Memoryless computation: new results, constructions, and extensions

Maximilien Gadouleau[*] Søren Riis[†]

August 21, 2014

Abstract

In this paper, we are interested in memoryless computation, a modern paradigm to compute functions which generalises the famous XOR swap algorithm to exchange the contents of two variables without using a buffer. In memoryless computation, programs are only allowed to update one variable at a time. We first consider programs which do not use any memory. We study the maximum and average number of updates required to compute functions without memory. We then derive the exact number of instructions required to compute any manipulation of variables. This shows that combining variables, instead of simply moving them around, not only allows for memoryless programs, but also yields shorter programs. Second, we show that allowing programs to use memory is also incorporated in the memoryless computation framework. We then quantify the gains obtained by using memory: this leads to shorter programs and allows us to use only binary instructions, which is not sufficient in general when no memory is used.

1 Introduction

How do you swap the contents of two Boolean variables x and y by updating one variable at a time? The common approach is to use a buffer t, and to do as follows (using pseudo-code).

$$t \leftarrow x$$
$$x \leftarrow y$$
$$y \leftarrow t.$$

However, a famous programmer's trick consists in using XOR, which can be viewed as addition over a binary vector space:

$$x \leftarrow x + y$$
$$y \leftarrow x + y$$
$$x \leftarrow x + y.$$

The swap can thus be performed without any buffer. The aim is to generalise this idea to compute any possible function without additional memory.

Memoryless computation (MC)–referred to as closed iterative calculus in [4] and *in situ* programs or computation with no memory in [7]–is a modern paradigm for computing functions, which offers two main innovations. The first introduces a completely different view on how to compute functions. The basic example is the XOR swap described above. Unlike traditional computing, which views the registers as "black boxes," MC takes advantage of the nature of the information contained in those registers and combines the values of the different registers. Thus, it can be seen as the computing analogue of Network Coding, a revolutionary technique to transmit data through a network which lets the intermediate nodes combine the messages they receive [24]. In particular, the XOR swap is the analogue of the canonical example of Network Coding, the so-called butterfly network [1].

[*]School of Engineering and Computing Sciences, Durham University, Durham, UK. Email: `m.r.gadouleau@durham.ac.uk`

[†]School of Electronic Engineering and Computer Science, Queen Mary, University of London, Mile End Road, London E1 4NS, UK. Email: `smriis@eecs.qmul.ac.uk`

The second main innovation lies in the computational model used for MC, which can be briefly described as follows. A processing unit has n registers $x_1, \dots, x_n$ containing data over a finite alphabet A and has to compute a function $f : A^n \to A^n$ which possibly modifies the values of all registers. It is allowed any updates which only modify one register at a time (i.e., $x_i \leftarrow g(x_1, \dots, x_n)$ for some $g : A^n \to A$), which are called *instructions*. A sequence of instructions computing a given function is a *program* for that function. Because an instruction is viewed as a quantum of complexity (akin to a clock cycle), the (memoryless) *complexity* of a function is defined as the minimum length of a program computing that function. For instance, the complexity of the swap of two bits is equal to three instructions.

Although MC is still mainly treated from a theoretical point of view, it has a wide range of possible long term applications, especially for computationally expensive problems. It can already be show to offer several advantages over traditional computing. First, MC offers a computational speed-up at the core level. Indeed, MC yields arbitrarily shorter programs than traditional computing when manipulating variables (see Corollary 2 for more details). Secondly, MC does not rely on additional buffers and hence performs computations in line. Memory management is a tedious task which can significantly slow down computations [20] by bringing a significant overhead. This problem is particularly important for parallel architectures with shared memory [20]. Instead, MC uses no data memory and thus eases concurrent execution of different tasks by preventing memory conflicts.

While the XOR swap described above is folklore, MC was developed by Burckel et. al. in [4, 5, 8, 9, 10, 6] and more extensively in [7]. It is notably proved that any function can be computed without memory. Moreover, only $2n - 1$ instructions are needed to compute any bijective function $f : A^n \to A^n$; any function $f : A^n \to A^n$ can be computed in only $4n - 3$ instructions. A survey of these results and a striking relation between MC and switching networks [21] and provided in [7].

We would like to emphasize the novelty of the results of this paper and how they differ from those in the literature. Firstly, many aspects considered in this paper are completely novel. These include the study of the average memoryless complexity in Sections 3.1 and 3.3, the study of manipulations of variables in Section 4, the use of binary instructions in Theorem 6 and the use of additional registers in Section 5. Moreover, some of the results presented in this paper extend or generalise some of those given in the literature. For instance, by extending a key lemma in [7], we manage to extend the flexible approach to constructing programs of length $4n - 3$ introduced in [6] and [7, Section 5] for the Boolean case to any alphabet, thus answering part of Open problem 2 in [7]. Other results provide some matching upper and lower bounds which are absent in the literature, e.g. in Theorem 2. Finally, we also provide an alternative and much shorter proof to the seminal Theorem 1, which states that any function can be computed without memory.

The rest of the paper is organised as follows. Section 2 reviews the memoryless computation model and proves that it is universal: any function can be computed without memory. Section 3 then investigates the number of updates required to compute any function. Section 4 determines the complexity of manipulating variables without memory and shows that memoryless computation yields shorter programs than traditional methods. Section 5 finally proves that additional registers (or memory) can be added into the memoryless computation model without loss of generality.

2 Model for memoryless computation

2.1 Instructions and programs

We first review the model for memoryless computation introduced in [4] and subsequently developed in [5, 8, 9, 10, 6, 7] and surveyed in [7].

Let A be a finite set, referred to as the *alphabet*, of cardinality q and let n be a positive integer (without loss, we shall usually regard A as $\mathbb{Z}_q$ or $\mathrm{GF}(q)$ when q is a prime power). We refer to any element of A^n as a *state*. We view any transformation f of A^n (i.e., any function $f : A^n \to A^n$) as a tuple of functions $f = (f_1, \dots, f_n)$, where $f_i : A^n \to A$ is referred to as the i-th coordinate function of f. In particular, a coordinate function is *trivial* if it is equal to the identity, i.e. $f_i(x) = x_i$; it is nontrivial otherwise. When considering a sequence of transformations, we shall use superscripts, e.g. $f^k : A^n \to A^n$ for all k–and hence f^k shall never mean taking f to the power k.

Definition 1 (Instruction)**.** An *instruction* is a transformation g of A^n with at most one non-trivial coordinate function g_i. If g is not the identity, we say that the instruction *updates* y_i for $y = (y_1, \dots, y_n) \in A^n$ and we denote it as

$$y_i \leftarrow g_i(y).$$

A permutation instruction is an instruction which maps A^n bijectively onto A^n (i.e. is a permutation of A^n).

By definition, the identity is an instruction, which can be represented by $y_i \leftarrow y_i$ for any $1 \leq i \leq n$.
We denote the set of instructions of A^n as $\bar{\mathcal{I}}(A^n)$ and the set of permutation instructions as $\mathcal{I}(A^n)$. We shall simply write $\bar{\mathcal{I}}$ and $\mathcal{I}$ when there is no ambiguity. For instance, if $A = \mathrm{GF}(2)$ and $n = 2$, then $\mathcal{I}$ is given by

$$\{(x_1, x_2), (x_1 + 1, x_2), (x_1 + x_2, x_2), (x_1 + x_2 + 1, x_2), (x_1, x_2 + 1), (x_1, x_1 + x_2), (x_1, x_1 + x_2 + 1)\}.$$

In update form, $\mathcal{I}$ can be written as follows:

$$\begin{array}{llll} \{y_1 \leftarrow y_1, & y_1 \leftarrow y_1 + 1, & y_1 \leftarrow y_1 + y_2, & y_1 \leftarrow y_1 + y_2 + 1, \\ y_2 \leftarrow y_2, & y_2 \leftarrow y_2 + 1, & y_2 \leftarrow y_1 + y_2, & y_2 \leftarrow y_1 + y_2 + 1\}, \end{array}$$

where the identity is represented by $y_1 \leftarrow y_1$ and $y_2 \leftarrow y_2$.

Definition 2 (Program)**.** For any transformation f of A^n, a *program* of length L computing f is a sequence of instructions $g^1, \dots, g^L$ such that

$$f = g^L \circ \ldots \circ g^1.$$

We shall write the instructions of a program in their update form one below the other. Although the identity is an instruction, any instruction in a program is not the identity unless specified otherwise. Also, since the set of instructions updating a given coordinate is closed under composition, without loss we can always assume that g^{k+1} updates a different coordinate than g^k for all k. The cases where $q = 1$ or $n = 1$ being trivial, we shall assume $q \geq 2$ and $n \geq 2$ henceforth.

We consider a processing unit which has access to a finite number n of registers and only allows programs of the form described above. We use $y = (y_1, \dots, y_n)$ to represent the state, i.e. content of the registers, during the program, $x = (x_1, \dots, x_n)$ to represent the input and $f(x) = (f_1(x), \dots, f_n(x))$ to represent the output. Hence $y = x$ before the first instruction, and $y = f(x)$ after the last instruction. Note that we will also use the shorthand notation $y_i \leftarrow h(x)$ to reflect how the content of the registers relates with the program input. In particular, note that the last update of y_i must be $y_i \leftarrow f_i(x)$.

