

COMPUTING WITHOUT MEMORY

MAXIMILIEN GADOULEAU* AND SØREN RIIS†

Abstract. In this paper, we generalise the famous algorithm for swapping the contents of two variables without using a buffer. We introduce a novel combinatorial framework for procedural programming languages, where programs are only allowed to update one variable at a time. We first consider programs which do not have any memory. We prove that any function of all the variables can be computed this way in a number of updates which grows linearly with the number of variables. Similarly, any linear function can be computed using a linear number of linear instructions. We then derive the exact number of instructions required to compute any manipulation of variables. This shows that the idea of 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 our framework. We 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.

Key words. Models of computation, Computational difficulty, Symmetric group, Theory of data, Combinatorics

AMS subject classifications. 68Q05

1. Introduction. How do you swap the contents of variables x and y using a procedural programming language? The common approach is to use a buffer t , and to do as follows (using pseudo-code).

$$\begin{aligned} t &\leftarrow x \\ x &\leftarrow y \\ y &\leftarrow t. \end{aligned}$$

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

$$\begin{aligned} x &\leftarrow x + y \\ y &\leftarrow x + y \\ x &\leftarrow x + y. \end{aligned}$$

We thus perform the swap without any use of memory. Our aim is to generalise this idea to compute transformations without memory.

While the example described above is folklore, the idea to compute functions without memory was developed in [2, 3, 4, 5, 6, 7] for the case of boolean variables. We would like to emphasize the novelty of the results of this paper and how they differ from those in the literature. First, the results presented in this paper generalise those given in the literature, as we consider any finite alphabet while only the binary alphabet was usually considered in the literature. Second, we provide simpler proofs, which is especially true for Theorems 2.4 and 3.13. Third, we also give some matching upper and lower bounds which are absent in the literature, e.g. in Theorem 3.5. Fourth, many aspects considered here, such as the study of manipulations of variables in Section 4.3, the use of binary instructions in Theorem 5.8 and the use of memory in Section 5, are completely novel.

2. Combinatorial model for memoryless computations.

2.1. Instructions and programs. We formalise our ideas as follows. 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 $\text{GF}(q)$ when q is a prime power). The cases where $q = 1$ or $n = 1$ being trivial, we shall assume $q \geq 2$ and $n \geq 2$ henceforth. We refer to any element of A^n as a word. We view any transformation f of A^n (i.e., $f : A^n \rightarrow A^n$) as a tuple of functions $f = (f_1, \dots, f_n)$, where $f_i : A^n \rightarrow A$ is referred to as the

*School of Engineering and Computing Sciences, Durham University, Durham, UK. (m.r.gadouleau@durham.ac.uk)

†School of Electronic Engineering and Computer Science, Queen Mary, University of London, London, UK. (sm-riis@eecs.qmul.ac.uk)

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. The size of the image of f is referred to as its rank. When considering a sequence of transformations, we shall use superscripts, e.g. $f^k : A^n \rightarrow A^n$ for all k —and hence f^k shall never mean taking f to the power k .

DEFINITION 2.1 (Instruction). *An instruction is a transformation g of A^n with at most one nontrivial coordinate function g_i . 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 convention, 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 $A = \text{GF}(2)$ 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:

$$\{y_1 \leftarrow y_1, \quad y_1 \leftarrow y_1 + 1, \quad y_1 \leftarrow y_1 + y_2, \quad y_1 \leftarrow y_1 + y_2 + 1, \\ y_2 \leftarrow y_2, \quad y_2 \leftarrow y_2 + 1, \quad y_2 \leftarrow y_1 + y_2, \quad y_2 \leftarrow y_1 + y_2 + 1\},$$

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

DEFINITION 2.2 (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*

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

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 .

We consider a basic procedural programming language which has a finite number of inputs $x = (x_1, \dots, x_n) \in A^n$ and only allows programs of the form described above. Therefore, it only allows in-place calculations, without loops, pointers, and more importantly without any memory. We use $y = (y_1, \dots, y_n)$ to represent the content of the registers during the program. Hence $y = x$ before the first instruction, and $y = f(x)$ after the last instruction. Note that we will also use the shortcut notation $y_i \leftarrow h(x)$ to reflect how the content of the memory relates with the program input. In particular, note that the last update of y_i must be

$$y_i \leftarrow f_i(x).$$

To be absolutely rigorous, we should let y take into account the instruction number: $y^0 = x, y^1, \dots, y^L = f(x)$, where L is the length of the program. However, our calculations will not require such level of rigour, and we simply use y instead.

