no. NN9535K).

References

Abrudan, T., Eriksson, J., and Koivunen, V. Conjugate gradient algorithm for optimization under unitary matrix constraint. *Signal Processing*, 89(9):1704–1714, 2009.

Abrudan, T. E., Eriksson, J., and Koivunen, V. Steepest descent algorithms for optimization under unitary matrix constraint. *IEEE Transactions on Signal Processing*, 56 (3):1134–1147, 2008.

Absil, P.-A., Mahony, R., and Sepulchre, R. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2009.

Adams, J. F. and Walker, G. On complex stiefel manifolds. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 61, pp. 81–103. Cambridge University Press, 1965.

Angluin, D. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

Barrington, D. A. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

Beame, P. W., Cook, S. A., and Hoover, H. J. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15(4):994–1003, 1986.

Bédard, F., Lemieux, F., and McKenzie, P. Extensions to barrington's m-program model. *Theoretical Computer Science*, 107(1):31–61, 1993.

Bergadano, F., Bshouty, N. H., Tamon, C., and Varricchio, S. On learning branching programs and small depth circuits. In *European Conference on Computational Learning Theory*, pp. 150–161. Springer, 1997.

Boumal, N., Mishra, B., Absil, P.-A., and Sepulchre, R. Manopt, a matlab toolbox for optimization on manifolds. *The Journal of Machine Learning Research*, 15(1):1455–1459, 2014.

Bshouty, N. H., Tamon, C., and Wilson, D. K. On learning width two branching programs. *Information Processing Letters*, 65(4):217–222, 1998.

Caussinus, H. A note on a theorem of barrington, straubing and thérien. *Information Processing Letters*, 58(1):31–33, 1996.

Chiu, A., Davida, G., and Litow, B. Division in logspace-uniform nc. *RAIRO-Theoretical Informatics and Applications*, 35(3):259–275, 2001.

Cleve, R. Towards optimal simulations of formulas by bounded-width programs. *Computational Complexity*, 1 (1):91–105, 1991.

Crow, M. L. and Ilic, M. The waveform relaxation method for systems of differential/algebraic equations. In *29th IEEE Conference on Decision and Control*, pp. 453–458. IEEE, 1990.

de Oliveira Oliveira, M. Size-Treewidth Tradeoffs for Circuits Computing the Element Distinctness Function. In *Proc. of the 33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016)*, volume 47 of *LIPIcs*, pp. 56:1–56:14, 2016.

Elberfeld, M., Jakoby, A., and Tantau, T. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In *29th International Symposium on Theoretical Aspects of Computer Science*, pp. 66, 2012.

Ergün, F., Kumar, S. R., and Rubinfeld, R. On learning bounded-width branching programs. In *Proceedings of the eighth annual conference on Computational learning theory*, pp. 361–368, 1995.

Find, M. G., Golovnev, A., Hirsch, E. A., and Kulikov, A. S. A better-than-3n lower bound for the circuit complexity of an explicit function. *Electronic Colloquium on Computational Complexity (ECCC)*, 22:166, 2015.

Huang, W., Absil, P.-A., Gallivan, K. A., and Hand, P. Roptlib: an object-oriented c++ library for optimization on riemannian manifolds. *ACM Transactions on Mathematical Software (TOMS)*, 44(4):1–21, 2018.

Iwama, K. and Morizumi, H. An explicit lower bound of 5n-o (n) for boolean circuits. In *Mathematical foundations of computer science 2002*, pp. 353–364. Springer, 2002.

Janssen, J. and Vandewalle, S. On sor waveform relaxation methods. *SIAM journal on numerical analysis*, 34(6): 2456–2481, 1997.

Jukna, S. and Schnitger, G. Triangle-freeness is hard to detect. *Combinatorics Probability and Computing*, 11(6): 549–570, 2002.

Lachish, O. and Raz, R. Explicit lower bound of 4.5 n-o (n) for boolena circuits. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pp. 399–408. ACM, 2001.

Lelarasmee, E., Ruehli, A. E., and Sangiovanni-Vincentelli, A. L. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *IEEE transactions on computer-aided design of integrated circuits and systems*, 1(3):131–145, 1982.

Mansour, Y. and McAllester, D. Boosting using branching programs. *Journal of Computer and System Sciences*, 64 (1):103–112, 2002.

Nakanishi, M., Hamaguchi, K., and Kashiwabara, T. Ordered quantum branching programs are more powerful than ordered probabilistic branching programs under a bounded-width restriction. In *International Computing and Combinatorics Conference*, pp. 467–476. Springer, 2000.

Nečiporuk. On a Boolean function. *Soviet Math. Dokl.*, 7 (4):999–1000, 1966.

Raghavan, V. and Wilkins, D. Learning μ-branching programs with queries. In *Proceedings of the Sixth Annual ACM Conference on Computational Learning Theory*, pp. 27–36. ACM Press, New York, NY, 1993.

Roychowdhury, V. P. and Vatan, F. Quantum formulas: A lower bound and simulation. *SIAM Journal on Computing*, 31(2):460–476, 2001.

Turán, G. and Vatan, F. On the computation of boolean functions by analog circuits of bounded fan-in. *Journal of Computer and System Sciences*, 1(54):199–212, 1997.

Viola, E. *On the power of small-depth computation*. Now Publishers Inc, 2009.