In order to illustrate the notation, let us rewrite the program computing the swap of two variables, i.e. $f : A^2 \to A^2$ where $f(x_1, x_2) = (x_2, x_1)$. It is given as follows (all operations being done mod q):

$$\begin{array}{ll} y_1 \leftarrow y_1 + y_2 & (y_1 \leftarrow x_1 + x_2) \\ y_2 \leftarrow y_1 - y_2 & (y_2 \leftarrow x_1) \\ y_1 \leftarrow y_1 - y_2 & (y_1 \leftarrow x_2). \end{array}$$

Definition 3. Let B, C be two alphabets and $f, g : B \to C$. We say g *dominates* f if $g(x) = g(x') \Rightarrow f(x) = f(x')$ for all $x, x' \in B$. In other words, $f = h \circ g$ for some transformation h.

A program for f with instructions $g^1, \dots, g^L$ induces a sequence of transformations $h^1, \dots, h^L = f$ of A^n where h^1 is an instruction, h^i and h^{i+1} differ in only one coordinate, and h^i dominates h^{i+1} for all i. Indeed, simply let $h^{i+1} = g^{i+1} \circ h^i$; equivalently h^i represents the content of y after the i-th instruction of the program. In particular, if f is a permutation, then all intermediate transformations must be permutations as well.

We remark that this framework only allows to compute one function. However, it may be fair to ask the program to sequentially compute different functions. This can be incorporated in this framework if all the outputs are permutations. However, the case of general transformations is more troublesome: for instance, if we ask to return $f^1(x_1, x_2) = (x_1, x_1 + 1)$ and then $f^2(x_1, x_2) = (x_2, x_2 + 1)$, then it is clear that f^2 cannot be computed after f^1. In general, a program can sequentially compute $f^1, \dots, f^K$ only if f^i dominates f^{i+1} for all $1 \leq i \leq K-1$ (the results in this paper will show that this is necessary and sufficient). Therefore, this program can be broken down into K shorter programs, each computing one function. In view of these considerations, we shall only consider programs which compute one transformation f in the remaining of this paper.

2.2 All transformations are computable without memory

We are now interested in the general case of computing any transformation of n variables. We first give in Theorem 1 a new, more concise, proof of the seminal result of memoryless computation: any transformation can be computed without memory, a result from [4]. This proof notably unveils relations with combinatorics (via the Gray code) and above all algebra (via generating sets) which will be crucial in the development of further research in the subsequent papers in memoryless computation by Cameron, Fairbairn and Gadouleau [11, 12].

We introduce some useful notation for any states $u, v \in A^n$. First, the *transposition* of u and v, denoted as (u, v), is the permutation of A^n which maps u to v, v to u, and fixes any other state in A^n. Second, the *assignment* of u to v, denoted as $(u \to v)$, is the transformation which maps u to v and fixes any other state in A^n. Third, we denote the all-zero state as e^0 and the k-th unit state as $e^k \in A^n$, where $e^k_i = \delta(i, k)$ and δ is the Kronecker delta function. Therefore, if $v = u + e^i$ for some i, the transposition (u, v) is an instruction with update form

$$y_i \leftarrow y_i + \delta(y, u) - \delta(y, v),$$

Moreover, the assignment $(e^0 \to e^1)$ is an instruction with update form

$$y_1 \leftarrow y_1 + \delta(y, e^0).$$

Theorem 1 (see [4]). *Any transformation of A^n can be computed by a program which only consists of transpositions (u, v) where $v = u + e^i$ for some i and the assignment $(e^0 \to e^1)$.*

Proof. If we order the states of A^n according to the Gray code in [19], then any two consecutive states v^j and v^{j+1} satisfy $v^{j+1} = v^j \pm e^{i_j}$ for some i_j. The transpositions above are exactly the Coxeter generators $\{(v^j, v^{j+1}) : 1 \leq j \leq q^n - 1\}$ corresponding to this ordering. Therefore, any permutation of A^n can be computed using these instructions. Furthermore, adding any transformation with $q^n - 1$ images to a generating set of $\mathrm{Sym}(A^n)$ yields a generating set of the transformation monoid of A^n [17, Theorem 3.1.3]. Since the assignment $(e^0 \to e^1)$ is an instruction with $q^n - 1$ images, we obtain the result. $\square$

3 Memoryless complexity

Definition 4 (Memoryless complexity). The shortest length of a program computing a transformation f of A^n is referred to as the *memoryless complexity* of f and is denoted as $\mathcal{L}(f)$. By convention, the identity has memoryless complexity 0.

We have $\mathcal{L}(f \circ g) \leq \mathcal{L}(f) + \mathcal{L}(g)$ for any two transformations f and g. Furthermore, if f is a permutation, then it is easy to show that $\mathcal{L}(f^{-1}) = \mathcal{L}(f)$. We then obtain that

$$d(f, g) := \mathcal{L}(f \circ g^{-1})$$

defines a metric on the symmetric group of A^n, which is the word metric, with generators given by all the permutation instructions.

We believe that memoryless computation is an interesting model to evaluate the true complexity of computations operated on cores. Indeed, such computations mostly involve manipulations of registers. Also, the only accurate measure of complexity would be the time it takes for a processor to perform that computation. Because each instruction is counted equal, regardless its nature, the memoryless complexity model only takes the number of clock cycles it takes to compute a given function. Obviously, the model remains theoretical for it allows any possible update; the search for efficient instruction sets is work in progress.

An intriguing relation between memoryless computation and multistage interconnection networks [21] was discovered in [6] and developed in [7]. Remarkably, [7] shows that the $2n-1$ upper bound on the maximum complexity of permutation is equivalent to a result in [2], while the $4n-3$ upper bound on the complexity of any transformation is equivalent to a result in [23].

3.1 Memoryless complexity of permutations

In this section we first determine the maximum memoryless complexity of a permutation of A^n. It is remarkable that the permutation which maximises the memoryless complexity is very "simple" to describe (see Proposition 1); this highlights the difference between memoryless complexity and other complexity measures.

Theorem 2. *The maximum memoryless complexity of a permutation of A^n is $2n-1$ instructions.*

Proof. The fact that any permutation can be computed in $2n-1$ instructions was already given in [6]. The matching lower bound on the memoryless complexity is given in Proposition 1 below. □

Proposition 1. *The memoryless complexity of the transposition (a, b) of two states $a, b \in A^n$ is $2d-1$ instructions, where d is the Hamming distance between a and b: $d = |\{i : a_i \neq b_i\}|$.*

Proof. Without loss, let a and b disagree on their d first coordinates. Denoting

$$v^k = (b_1, \ldots, b_k, a_{k+1}, \ldots, a_n)$$

for $1 \leq k \leq d$, we obtain

$$(a, b) = (a, v^1) \circ \cdots \circ (v^{d-2}, v^{d-1}) \circ (v^{d-1}, b) \circ \cdots \circ (v^1, v^2) \circ (a, v^1).$$

Each transposition involves states differing in at most one position, and hence is an instruction. For instance, (a, v^1) is the instruction

$$y_1 \leftarrow y_1 + (b_1 - a_1)\left(\delta(y, a) - \delta(y, v^1)\right).$$

Therefore, the memoryless complexity is at most $2d-1$ instructions.

Conversely, suppose that there exists a program computing (a, b) with fewer than $2d-1$ instructions. In that program, at least two coordinates are only updated once (say i before j). Denote the images of a and b before the update of y_j as a' and b', respectively. Note that $a'_i = b_i$ and $b'_i = a_i$, since y_i will not be updated any further. The update of y_j is given by

$$y_j \leftarrow y_j + (b_j - a_j)(\delta(y, a') - \delta(y, b')),$$

since coordinate j cannot be modified for any program input other than a or b, and it must indeed give the correct values for these two inputs. However, this update is not bijective, for a' and b' differ in coordinate i. □

We can represent a program for any permutation of A^n as progressing around the Cayley graph of $\mathrm{Sym}(A^n)$ with generating set $\mathcal{I}$: $\mathrm{Cay}(\mathrm{Sym}(A^n), \mathcal{I})$ [18]. The set of permutation instructions $\mathcal{I} \subseteq \mathrm{Sym}(A^n)$ is described as follows. We remark that the set of permutation instructions updating a given coordinate forms a group, isomorphic to $\mathrm{Sym}(A)^{q^{n-1}}$.

Lemma 1. *For any alphabet A of cardinality q and any n, we have $|\bar{\mathcal{I}}| = nq^{q^n} - n + 1$ and*

$$|\mathcal{I}| = n(q!)^{q^{n-1}} - n + 1.$$

Proof. The number of instructions is easy: there are q^{q^n} choices for a function $A^n \to A$, and the identity is counted n times. For any $v \in A^{n-1}$ and any $1 \leq i \leq n$, let

$$A(v,i) = \{u \in A^n : (u_1, \dots, u_{i-1}, u_{i+1}, \dots, u_n) = v\}.$$

Let g be the instruction $y_i \leftarrow g_i(y)$. Then g is a permutation if and only if $g_i : A^n \to A$ satisfies $g_i(A(v,i)) = A$ for all $v \in A^{n-1}$. There are hence $q!$ choices for the reduction of g_i to each $A(v,i)$, and hence $(q!)^{q^{n-1}}$ choices for a permutation instruction updating y_i. We obtain $n(q!)^{q^{n-1}}$ possible instructions, where the identity has been counted n times, whence the result follows. □

We have determined the maximum memoryless complexity in Theorem 2. We are now interested in the average complexity. Proposition 2 gives a lower bound on that quantity.