In order to illustrate our notations, let us rewrite the program computing the swap of two variables, i.e. $f : A^2 \rightarrow A^2$ where $f(x_1, x_2) = (x_2, x_1)$. It is given as follows:

$$y_1 \leftarrow y_1 + y_2 \quad (= x_1 + x_2) \\ y_2 \leftarrow y_1 - y_2 \quad (= x_1) \\ y_1 \leftarrow y_1 - y_2 \quad (= x_2).$$

DEFINITION 2.3. *Let B, C be two alphabets and $f, g : B \rightarrow C$. We say g dominates f if and only 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 induces a sequence of transformations $h^1, \dots, h^L = f$ 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 our programming language only allows to return one output: the transformation f computed by the program. However, it may be fair to ask the program to sequentially return outputs. This can be incorporated in our 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 $1 \leq i \leq K - 1$ (our results will show that this is necessary and sufficient). Therefore, this program can be broken down into K shorter programs, each computing one output. In view of these considerations, we shall only consider programs which compute one output 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 prove in Theorem 2.4 that any transformation can be computed. Although the program in the proof has exponential length, we shall prove that any transformation has a program of linear length.

We introduce some useful notations for any words $u, v \in A^n$. 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 word in A^n . Second, the *assignment* of u to v , denoted as $(u \rightarrow v)$, is the transformation which maps u to v and fixes any other word in A^n . Third, we denote the all-zero word as e^0 and the k -th unit word as $e^k \in A^n$, where $e_i^k = \delta(i, k)$ and δ is the Kronecker delta function.

THEOREM 2.4. *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 $(e^0 \rightarrow e^1)$. These instructions are respectively represented by*

$$\begin{aligned} y_i &\leftarrow y_i + \delta(y, u) - \delta(y, v), \\ y_1 &\leftarrow y_1 + \delta(y, e^0). \end{aligned}$$

Proof. First of all, a generating set of $\text{Sym}(A^n)$ together with any transformation of rank $q^n - 1$ generates all transformations [11, Theorem 3.1.3]. Since the assignment $(e^0 \rightarrow e^1)$ is an instruction of rank $q^n - 1$ (clearly represented in the bottom row above in update form), we only need to generate $\text{Sym}(A^n)$.

Order the words of A^n according to the Gray code in [13], then any two consecutive words v^j and v^{j+1} satisfy $v^{j+1} = v^j \pm e^{i_j}$ for some i_j . The Coxeter generators $\{(v^j, v^{j+1}) : 1 \leq j \leq q^n - 1\}$ corresponding to this ordering thus are instructions, e.g. if $v^{j+1} = v^j + e^i$, then (v^j, v^{j+1}) is represented by

$$y_i \leftarrow y_i + \delta(y, v^j) - \delta(y, v^{j+1}).$$

□

Our framework is particularly interesting for computing using registers only, or equivalently without requiring to use primary memory. The instructions in Theorem 2.4 are encoded in assembly in Figure 2.1.

The instructions are explained as follows.

- **bne** $y a 1$ (branch not equal) will jump to the instruction labelled by 1 if $y \neq a$.
- **addi** $y b$ (add immediate) adds b to the value stored in y (without carry-out).
- **j 1** (jump) jumps to the instruction labelled by 1.

3. Procedural complexity. **DEFINITION 3.1** (Procedural complexity). *The shortest length of a program computing f is referred to as the procedural complexity of f and is denoted as $\mathcal{L}(f)$. By convention, the identity has procedural 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

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

		bne	y_1	u_1	12		
		:					
		bne	y_{i-1}	u_{i-1}	12		
		bne	y_{i+1}	u_{i+1}	12		
		:					
		bne	y_n	u_n	12		
		bne	y_i	u_i	11		bne y_1 0 1
		addi	y_i	1			:
		j	12				:
11	:	bne	y_i	v_i	12		bne y_n 0 1
		addi	y_i	-1			addi y_i 1
12	:						

(a) $y_i \leftarrow y_i + \delta(y, u) - \delta(y, v)$
(b) $y_1 \leftarrow y_1 + \delta(y, e^0)$

FIG. 2.1. Encoding the instructions in Theorem 2.4

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

We would like to emphasize that the procedural complexity strongly differs from other measures seen in complexity theory. For instance, the procedural complexity of any decision problem is simply 1, for it can be expressed as computing the instruction whose value is 1 if the instance has an affirmative answer and 0 otherwise. Also, the procedural complexity is based on the set of all instructions, and not only on circuits formed of certain types of gates. Therefore, each instruction can be arbitrarily “complex.”

3.1. Procedural complexity of permutations. The main purpose of this section is to prove that the maximum procedural complexity of a permutation in $\text{Sym}(A^n)$ is $2n - 1$, which is independent from the cardinality of the alphabet A .

Proposition 3.2 below shows that this quantity is at least $2n - 1$. It is remarkable that the permutation which maximises the procedural complexity is very “simple” to describe; this fact highlights the difference between the procedural complexity and other complexity measures.

PROPOSITION 3.2. *The procedural complexity of the transposition (a, b) of two words $a, b \in A^n$ 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 $v^k = (b_1, \dots, b_k, a_{k+1}, \dots, a_n)$ for $1 \leq k \leq d$, we obtain

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

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

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

Therefore, the procedural 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 . \square

To prove an upper bound on the procedural complexity, we need to study the properties of functions. We use the terminology of [9]. Although this upper bound was proved in [4], we give an alternate proof below, which connects the topic of this paper to the study of coordinate functions and combinatorial representations from [9].

DEFINITION 3.3. *Let B, C be two alphabets. A function $f : B \times C \rightarrow B$ is balanced if $|f^{-1}(b)| = |C|$ for all $b \in B$.*

It is easily shown that for any two functions $f : B \times C \rightarrow B$ and $h : B \times C \rightarrow C$, (f, h) is a permutation of $B \times C$ if and only if f is balanced and $h(f^{-1}(b)) = C$ for all $b \in B$ [9].

PROPOSITION 3.4. *For any pair of balanced functions $f, g : B \times C \rightarrow B$, there exists $h : B \times C \rightarrow C$ such that (f, h) and (g, h) are permutations of $B \times C$.*

Proof. Let G be the bipartite graph with vertex set given by two copies of B and with $|(f, g)^{-1}(i, j)|$ edges between i and j . Since f and g are balanced, G is $|C|$ -regular and hence its edges are C -colourable. Let h be such colouring. Then for all $i \in B$, we have $h(f^{-1}(i)) = \bigcup_{j \in B} h((f, g)^{-1}(i, j)) = C$ and similarly $h(g^{-1}(j)) = C$. This is equivalent to (f, h) and (g, h) being permutations. \square

THEOREM 3.5. *The maximum procedural complexity of a permutation of A^n is $2n - 1$ instructions.*

Proof. Proposition 3.2 shows that the maximum complexity is at least $2n - 1$. We then prove that any permutation f can be computed by a program with at most $2n - 1$ instructions.

We prove the following claim: for any $1 \leq k \leq n - 1$, there exists a function $h_k : A^n \rightarrow A$ of x such that $(h_1, \dots, h_k, x_{k+1}, \dots, x_n)$ and $(h_1, \dots, h_k, f_{k+1}, \dots, f_n)$ are permutations. This is clear for $k = 1$: apply Proposition 3.4 to (f_2, \dots, f_n) and (x_2, \dots, x_n) . Let us assume it is true for up to $k - 1$, then by hypothesis, $g^1 := (h_1, \dots, h_{k-1}, x_{k+1}, \dots, x_n)$ and $g^2 := (h_1, \dots, h_{k-1}, f_{k+1}, \dots, f_n)$ are both balanced functions from A^n to A^{n-1} (since (g^1, x_k) and (g^2, f_k) are permutations, respectively). Applying Proposition 3.4 to these functions then proves the claim.

The program then proceeds as follows:

- Step 1. For k from 1 to $n - 1$, do $y_k \leftarrow h_k(x)$.
- Step 2. For k from n down to 1, do $y_k \leftarrow f_k(x)$.

\square

We can represent computations of any permutation of A^n as progressing around the Cayley graph [12]

$$\text{Cay}(\text{Sym}(A^n), \mathcal{I}).$$

The set of permutation instructions $\mathcal{I} \subseteq \text{Sym}(A^n)$ is described as follows. Let g be the instruction $y_i \leftarrow g_i(y)$. Then in view of the remarks made after Definition 3.3, g is a permutation if and only if $g_i : A^n \rightarrow A$ satisfies

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

for all $v \in A^{n-1}$. There are hence $q!$ choices for the reduction of g_i to each pre-image, and hence $(q!)^{q^{n-1}}$ choices for g_i . Since the identity has been counted n times, there are

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

instructions.

Note that the inverse of g is given by the instruction h which also updates the i -th coordinate and satisfies

$$h_i(x_1, \dots, x_{i-1}, g_i(x), x_{i+1}, \dots, x_n) = x_i.$$

Therefore, the set of permutation instructions updating a given coordinate forms a group, isomorphic to $\text{Sym}(A)^{q^{n-1}}$.