Proposition 2. *The proportion of permutations of A^n with memoryless complexity at least*

$$\left\lfloor \frac{n \log q - 1}{q^{-1} \log q! + q^{-n} \log n} \right\rfloor + 1$$

tends to 1 *when n tends to infinity, where $q = |A|$.*

Proof. Any transformation with memoryless complexity l can be expressed as a program of l instructions. Therefore, the number of permutations with memoryless complexity at most l is no more than the number of l-tuples of permutation instructions, given by $|\mathcal{I}|^l$. By Lemma 1, we have

$$|\mathcal{I}| \leq n(q!)^{q^{n-1}} = \exp(\log n + q^{n-1} \log q!),$$
$$|\mathrm{Sym}(A^n)| = q^n! \geq \sqrt{2\pi q^n} q^{nq^n} \exp(-q^n) = \sqrt{2\pi q^n} \exp(q^n(n \log q - 1)).$$

Denoting $B = \frac{n \log q - 1}{q^{-1} \log q! + q^{-n} \log n}$ we obtain $|\mathrm{Sym}(A^n)| \geq \sqrt{2\pi q^n} |\mathcal{I}|^B$ and hence the proportion of permutations with memoryless complexity at most $\lfloor B \rfloor$ is upper bounded by $(2\pi q^n)^{-1/2}$. □

In particular, Proposition 2 shows that for n large, almost all permutations of $\mathrm{GF}(2)^n$ have computational complexity at least $2n - 2$. Therefore, they are very close to the maximum of $2n - 1$. However, the bound in Proposition 2 decreases with q.

3.2 Complexity and ordered functions

We now show how the problem of determining the memoryless complexity of a given permutation can be reduced to the case of so-called ordered permutations for nearly all permutations.

Definition 5 (Ordered function)**.** Let A and A^n be ordered (say, using the lexicographic order). For any function $f_i : A^n \to A$ and any $a \in A$, we denote the minimum element of $f_i^{-1}(a)$ as $m(a)$. We say f_i is ordered if $m(0) \leq m(1) \leq \ldots \leq m(q-1)$.

Any function $f_i : A^n \to A$ can be uniquely expressed as $f_i = \sigma_i \circ f_i^*$ where $\sigma_i \in \mathrm{Sym}(A)$ and f_i^* is ordered. In this case, we say that f_i is *parallel* to f_i^* [13].

By extension, we say that f is ordered if all its coordinate functions are ordered. Therefore, to any permutation f, we associate the ordered permutation f^* where $f_i = \sigma_i \circ f_i^*$ for some $\sigma_1, \dots, \sigma_n \in \mathrm{Sym}(A)$.

Proposition 3. *There exists a shortest program computing f^* using only ordered instructions. Furthermore, its length satisfies*

$$\mathcal{L}(f^*) \leq \mathcal{L}(f) \leq \mathcal{L}(f^*) + T(f),$$

where $T(f)$ is the number of nearly trivial (parallel to the trivial coordinate function) coordinate functions of f:

$$T(f) = |\{i : f_i^* = x_i, f_i \neq x_i\}|.$$

Proof. We first prove that there exists a shortest program computing f^* using only ordered instructions. Let $f^* = g^L \circ \dots g^1$ be a shortest program computing f^*. We can easily convert it to another program $h^L \circ \dots \circ h^1$ also computing f^* using only ordered instructions as follows. First let $h^1 = g^{1*}$. Then before g^j, we can express the content of the i-th cell as $y_i = \rho_i \circ y_i^*$ for all $1 \le i \le n$. Replace the instruction $y_i \leftarrow g_i^j(y)$ by

$$y_i \leftarrow h_i^j(y) = \tau g_i^j(\rho_1 \circ y_1, \dots, \rho_n \circ y_n),$$

where $\tau \in \mathrm{Sym}(A)$ guarantees that the instruction h^j is indeed ordered. It is easy to check that converting all instructions in this fashion does yield a program computing f^*.

We now prove that $\mathcal{L}(f^*) \le \mathcal{L}(f)$. Consider a shortest program $g^L \circ \dots \circ g^1$ computing f and convert it as follows to compute f^*. First, replace any final update $y_i \leftarrow f_i(x)$ by $y_i \leftarrow \sigma_i^{-1} \circ f_i(x) = f_i^*(x)$. Second, after this final update, replace any occurrence of y_i by $\sigma_i y_i$.

We finally prove that $\mathcal{L}(f) \le \mathcal{L}(f^*) + T(f)$. Consider a shortest program $h^L \circ \dots \circ h^1$ computing f^* (note that it may or may not update any of the coordinates y_i for which f_i is nearly trivial) and convert it as follows to compute f. First, replace any final update $y_i \leftarrow f_i^*(x)$ by $y_i \leftarrow \sigma_i \circ f_i^*(x) = f_i(x)$. Second, after this final update, replace any occurrence of y_i by $\sigma_i^{-1} y_i$. Third, update the eventual nearly trivial coordinate functions which have not been updated yet (there are at most $T(f)$ of them). □

3.3 Program computing linear transformations

We are now concerned with the case where q is a prime power and the inputs $x_1, \dots, x_n$ are elements of a finite field $A = \mathrm{GF}(q)$, and we want to compute a linear transformation f of A^n, i.e.

$$f(x) = xM^\top$$

for some matrix $M \in A^{n \times n}$. Each coordinate function f_i of f can be viewed as the inner product of a row of M with the input vector x. Therefore, we shall abuse notations slightly and refer to that row as f_i: $f_i(x) = f_i \cdot x$. In this section, we restrict ourselves to linear instructions only, i.e. instructions of the form

$$y_i \leftarrow a \cdot y = \sum_{j=1}^n a_j y_j,$$

for some $a = (a_1, \dots, a_n) \in A^n$.

Computing f is equivalent to calculating the matrix M as a product of matrices $M = M_1 \dots M_L$, where M_i is a matrix which only modifies one row. If M is nonsingular, this is also equivalent to a sequence of matrices $N_0 = I_n, N_1, \dots, N_{L-1}, N_L = M$ where N_i is nonsingular and N_i and N_{i+1} only differ by one row for all i.

Gaussian elimination indicates that any matrix can be computed by linear instructions involving only two rows. The number of such instructions required to compute any matrix is on the order of n^2. However, since we allow any linear instruction involving all n rows, we can obtain shorter programs. In [6, 7], it is proved that all matrices can be computed in $2n - 1$ linear instructions; in fact, their result holds not only for finite fields but for a much larger class of rings. In [12], the bound is lowered to $\lfloor 3n/2 \rfloor$ for matrices over finite fields only; Corollary 1 gives a matching upper bound.

Let us characterise the set $\mathcal{M}(\mathrm{GF}(q)^n)$ of invertible linear instructions. It is given by the set of nonsingular matrices with at most one nontrivial row: $\mathcal{M} = \{S(i, v) : 1 \le i \le n, v \in A^n(i)\}$, where

$$A^n(i) = \{v \in A^n, v_i \neq 0\} \text{ for all } 1 \le i \le n,$$

$$S(i, v) = \left(\begin{array}{c|c} I_{i-1} & 0 \\ \hline \multicolumn{2}{c}{v} \\ \hline 0 & I_{n-i} \end{array} \right) \in A^{n \times n}.$$

Remark that $S(i, v)^{-1} = S(i, -v_i^{-1} v)$ for all i, v and $|\mathcal{M}| = nq^{n-1}(q-1) - n + 1$. Computing a nonsingular matrix is hence equivalent to progressing around the Cayley graph $G := \mathrm{Cay}(\mathrm{GL}(n, q), \mathcal{M})$.

The following are equivalent:

1. M and N are adjacent in G.
2. $M = S(i,v)N$ and $N = S(i, -v_i^{-1}v)M$ for some i and $v \in A^n(i)$.
3. M and N only differ in one row.

Therefore, G is the subgraph of the Hamming graph $H(n, q^n)$ induced by $\mathrm{GL}(n,q)$.

When the field A is large, then almost all $n \times n$ matrices can be computed in no more than n linear instructions. The result below should be compared to the average memoryless complexity result of Proposition 2.