We have determined the maximum procedural complexity in Theorem 3.5. We are now interested in the expected complexity. Proposition 3.6 gives a lower bound on that quantity.

PROPOSITION 3.6. *The proportion of permutations with computational 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.

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

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

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

$$\frac{|\mathcal{I}|^{\lfloor B \rfloor}}{|\text{Sym}(A^n)|} < \frac{1}{\sqrt{2\pi q^n}},$$

which tends to zero. \square

In particular, Proposition 3.6 shows that for n large, almost all permutations of $\text{GF}(2)^n$ have computational complexity at least $2n - 2$. However, the bound in Proposition 3.6 decreases with q .

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

DEFINITION 3.7 (Ordered function). *Let A and A^n be ordered (say, using the lexicographic order). For any balanced function $f_i : A^n \rightarrow A$ 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 $m(0) \leq m(1) \leq \dots \leq m(q-1)$.*

Any function $f_i : A^n \rightarrow A$ can be uniquely expressed as

$$f_i = \sigma_i \circ f_i^*$$

where $\sigma_i \in \text{Sym}(A)$ and f_i^* is ordered. In this case, we say that f_i is *parallel* to f_i^* [9].

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 $\sigma_1, \dots, \sigma_n \in \text{Sym}(A)$.

PROPOSITION 3.8. *There exists a shortest program computing f^* using only ordered instructions. Furthermore, its length satisfies*

$$\mathcal{L}(f^*) \leq \mathcal{L}(f) \leq \mathcal{L}(f^*) + T(f), \tag{3.1}$$

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. The proof of the different claims all use the idea of converting programs.

We first prove that there exists a shortest program computing f^* using only ordered instructions. Let $f^* = g^L \circ \dots \circ g^1$ be a shortest program computing f^* . We can easily convert it to another program $h^L \circ \dots \circ h^1$ 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 $1 \leq i \leq n$. 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 $\tau \in \text{Sym}(A)$ guarantees that the instruction h_i^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 $\mathcal{L}(f^*) \leq \mathcal{L}(f)$. 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 $\mathcal{L}(f) \leq \mathcal{L}(f^*) + T(f)$. 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). \square

3.2. Procedural complexity of all transformations. We have shown that any permutation of A^n can be computed in $2n - 1$ memoryless instructions. We have also shown that any transformation can be computed by some memoryless program. The aim of this section is to derive an upper bound on the procedural complexity of any transformation which only depends on n ; this upper bound turns out to be $4n - 3$ instructions.

DEFINITION 3.9 (Lexicographic order). *For any $a = (a_1, \dots, a_n) \in A^n$ ($A = \mathbb{Z}_q$), we define the lexicographic order of a as the integer $\sum_{i=1}^n a_i q^{i-1}$. For the sake of conciseness and clarity of notation, we shall abuse notation and identify a with its lexicographic order.*

In the lexicographic order, the all-zero word is in zero-th position, then $(1, 0, \dots, 0)$ is in first, $(0, 1, 0, \dots)$ is in q -th position, and so on until $(q-1, \dots, q-1)$ in last and $(q^n - 1)$ -th position. The lexicographic order yields the concept of interval, defined below.

DEFINITION 3.10. *An interval of A^n is any subset of the form*

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

for any $0 \leq b \leq c \leq q^n - 1$.

Recall that an integer partition of an integer s is a sequence of positive integers whose sum is equal to s . Although the terms in the sequence are usually sorted in decreasing order, we do not do so in this paper.

DEFINITION 3.11. *For any integer partition $\lambda = (\lambda_1, \dots, \lambda_k)$ of q^n , p^λ is defined to be the transformation of A^n such that*

$$p^\lambda([0, \lambda_1 - 1]) = 0, \\ p^\lambda \left(\left[\sum_{i=1}^{j-1} \lambda_i, \sum_{i=1}^j \lambda_i - 1 \right] \right) = j - 1 \quad \text{for all } 1 \leq j \leq k.$$

PROPOSITION 3.12. *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 integer partition λ of q^n , the transformation $g = f \circ p^\lambda$ can also be computed as a product of n instructions updating y_1 to y_n .*

Proof. In order to simplify notations, we denote p^λ as p . We first prove the following claim: if $a \geq b \in A^n$ agree on coordinates i to n for some i , then $p(a)_l = p(b)_l$ for all $l \geq i$.

Proof of claim: We have $p(c+1) \in \{p(c), p(c)+1\}$ for any $0 \leq c < q^n - 1$ and hence

$$0 \leq p(a) - p(b) \leq a - b.$$

Therefore, if $a_l = b_l$ for all $l \geq i$, then $a - b < q^{i-1}$, which yields $p(a) - p(b) < q^{i-1}$ and hence these two words agree on positions from i to n .