Proposition 4. *There are exactly*

$$(q-1)^n q^{n(n-1)} = \left(1 - \frac{1}{q}\right)^n q^{n^2}$$

$n \times n$ nonsingular matrices over GF(q) *which can be computed simply by updating their rows from* 1 *to n in increasing order.*

Proof. Let us count such matrices M with rows f_i. After the first instruction, we obtain the matrix whose first row is equal to f_1, while the last $n-1$ rows do not depend on the matrix we are computing and are equal to $(0|I_{n-1})$. Then f_1 can be chosen as any vector not in the span of the last $n-1$ rows: there are hence $(q-1)q^{n-1}$ choices for f_1. Once f_1 is fixed, similarly there are $(q-1)q^{n-1}$ choices for f_2, and so on. $\square$

Similar to the general case, we can reduce the problem of determining the complexity of nearly any nonsingular matrix to the case of so-called *scaled* matrices. Note that this concept is not necessarily consistent with the concept of ordered permutations; however, it can be viewed as an analogue. For any matrix M, we denote the minimum length of a linear program computing this matrix as $\mathcal{L}'(M)$.

Definition 6. A nonzero vector whose leading nonzero coefficient is equal to 1 is said to be *scaled.* A nonsingular matrix is *scaled* if all its rows are scaled.

For instance, the identity matrix is the only scaled diagonal matrix. For any nonzero vector $v \in \mathrm{GF}(q)^n$ with leading nonzero coordinate v_j, then $v^* := v_j^{-1}v^*$ is a scaled vector. For any nonsingular matrix M with rows f_i, let M^* be the corresponding scaled matrix with rows f_i^*. We obtain the linear analogue of Proposition 3.

Proposition 5. *There exists a shortest linear program computing M^* with only scaled instructions. Its length satisfies*

$$\mathcal{L}'(M^*) \leq \mathcal{L}'(M) \leq \mathcal{L}'(M^*) + T'(M),$$

where $T'(M)$ is the number of nearly trivial (equal to multiples of the corresponding unit vectors) rows of M:

$$T'(M) = |\{i : f_i = \mu_i e^i, \mu_i \in \mathrm{GF}(q)\backslash\{0,1\}\}| = |\{i : f_i \neq e^i, f_i^* = e^i\}|.$$

3.4 Memoryless complexity of all transformations

We have seen that any permutation of A^n can be computed in $2n-1$ memoryless instructions. Moreover, any transformation can be computed in $4n-3$ memoryless instructions. This a result first given in [6] for the Boolean case and then generalised to any alphabet in [7]. However, as noted in [7], this result can already be found in much earlier networks literature; [7] offers an explicit construction of the program computing any transformation in $4n-3$ instructions. In this section, we generalise a key lemma of [7] to answer part of Open Problem 2 in [7]; the rest of our proof follows exactly that of [6] for the Boolean case (also given in [7, Section 5]) and is only given to make the paper self-contained.

It is worth noting that although the $2n-1$ bound for permutations is tight, the $4n-3$ bound for general transformations is not: it is easy to check that for $q=2$ and $n=2$, any transformation of $\{0,1\}^2$ can be computed in at most three instructions.

Definition 7 (Lexicographic ordering)**.** For any $a = (a_1, \dots, a_n) \in A^n$ ($A = \mathbb{Z}_q$), the *lexicographic index* of a is the integer $\sum_{i=1}^{n} a_i q^{i-1}$. For the sake of conciseness, we shall abuse notation and identify a with its lexicographic index. An *interval* of A^n is any subset of the form

$$[b, c) := \{x \in A^n : b \leq x < c\}$$

for any $0 \leq b \leq c \leq q^n - 1$. For any $0 \leq i \leq n$ and $0 \leq j < q^{n-i}$, the *j-th block of level i* is the interval

$$B_{i,j} := [jq^i, (j+1)q^i).$$

We let λ be an integer partition of q^n, i.e. $\lambda : A^n \to \mathbb{Z}$, where $\lambda_a \geq 0$ for all $a \in A^n$ and $\sum_{a \in A^n} \lambda_a = q^n$.

Definition 8 (Extension of Definition 16 in [7])**.** We say λ is *proper* if for all $0 \leq i \leq n$ and all $0 \leq j < q^{n-i}$,

$$\sum_{a \in B_{i,j}} \lambda_a = 0 \mod q^i.$$

Lemma 2. *[Generalisation of [7, Lemma 17]] Any integer partition λ of q^n can be sorted properly, i.e. there exists $h \in \mathrm{Sym}(A^n)$ such that $\lambda \circ h$ is proper.*

Proof. We first prove the following claim. Any sequence of rq elements $a_0, \dots, a_{rq-1}$ of $\mathbb{Z}_q$ satisfying $a_0 + \ldots + a_{rq-1} = 0$ can be re-ordered such that

$$a_0 + \ldots + a_{q-1} = a_q + \ldots + a_{2q-1} = \ldots = a_{(r-1)q} + \ldots + a_{rq-1} = 0.$$

The proof easily follows from the following theorem due to Erdös, Ginzburg and Ziv [16]: any sequence $b_0, \dots, b_{2q-2}$ of $2q - 1$ elements of $\mathbb{Z}_q$ can be re-ordered such that $b_0 + \ldots + b_{q-1} = 0$.

We now build the ordering recursively (we shall follow the construction in [6]). Begin at level $i = 0$ with q^n blocks of size 1 having each value in the sequence $\lambda_0, \dots, \lambda_{q^n-1}$. At level $i + 1$, gather the elements of the sequence into groups of q elements, whose values sum up to a multiple of q (this is possible due to our claim), say kq. Define the value of this new block as k. This defines a new sequence of q^{n-i-1} non-negative integers whose sum is q^{n-i}. We finish at level n. $\square$

Example 1 (Illustration of Lemma 2). Let $q = n = 3$ and $\lambda = (5, 4, 4, 3, 3, 2, 2, 1, 1, 1, 1, 0, \dots, 0)$ with 16 zeros. We obtain the following, where the subscripts denote the value of the block in the next level:

$$\begin{aligned}
&[5,4,3]_4[4,3,2]_3[2,1,0]_1[1,1,1]_1[0,0,0]_0[0,0,0]_0[0,0,0]_0[0,0,0]_0[0,0,0]_0\\
&[4,1,1]_2[3,0,0]_1[0,0,0]_0\\
&[2,1,0]_1
\end{aligned}$$

Therefore, the corresponding proper partition is $(5, 4, 3, 2, 1, 0, 1, 1, 1, 4, 3, 2, 0, \dots, 0)$ with 15 zeros at the end.

Now that we have the key arithmetic property of Lemma 2, the rest of the proof follows [6]. It is given here in detail in order to make the paper complete and self-contained.

Definition 9. For any proper integer partition λ of q^n, Λ is the transformation of A^n such that

$$\Lambda\left(\left[\sum_{b=0}^{a-1} \lambda_b, \sum_{b=0}^{a} \lambda_b\right)\right) = a$$

for all $a \in A^n$.

Lemma 3. *[Extension of [7, Lemma 21]] For any proper λ, the transformation Λ satisfies the following property: if $a, b \in A^n$ agree on coordinates i to n for some i, then so do $\Lambda(a)$ and $\Lambda(b)$.*

Proof. First, we prove the following claim. For every i, j as above, there exist $0 \leq k, k' < q^{n-i}$ such that $\Lambda^{-1}(B_{i,j}) = \bigcup_{k \leq l \leq k'} B_{i,l}$.

Proof of claim: We remark that the pre-image by p of an interval is an interval itself. By definition of Λ, we have

$$|\Lambda^{-1}(B_{i,j})| = \sum_{a \in B_{i,j}} \lambda_a = 0 \mod q^i,$$

since λ is proper.

For a fixed i we prove the claim by induction on j. If $j = 0$ then $|\Lambda^{-1}(B_{i,0})| = kq^i$ for some k: it is either empty or is the interval $[0, kq^i) = \bigcup_{0 \leq l \leq k} B_{i,l}$.

If the property is true for all l with $0 \leq l < j$, then $\Lambda^{-1}(\bigcup_{0 \leq l < j} B_{i,l}) = \bigcup_{0 \leq l \leq k'} B_{i,l}$ for some k'. Again, since $|\Lambda^{-1}(B_{i,j})| = kq^i$ for some k, we have $\Lambda^{-1}(B_{i,j}) = \bigcup_{k' < l \leq k+k'} B_{i,l}$.

We now prove the lemma itself. Suppose $a, b \in A^n$ satisfy $(a_i, \dots, a_n) = (b_i, \dots, b_n)$, then $a, b \in B_{i-1,j}$ where $j = \sum_{l=i}^n a_l$. By our claim, $\Lambda(B_{i-1,j})$ is an interval contained in a block $B_{i-1,k}$ for some k. Hence $\Lambda(a)_l = \Lambda(b)_l$ for all $l \geq i$. $\square$

Lemma 4. *[Extension of [7, Proposition 23]] Let f be a permutation of A^n which can be computed as a product of n instructions updating y_1 to y_n. Then for any proper integer partition λ of q^n, the transformation $g = f \circ \Lambda$ can also be computed as a product of n instructions updating y_1 to y_n.*

Proof. Let $f = f^n \circ \cdots \circ f^1$, where f^i is an instruction updating y_i for all i. Let g^i be the transformation obtained after the instructions $y_m \leftarrow g_m(y)$ for m from 1 to i; we have

$$g^i(x) = (g_1(x), \dots, g_i(x), x_{i+1}, \dots, x_n).$$

Then we only need to prove that for all $1 \leq i \leq n-1$ and all $a \geq b \in A^n$, $g^i(a) = g^i(b) \Rightarrow g(a) = g(b)$.

For any $m \leq i$, we have $g^i_m = g_m = (f \circ \Lambda)_m = f_m \circ p$. Therefore, $g^i(a) = g^i(b)$ if and only if $f_m(\Lambda(a)) = f_m(\Lambda(b))$ for all $m \leq i$ and $a_l = b_l$ for all $l \geq i+1$. By Lemma 3, we obtain $\Lambda(a)_l = \Lambda(b)_l$ for all $l \geq i+1$. Thus $g^i(a) = g^i(b)$ implies $h(\Lambda(a)) = h(\Lambda(b))$, where

$$h(x) = (f^i \circ \cdots \circ f^1)(x) = (f_1(x), \dots, f_i(x), x_{i+1}, \dots, x_n).$$

Since h is a permutation, we obtain $\Lambda(a) = \Lambda(b)$ and hence $g(a) = g(b)$. $\square$

We can now extend the particular method invented in [6] and [7, Section 5] to produce programs for arbitrary transformations of $\{0,1\}^n \to \{0,1\}^n$ to the case of general, non-necessarily Boolean alphabets. However, the result we obtain is not new, for it already appears in the switching network literature (see [7] and references therein).

Theorem 3. *[Extension of [7, Theorem 25]] Any transformation of A^n can be computed by a program with at most $4n-3$ instructions.*

Proof. Let f be a transformation of A^n and consider the integer partition μ of q^n with $\mu_a = |f^{-1}(a)|$ for all $a \in A^n$. Sort μ properly: we obtain $\lambda_a = |f^{-1}(h(a))|$ for some permutation h of A^n. Then f can be expressed as $f = h \circ \Lambda \circ g$, where g is a permutation of A^n satisfying

$$g(f^{-1}(h(a))) = \left[\sum_{b=0}^{a-1} \lambda_b, \sum_{b=0}^{a} \lambda_b\right),$$

for all $a \in A^n$.

By Theorem 2, g and h can be computed as follows, where the superscript indicates which coordinate is updated by each instruction:

$$g = \bar{g}^1 \circ \cdots \circ \bar{g}^{n-1} \circ g^n \circ \cdots \circ g^1,$$
$$h = \bar{h}^1 \circ \cdots \circ \bar{h}^{n-1} \circ h^n \circ \cdots \circ h^1.$$

By Lemma 4, the transformation $h^n \circ \cdots \circ h^1 \circ \Lambda$ can be computed in n instructions $\Lambda^n \circ \cdots \circ \Lambda^1$. Furthermore, Λ^1 and $\bar{g}^1$ being instructions updating y_1, their product $q^1 = \Lambda^1 \circ \bar{g}^1$ is another instruction updating y_1. Thus, f can be computed by the following program of length $4n-3$:

$$f = \bar{h}^1 \circ \cdots \circ \bar{h}^{n-1} \circ \Lambda^n \circ \cdots \circ \Lambda^2 \circ q^1 \circ \bar{g}^2 \circ \cdots \circ \bar{g}^{n-1} \circ g^n \circ \cdots \circ g^1.$$

$\square$

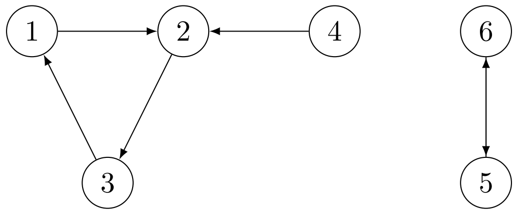


Figure 1: Representing a transformation via a graph.

4 Manipulating variables

We generalise the XOR swap by considering what we call manipulations of variables. We distinguish between a transformation ϕ of $[n]$ (where we denote $[n] = \{1, \dots, n\}$) which represents the formal movement of variables and the transformation f^ϕ of A^n it induces on all the possible values of the variables. We remark that $f^\phi \in \mathrm{Sym}(A^n)$ if and only if $\phi \in \mathrm{Sym}(n)$. We always use the postfix notation for ϕ, i.e. the image of i under ϕ is denoted as $i\phi$. For $\phi : [n] \to [n]$, ϕ^k does represent the k-th power of ϕ according to composition.

Definition 10. A *manipulation of variables* is a transformation f^ϕ of A^n of the form

$$f^\phi(x_1, \dots, x_n) = (x_{1\phi}, \dots, x_{n\phi})$$

for some transformation ϕ of $[n]$.

The transformation ϕ can be represented using a directed graph on $[n]$ with n arcs $(i, i\phi)$ (see [17] for a detailed review of this representation of transformations). This directed graph has cycles of two kinds:

- A cycle $(i, i\phi, \dots, i\phi^{k-1})$ (where $i\phi^k = i$) is *detached* if for all $0 \le l \le k-1$, there is no $j_l \neq i\phi^{l-1}$ such that $j_l\phi = i\phi^l$. Equivalently, the cycle is an entire connected component of the graph.
- A cycle $(i, i\phi, \dots, i\phi^{k-1})$ is *attached* otherwise, i.e. if there exists $0 \le l \le k-1$ and $j \in [n], j \neq i\phi^{l-1}$ such that $j\phi = i\phi^l$.

Note that if ϕ is a permutation, then all its cycles are detached.

For instance, consider $\phi : [6] \to [6]$ defined as $1\phi = 2$, $2\phi = 3$, $3\phi = 1$, $4\phi = 2$, $5\phi = 6$, $6\phi = 5$. Then the cycle $(1, 2, 3)$ is attached to 4, while the cycle $(5, 6)$ is detached, as seen on Figure 1.

Let us first consider the case of a cyclic shift of variables. A similar result to Proposition 6 below is given in [7].

Proposition 6. *Let $\kappa = (1, 2, \dots, n)$. Then the cyclic shift of n variables $f^\kappa : A^n \to A^n$ can be computed in $n+1$ instructions if and only if the order of updates (up to starting point) is $y_1, y_n, \dots, y_2, y_1$.*

Proof. Let us prove that if the order is correct, then we can compute the cyclic shift. This is done via the following program:

$$\begin{aligned} y_1 &\leftarrow \sum_{i=1}^n y_i \\ y_n &\leftarrow y_1 - \sum_{j=2}^n y_j \\ &\vdots \\ y_1 &\leftarrow y_1 - \sum_{j=2}^n y_j. \end{aligned}$$

We prove the correctness of this program by induction: we claim that after the update of y_{n-i}, all variables $y_n, y_{n-1}, \dots, y_{n-i}$ have the correct values $x_1, x_n, \dots, x_{n-i+1}$ for i from 0 to $n-1$. For $i = 0$, we have

$$y_n \leftarrow y_1 - \sum_{j=2}^{n} y_j = \sum_{i=1}^{n} x_i - \sum_{j=2}^{n} x_j = x_1.$$

Now suppose it holds for up to $i-1$, we then have

$$y_{n-i} \leftarrow y_1 - \sum_{j=2}^{n-i} y_j - \sum_{j=n-i+1}^{n} y_j = \sum_{i=1}^{n} x_i - \sum_{j=2}^{n-i} x_j - \sum_{k=n-i+2}^{n} x_k - x_1 = x_{n-i+1}.$$

We now prove the reverse implication. Consider a program computing the shift of variables with $n+1$ instructions, and let y_1 be updated first. Then, suppose y_i is updated before y_{i+1}. After $y_i \leftarrow x_{i+1}$, the content of (y_i, y_{i+1}) is (x_{i+1}, x_{i+1}) and the resulting transformation is not a permutation. Thus, for any $1 \leq i \leq n-1$, the update of y_i must occur after that of y_{i+1} and the only possible order of updates is $y_1, y_n, \dots, y_1$. □

We can then determine the memoryless complexity of any manipulation of variables. The case of a permutation of variables is also given in [7] and [14].

Theorem 4. *Let $\phi : [n] \to [n]$ have F fixed points and D detached cycles. Then the memoryless complexity of the manipulation of variables $f^\phi : A^n \to A^n$ is exactly*

- *$n - F + D$ instructions if ϕ is a permutation;*
- *$n - F + 1$ instructions if ϕ is not a permutation and $D > 0$;*
- *$n - F$ instructions otherwise.*

Proof. Let us first suppose that ϕ is a permutation. Then computing one cycle after the other yields a program of length $n-F+D$ by Proposition 6. Conversely, assume that there is a program computing f^ϕ in fewer than $n - F + D$ instructions. For this program there must be at least one cycle of ϕ such that each coordinate in the cycle is updated only once; denote the first update for that cycle as $y_i \leftarrow x_{i\phi}$. Then after the update, we have $y_i = y_{i\phi} = x_{i\phi}$ and hence the resulting transformation is not a permutation.

Let us now suppose that ϕ is not a permutation. Let m denote the number of variables which are not fixed and do not belong to any cycle. The subgraph induced on these vertices is acyclic, hence we can order them as $a_1, \dots, a_m$ such that $a_i = a_j\phi$ only if $i > j$ [3]. The first part of the program consists in updating all these vertices but the last in the correct order: for i from 1 to $m-1$, do

$$y_{a_i} \leftarrow y_{a_i\phi}.$$

The second part is to perform the cycles by using y_{a_m} as memory. Let $\{i_c : 1 \leq c \leq C\}$ denote a member of each (detached or attached) cycle of length l_c, then do the following instruction:

$$y_{a_m} \leftarrow \sum_{c=1}^{C} y_{i_c}.$$

Then for all c from 1 to C do

$$\begin{aligned} y_{i_c} &\leftarrow y_{i_c\phi} \\ &\vdots \\ y_{i_c\phi^{l_c-2}} &\leftarrow y_{i_c\phi^{l_c-2}} \\ y_{i_c\phi^{l_c-1}} &\leftarrow y_{a_m} - \sum_{b=1}^{c-1} y_{i_b\phi^{l_b-1}} - \sum_{b=c+1}^{C} y_{i_b}. \end{aligned}$$

It can be easily proved by induction on c that this program does compute all cycles. Eventually, we need the final update of y_{a_m}. Note that $a_m\phi$ is either a fixed point or it belongs to a cycle; therefore $x_{a_m\phi}$ is contained in $y_{a_m\phi^L}$, where $L = 0$ if $a_m\phi$ is a fixed point and $L = l_c - 1$ if it belongs to the cycle c. Thus, the final update is given by

$$y_{a_m} \leftarrow y_{a_m\phi^L}. \tag{1}$$

Since y_{a_m} is the only coordinate updated twice, this program has length $n - F + 1$.

We now simplify this program when ϕ has no detached cycles. This time, for i from 1 to m, do

$$y_{a_i} \leftarrow y_{a_i\phi}.$$

Then for all c from 1 to C, there exists $\alpha_c \in \{a_1, \dots, a_m\}$ such that $\alpha_c\phi = i_c$, therefore do

$$\begin{aligned} y_{i_c} &\leftarrow y_{i_c\phi} \\ &\vdots \\ y_{i_c\phi^{l_c-2}} &\leftarrow y_{i_c\phi^{l_c-2}} \\ y_{i_c\phi^{l_c-1}} &\leftarrow y_{\alpha_c}. \end{aligned}$$

Since y_{a_m} already contains $x_{a_m\phi}$, there is no need to include the final update in (1).

Conversely, it is clear that at least $n - F$ instructions are needed to compute f^ϕ. Furthermore, assume $D > 0$ and that there is a program computing f^ϕ in exactly $n - F$ instructions. Let i in the cycle c be the first coordinate belonging to a detached cycle to be updated. Then the program first does $y_i \leftarrow x_{i\phi}$ and the value of x_i is lost; therefore, the update $y_{i\phi^{l_c-1}} \leftarrow x_i$ cannot occur. □

Theorem 4 indicates that disjoint cycles of a permutation cannot be computed "concurrently," for the shortest program which computes two cycles exactly consists of computing one before the other.

Corollary 1. *If $n = 2m$, then computing m disjoint transpositions of variables (e.g. $(1,2)$ $(3,4)$ $\cdots$ $(2m-1, 2m)$) takes exactly $3m$ instructions. If $n = 2m + 1$, then computing $m - 1$ disjoint transpositions and a cycle of length 3, (e.g. $(1,2)(3,4)\cdots(2m-3, 2m-2)(2m-1, 2m, 2m+1)$) takes exactly $3m + 1$ instructions. This is the maximum number of instructions for any manipulation of variables.*

In particular, if $x_1, \dots, x_{m^2}$ are the entries of an $m \times m$ matrix over A, then transposing that matrix takes exactly $3m(m-1)/2$ instructions.

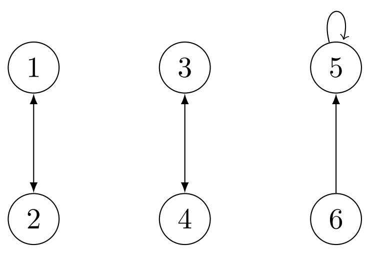
Another consequence of Theorem 4 is that when ϕ is not a permutation, we can obtain shorter programs by combining registers than by adopting the "black box" approach used for the swap of two variables described in the very beginning of the paper. Figure 2 shows the smallest example: computing f^ϕ takes 6 instructions when using the program described in the proof of Theorem 4, while it takes 7 instructions when we do not combine variables. Clearly, this example can be generalized by adding more cycles, thus yielding an arbitrarily large gap between the two approaches. The results are summarised in Proposition 7 and Corollary 2. We say an instruction is a *black box instruction* if it is of the form $y_i \leftarrow y_j$ for $i, j \in [n]$.

Proposition 7. *Let ϕ be a transformation of $[n]$ with F fixed points and D detached cycles. Then the manipulation of variables f^ϕ can be computed without memory by black box instructions if and only if ϕ is not a permutation (or is the identity). In that case, the shortest length of a black box program is $n - F + D$.*

Proof. The proof calls arguments similar to those used above; as such, we use the same notation. We further enforce that the last $D - C$ elements a_i are attached to different cycles, i.e. $a_{m-D+C+c}$ is attached to the cycle c.

The following program computes f^ϕ in $n - F + D$ instructions. First, for i from 1 to $m - D + C$ do

$$y_{a_i} \leftarrow y_{a_i\phi}.$$


(a) ϕ

With combinations		Black box	
$y_6 \leftarrow y_1 + y_3$	$(= x_1 + x_3)$	$y_6 \leftarrow y_1$	$(= x_1)$
$y_1 \leftarrow y_2$	$(= x_2)$	$y_1 \leftarrow y_2$	$(= x_2)$
$y_2 \leftarrow y_6 - y_3$	$(= x_1)$	$y_2 \leftarrow y_6$	$(= x_1)$
$y_3 \leftarrow y_4$	$(= x_4)$	$y_6 \leftarrow y_3$	$(= x_3)$
$y_4 \leftarrow y_6 - y_2$	$(= x_3)$	$y_3 \leftarrow y_4$	$(= x_4)$
$y_6 \leftarrow y_5$	$(= x_5)$	$y_4 \leftarrow y_6$	$(= x_3)$
		$y_6 \leftarrow y_5$	$(= x_5)$

(b) Programs for f^{ϕ}

Figure 2: The simplest manipulation of variables using a shorter program with arithmetic

Second, compute all detached cycles using y_{a_m} as memory. For the detached cycle $\{i, i\phi, \dots, i\phi^{l-1}\}$, do

$$\begin{aligned}
y_{a_m} &\leftarrow y_i \\
y_i &\leftarrow y_{i\phi} \\
&\vdots \\
y_{i\phi^{l-2}} &\leftarrow y_{i\phi^{l-1}} \\
y_{i\phi^{l-1}} &\leftarrow y_{a_m}.
\end{aligned}$$

This uses one extra instruction per detached cycle, i.e. D extra instructions in total. Third, compute all the attached cycles, using $a_{m-D-C+c}$ as memory for the cycle c (similar as above). This does not add any extra instruction.

It is clear that computing a detached cycle using instructions of the form $y_i \leftarrow y_j$ requires using another variable as memory, and hence an extra instruction. However, since this variable gets a value from only one detached cycle, it cannot be re-used for the computation of any other detached cycle. Thus, we need at least D extra instructions. $\square$

It is worth noting that the proof of Proposition 7 does not use the fact that we are computing without memory. Therefore, the black-box computation will always take $n - F + D$ instructions, regardless of how much memory is used.

Corollary 2. *If ϕ is not a permutation, then the ratio between the memoryless complexity of f^{ϕ} over the minimum length of a black box program computing f^{ϕ} is always greater than $2/3$. Conversely, for any $\epsilon > 0$, there exists ϕ for which that ratio is lower than $2/3 + \epsilon$.*

Proof. It takes at least $n - F$ instructions to compute f^{ϕ} without memory, and exactly $n - F + D$ instructions to do it using black box instructions. Since $n - F \geq 2D$, we easily obtain the lower bound of $2/3$.

Conversely, for any $k \geq 1$, let $n = 2k + 2$ and $\phi : [n] \to [n]$ be defined as

$$\phi = (1, 2) \cdots (2k - 1, 2k)(2k + 2 \to 2k + 1).$$

Then for any A, f^{ϕ} can be computed in n instructions, but takes $3n/2 - 2$ instructions of the form $y_i \leftarrow y_j$. $\square$

5 Using additional registers

In this section, we consider the case of computing f a function of n registers, when $n + m$ registers are available to manipulate and the additional m registers can be updated as we wish. In order to differentiate them, we shall refer to these additional registers as *memory cells*.

By convention, we shall denote the content of the m memory cells as $y_{n+1}, \dots, y_{n+m}$; we still use $y = (y_1, \dots, y_n)$. Then computing f using m memory cells is equivalent to computing some transformation $h(x_1, \dots, x_{n+m})$ of A^{n+m} such that the first n coordinate functions of h coincide with those of f. Let us denote the set of such transformations as $D(f, m)$. The shortest length of a program computing f using m memory cells is hence given by

$$\mathcal{L}(f|m) := \min_{h \in D(f,m)} \mathcal{L}(h).$$

Therefore, there exists h such that $\mathcal{L}(h) = \mathcal{L}(f|m)$ but it may be difficult to characterise that transformation h. However, Proposition 8 shows that there is a deterministically (and easily) described transformation $h \in D(f, m)$ for which $\mathcal{L}(h)$ and $\mathcal{L}(f|m)$ are in bijection. Therefore, the memoryless computation framework also considers the case of using memory.

Proposition 8. *For any transformation f of A^n and any $e = (e_1, \dots, e_m) \in A^m$, let $h^e \in D(f, m)$ and $h^e_{n+i} = e_i$ for $1 \le i \le m$. Then*

$$\mathcal{L}(h^e) = \mathcal{L}(f|m) + m.$$

Proof. Let $g \in D(f, m)$ such that $\mathcal{L}(g) = \mathcal{L}(f|m)$, then the shortest program computing g appended with the suffix $y_{n+i} \leftarrow e_i$ for i from 1 to m has length $\mathcal{L}(f|m) + m$ and computes h^e. Therefore, $\mathcal{L}(h^e) \le \mathcal{L}(f|m) + m$.

Conversely, consider the shortest program computing h^e. It contains m final updates $y_{n+i} \leftarrow e_i$ which, without loss, appear for i from m down to 1. Then any instruction $y_j \leftarrow g(y)$ occurring after $y_{n+k} \leftarrow e_k$ (hence $j \le n + k - 1$) can be replaced by $y_j \leftarrow g'(y_1, \dots, y_{n+k-1})$ where $g' : A^{n+k-1} \to A$ is defined as

$$g'(y_1, \dots, y_{n+k-1}) = g(y_1, \dots, y_{n+k-1}, e_k, \dots, e_m).$$

Now remove all the $y_{n+i} \leftarrow e_i$ updates; we are left with a program computing some transformation in $D(f, m)$ of length $\mathcal{L}(h^e) - m$. Thus $\mathcal{L}(f|m) \le \mathcal{L}(h^e) - m$. □

There is a linear analogue to Proposition 8. Namely, if $M \in \mathrm{GF}(q)^{n \times n}$, let $N \in \mathrm{GF}(q)^{n+m \times n} = (N, 0)$. Then it is easily shown that, when limiting ourselves to linear instructions, the memoryless complexity of N is equal to m plus the minimum length of a program computing M with m memory cells.

5.1 Shorter programs

We have shown in Theorem 1 that one need not use memory to compute any transformation. However, we shall prove that one may want to use memory in order to use shorter programs.

We have shown in Theorem 2 that any permutation can be computed without memory in at most $2n - 1$ instructions. On the other hand, using one memory cell necessarily yields a program with length at least $n + 1$. Propositions 1 and 9 show that these two results are simultaneously tight: there exists a permutation $f \in \mathrm{Sym}(A^n)$ for which $\mathcal{L}(f) = 2n - 1$ while $\mathcal{L}(f|1) = n + 1$.

Proposition 9. *The transposition (a, b) of two states $a, b \in A^n$ at Hamming distance d can be computed with one memory cell in $d + 1$ instructions: $\mathcal{L}((a, b)|1) = d + 1$.*

Proof. Without loss, let a and b disagree on their first d coordinates. Then the following program computes (a, b):

$$\begin{aligned} y_{n+1} &\leftarrow \delta(y, a) - \delta(y, b) \\ y_1 &\leftarrow y_1 + (b_1 - a_1) y_{n+1} \\ &\vdots \\ y_d &\leftarrow y_d + (b_d - a_d) y_{n+1}. \end{aligned}$$

□

In Theorem 3, we have given an upper bound on the complexity of any transformation which only depends on the number of variables. This upper bound is larger than $2n-1$ obtained for permutations; however, using memory cells yields a program using $2n-1$ instructions, as seen below.

Proposition 10. *Any transformation f of A^n can be computed with $n-1$ memory cells and no more than $2n-1$ instructions: $\mathcal{L}(f|n-1) \leq 2n-1$.*

Proof. The following program computes f using $n-1$ memory cells and $2n-1$ instructions:

$$\begin{aligned}
y_{n+1} &\leftarrow y_1\\
&\vdots\\
y_{2n-1} &\leftarrow y_{n-1}\\
y_1 &\leftarrow f_1(y_{n+1}, \dots, y_{2n-1}, y_n)\\
&\vdots\\
y_n &\leftarrow f_n(y_{n+1}, \dots, y_{2n-1}, y_n).
\end{aligned}$$

□

Proposition 10 indicates that we do not need any more than $n-1$ memory cells. Indeed, if we use n memory cells, then the program will have at least $2n$ instructions (unless some memory cells are not updated, which is equivalent to not using them). Therefore, $\mathcal{L}(f|m) = \mathcal{L}(f|n-1)$ for any $m \geq n-1$.

We remark that this upper bound on the amount of memory needed follows from the fact that we allow any instruction. In practice, using a large amount of memory is the price paid for using only a restricted number of basic instructions.

The ideas behind [6, Theorem 3.1] (i.e. any permutation has memoryless complexity at most $2n-1$) can be adapted to the case of using memory to yield a refinement of Proposition 10 for permutations.

Theorem 5. *If $n = 2m$ is even, then any permutation of A^n can be computed in at most $3m$ instructions with m memory cells. If $n = 2m+1$ is odd, then any permutation of A^m can be computed in at most $3m+3$ instructions with $m+2$ memory cells.*

Proof. The main tool of the proof of [6, Theorem 3.1] is that for any two functions $f, g : B \times C \to B$ such that the pre-image of each element of B has cardinality $|C|$ under f or g, there exists $h : B \times C \to C$ such that $(f,h), (g,h) : B \times C \to B \times C$ are bother permutations of $B \to C$.

Suppose $n = 2m$ and let $f \in \mathrm{Sym}(A^n)$. By the result above, there exist m functions $g_1, \dots, g_m : A^n \to A$ such that

$$(f_1, \dots, f_m, g_1, \dots, g_m) \qquad \text{and} \qquad (x_{m+1}, \dots, x_n, g_1, \dots, g_m)$$

both form permutations of A^n. The program goes as follows:

- Step 1 (m instructions). For i from 1 to m, do $y_{n+i} \leftarrow g_i(x)$.
- Step 2 (m instructions). For i from 1 to m, do $y_i \leftarrow f_i(x)$. This is possible since

  $$(x_{m+1}, \dots, x_n, g_1, \dots, g_m)$$

  form a permutation of A^n, and hence $f_i(x)$ can be expressed as a function of

  $$(y_{m+1}, \dots, y_n, y_{n+1}, \dots, y_{n+m}).$$

- Step 3 (m instructions). For i from $m+1$ to n, do $y_i \leftarrow f_i(x)$. This is possible since $(f_1, \dots, f_m, g_1, \dots, g_m)$ form a permutation of A^n, and hence $f_i(x)$ can be expressed as a function of $(y_1, \dots, y_m, y_{n+1}, \dots, y_{n+m})$.

Now let $n = 2m + 1$ be odd. Then add one memory cell and consider the extended permutation $g \in D(f, 1)$ such that $g_{2m+2}(x) = x_{2m+2}$. Then g can be computed in $3m + 3$ instructions and $m + 1$ memory cells. □

Therefore, we do not want more than around $n/2$ memory cells to compute any permutation; adding any more would be superfluous. There is a linear analogue to Theorem 5.

Proposition 11. *If $n = 2m$ is even, then any linear permutation of A^n can be computed in at most $3m$ linear instructions with m memory cells. If $n = 2m+1$ is odd, then any linear permutation of A^n can be computed in at most $3m + 3$ linear instructions with $m + 2$ memory cells.*

Proof. Suppose $n = 2m$. Let $f(x) = xM^\top$ and denote the first m rows of M as M_1 and the matrix $J = (0|I_m) \in A^{m\times n}$. We claim that there exists a matrix $N \in A^{m\times n}$ such that $(M_1^\top, N^\top)$ and $(J^\top, N^\top)$, both in $A^{n\times n}$, are nonsingular. Then the algorithm simply places N in the memory, then replaces the first m rows by M_1, and finally updates the last m rows to those of M.

We now justify our claim. This is equivalent to showing that for any two subspaces in the Grassmannian $G(q, 2m, m)$ of m-dimensional subspaces of $\mathrm{GF}(q)^{2m}$, there exists a third subspace in the same Grassmannian at subspace distance $2m$ from both [22] (where the subspace distance between $U, V \in G(q, 2m, m)$ is given by $2\dim(U + V) - 2m$). Since the Grassmannian endowed with the subspace distance forms an association scheme [15], we only have to check for the row space of J and one subspace at distance $2d$ for each $0 \le d \le m$. Let us then assume $M_1 = (0_{m-d}|I_m|0_d)$ whose row space is at subspace distance $2d$ from that of J. Then it is easily checked that the row space of

$$N = \left(\begin{array}{c|c|c} I_m & 0_d & \begin{array}{c} 0_{m-d} \\ \hline I_{m-d} \end{array} \end{array} \right)$$

is at distance $2m$ from the row spaces of M_1 and J.

The case $n = 2m + 1$ is settled by considering $M' \in A^{n+1\times n+1}$ given by

$$M' = \left(\begin{array}{c|c} M & 0 \\ \hline 0 & 1 \end{array} \right).$$

□

For manipulations of variables, we can completely determine the gain offered by using memory. In particular, using only one memory cell is optimal to compute any manipulation of variables.

Proposition 12. *Any manipulation of n variables with F fixed points can be computed with one memory cell in at most $n - F + 1$ instructions.*

Proof. By Theorem 4, we only need to prove the case where ϕ is a permutation of $[n]$. Let π be the transformation of $[n + 1]$ defined as $i\pi = i\phi$ for all $i \in [n]$ and $(n + 1)\pi = 1$. Then by Theorem 4, we can compute f^π in $n - F + 2$ instructions, where the last instruction updates y_{n+1}. By removing that last instruction, we compute f^ϕ in $n - F + 1$ instructions while using one memory cell y_{n+1}. □

By comparing with Theorem 4, we see that using only one memory cell reduces the length of the program from $n - F + D$ to $n - F + 1$ for permutations. In particular, for a disjoint product of m transpositions, the complexity goes down from $3m$ to only $2m + 1$.

Example 2. Let $\pi = (1, 2)(3, 4) \in \mathrm{Sym}([4])$ and let $f^\pi : A^4 \to A^4$ be the corresponding permutation of variables. By Corollary 1, two disjoint transpositions of variables must be computed in at least 6 instructions when no memory is used. However, adjoining one memory cell y_5 leads to a program with only 5 instructions, as seen below.

$$\begin{array}{lcll} y_5 & \leftarrow & y_1 + y_3 & (y_5 = x_1 + x_3) \\ y_1 & \leftarrow & y_2 & (y_1 = x_2) \\ y_2 & \leftarrow & y_5 - y_3 & (y_2 = x_1) \\ y_3 & \leftarrow & y_4 & (y_3 = x_4) \\ y_4 & \leftarrow & y_5 - y_2 & (y_4 = x_3) \end{array}$$

5.2 Binary instructions

Since the number of instructions $|\bar{\mathcal{I}}|$ is very large (see Lemma 1), one may want to use only a subset of instructions to compute any transformation. A natural choice is that of binary instructions, since any function can be computed as a composition of binary operations.

Definition 11. An instruction $y_i \leftarrow g_i(y)$ is *binary* if g only involves at most two variables: $g_i(y) = g_i(y_j, y_k)$ for some $j, k \in [n]$.

Using binary instructions is not sufficient when computing without memory; however, it is sufficient when only one memory cell is used.

Theorem 6. *If $A = \mathrm{GF}(2)$, then the set of all permutations of A^n which can be computed using binary instructions is the affine group $\mathrm{Aff}(n,2)$. On the other hand, when using one memory cell, any transformation over any alphabet can be computed by binary instructions.*

Proof. Note that any binary permutation instruction is of the form $y_i \leftarrow g_i(y_i, y_j)$ for some $j \in [n]$. If $A = \mathrm{GF}(2)$ and $n = 2$, then it is well known that $\mathrm{Sym}(\mathrm{GF}(2)^2) = \mathrm{Aff}(2,2)$. If $n > 2$, then any instruction of the form $y_i \leftarrow g(y_i, y_j)$ must correspond to a binary instruction for $\mathrm{GF}(2)^2$ acting on the coordinates y_i, y_j: it is also affine. Therefore, the group generated by binary permutation instructions is affine. Conversely, extending Gaussian elimination to the affine case shows that any affine permutation can be computed via binary instructions.

If one memory cell is used, we claim that the instructions in Theorem 1 can be computed by binary instructions. For the sake of simplicity, let us assume $i = 1$. For any $u \in A^n$ and $v = u + e^1$, we can decompose

$$\begin{aligned} \delta(y,u) &= \delta(y_1,u_1)\delta(y_2,u_2)\cdots\delta(y_n,u_n), \\ \delta(y,u) - \delta(y,v) &= (\delta(y_1,u_1) - \delta(y_1,v_1))\delta(y_2,u_2)\cdots\delta(y_n,u_n). \end{aligned}$$

Then the transposition (u, v) is computed as follows:

$$\begin{aligned} y_{n+1} &\leftarrow \delta(y_1,u_1) - \delta(y_1,v_1) \\ y_{n+1} &\leftarrow y_{n+1}\delta(y_2,u_2) \\ &\vdots \\ y_{n+1} &\leftarrow y_{n+1}\delta(y_n,u_n) \\ y_1 &\leftarrow y_1 + y_{n+1} \end{aligned}$$

and the assignment $(e^0 \to e^1)$ is computed as:

$$\begin{aligned} y_{n+1} &\leftarrow \delta(y_1,0) \\ y_{n+1} &\leftarrow y_{n+1}\delta(y_2,0) \\ &\vdots \\ y_{n+1} &\leftarrow y_{n+1}\delta(y_n,0) \\ y_1 &\leftarrow y_1 + y_{n+1}. \end{aligned}$$

Since any transformation can be computed using these two types of instructions, it can be computed with binary instructions. □

6 Acknowledgment

The authors would like to thank Peter J. Cameron, Ben Fairbairn, Peter Keevash and Rasmus Petersen for stimulating discussions. They would also like to thank the anonymous reviewers for their highly valuable comments on the preliminary version of this paper.

References

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, *Network information flow*, IEEE Transactions on Information Theory **46** (2000), no. 4, 1204–1216.

[2] V. E. Beneš, *Optimal rearrangeable multistage connecting networks*, Bell System Technical Journal **43** (1964), 1641–1656.

[3] J.A. Bondy and U.S.R. Murty, *Graph theory*, Graduate Texts in Mathematics, vol. 244, Springer, 2008.

[4] Serge Burckel, *Closed iterative calculus*, Theoretical Computer Science **158** (1996), 371–378.

[5] ———, *Elementary decompositions of arbitrary maps over finite sets*, Journal of Symbolic Computation **37** (2004), no. 3, 305–310.

[6] Serge Burckel, Emeric Gioan, and Emmanuel Thomé, *Mapping computation with no memory*, Proc. International Conference on Unconventional Computation (Ponta Delgada, Portugal), September 2009, pp. 85–97.

[7] ———, *Computation with no memory, and rearrangeable multicast networks*, Discrete Mathematics and Theoretical Computer Science **16** (2014), 121–142.

[8] Serge Burckel and Marianne Morillon, *Three generators for minimal writing-space computations*, Theoretical Informatics and Applications **34** (2000), 131–138.

[9] ———, *Quadratic sequential computations of boolean mappings*, Theory of Computing Systems **37** (2004), no. 4, 519–525.

[10] ———, *Sequential computation of linear boolean mappings*, Theoretical Computer Science **314** (2004), 287–292.

[11] Peter J. Cameron, Ben Fairbairn, and Maximilien Gadouleau, *Computing in permutation groups without memory*, submitted (2012).

[12] Peter J. Cameron, Ben Fairbairn, and Maximilien Gadouleau, *Computing in matrix groups without memory*, submitted (2012).

[13] Peter J. Cameron, Maximilien Gadouleau, and Søren Riis, *Combinatorial representations*, Journal of Combinatorial Theory, Series A **120** (2013), no. 3, 671–682.

[14] Jean-Paul Delahaye, *Le calculateur amnésique*, Pour La Science **404** (2011), 88–92, Available at `http:///www2.lifl.fr/~delahaye/pls/2011/208`.

[15] Philippe Delsarte, *Association schemes and t-designs in regular semilattices*, Journal of Combinatorial Theory A **20** (1976), no. 2, 230–243.

[16] P. Erdös, A. Ginzburg, and A. Ziv, *Theorem in the additive number theory*, Bulletin of the Research Council of Israel **10F** (1961), 41–43.

[17] Olexandr Ganyushkin and Volodymyr Mazorchuk, *Classical finite transformation semigroups: An introduction*, Algebra and Applications, vol. 9, Springer-Verlag, London, 2009.

[18] Christopher David Godsil and Gordon Royle, *Algebraic graph theory*, Graduate Texts in Mathematics, vol. 207, Springer-Verlag, 2001.

[19] Dah-Jyn Guan, *Generalized Gray codes with applications*, Proc. Natl. Sci. Counc. ROC(A) **22** (1998), no. 6, 841–848.

[20] John Hennessy and David Patterson, *Computer architecture: a quantitative approach*, 5th ed., Morgan Kaufmann, 2011.

[21] F. K. Hwang, *The mathematical theory of nonblocking switching networks*, World Scientific Publishing Co., 2004.

[22] Ralf Kötter and Frank R. Kschischang, *Coding for errors and erasures in random network coding*, IEEE Transactions on Information Theory **54** (2008), no. 8, 3579–3591.

[23] C. D. Thompson, *Generalized connection network for parallel processor interconnection*, IEEE Transactions on Computers **27** (1978), 1119–1124.

[24] Raymond W. Yeung, Shuo-Yen Robert Li, Ning Cai, and Zhen Zhang, *Network coding theory*, vol. 2, Foundation and Trends in Communications and Information Theory, no. 4-5, now Publishers, Hanover, MA, 2006.