Let $f = f^n \circ \dots \circ f^1$, 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 $g_m^i = g_m = (f \circ p)_m = f_m \circ p$. Therefore, $g^i(a) = g^i(b)$ if and only if $f_m(p(a)) = f_m(p(b))$ for all $m \leq i$ and $a_l = b_l$ for all $l \geq i + 1$. By the claim above, we obtain $p(a)_l = p(b)_l$ for all $l \geq i + 1$. Thus $g^i(a) = g^i(b)$ implies $h(p(a)) = h(p(b))$, where

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

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

THEOREM 3.13. *Any transformation of A^n can be computed in at most $4n - 3$ instructions.*

Proof. Let f be a transformation of A^n and consider the integer partition λ induced by its pre-images: denote $f(A^n) = \{a_1, \dots, a_k\}$ and $\lambda_i = |f^{-1}(a_i)|$ for all $1 \leq i \leq k$. Then f can be expressed as

$$f = h \circ p^\lambda \circ g,$$

where g and h are permutations of A^n satisfying

$$g(f^{-1}(a_j)) = \left[\sum_{i=1}^{j-1} \lambda_i, \sum_{i=1}^j \lambda_i - 1 \right],$$

$$h(j-1) = a_j$$

for all $1 \leq j \leq k$.

By Theorem 3.5, g and h can be computed as follows, where the superscript indicates which coordinate is updated by each instruction:

$$g = \bar{g}^1 \circ \dots \circ \bar{g}^{n-1} \circ g^n \circ \dots \circ g^1,$$

$$h = \bar{h}^1 \circ \dots \circ \bar{h}^{n-1} \circ h^n \circ \dots \circ h^1.$$

By Proposition 3.12, the transformation $h^n \circ \dots \circ h^1 \circ p^\lambda$ can be computed in n instructions $p^n \circ \dots \circ p^1$. Furthermore, p_1 and \bar{g}^1 being instructions updating y_1 , their product $q^1 = p^1 \circ \bar{g}^1$ is another instruction updating y_1 . Thus, f can be computed by the following program of length $4n - 3$:

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

\square

We conclude this section with a remark on infinite alphabets. If A is infinite, there exists a bijection $h : A^n \rightarrow A$ and thus any transformation can be computed in $n + 1$ instructions by the following program:

$$y_n \leftarrow h(y)$$

$$y_1 \leftarrow f_1(x)$$

$$\vdots$$

$$y_n \leftarrow f_n(x).$$

4. Computing linear transformations.

4.1. 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 $A = \text{GF}(q)$, and we want to compute a linear transformation f of A^n , i.e.

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

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

This 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. Theorem 4.1 shows that all matrices can be computed in a linear number of instructions.

THEOREM 4.1. *Any $n \times n$ nonsingular matrix M can be computed by at most $2n - 1$ linear instructions. Furthermore, this can be done in two main steps:*

- *The first step updates row i for i from 1 to $n - 1$ to produce an upper unitriangular matrix (i.e., with ones on the diagonal)*
- *The second step updates row i for i from n down to 1 to produce M .*

In general, any $n \times n$ matrix with rank $\rho \geq 1$ can be computed in $n + \rho - 1$ linear instructions.

Proof. The proof of correctness of the algorithm for nonsingular matrices actually goes in reverse: we start from M and construct the identity matrix. We first justify the first step of the reversed algorithm: M can be triangularised in n instructions. Let us prove that after k instructions we can obtain a matrix M_k where the upper left $k \times k$ submatrix is upper unitriangular, by induction on k ($1 \leq k \leq n$). We shall consider the $(n - 1) \times k$ matrix N formed by the first k columns and all but the k -th row of M_{k-1} ($M_0 = M$). For $k = 1$, we need to consider two cases:

1. The $(1, 1)$ entry of M is nonzero, then scaling the first row will work:

$$y_1 \leftarrow M(1, 1)^{-1}y_1.$$

2. Otherwise, there exists a non-zero element $M(j, 1)$, then do

$$y_1 \leftarrow y_1 + M(j, 1)^{-1}y_j.$$

Now assume the claim holds for $k - 1$. Once again, we distinguish two cases on N :

1. The unit vector $\bar{e}^k = (0, \dots, 0, 1) \in A^k$ is not in the row span of N . Then simply replace row y_k with $e^k \in A^n$.
2. Otherwise, by hypothesis the $k \times k$ matrix whose rows are given by the first $k - 1$ rows of N together with \bar{e}^k is upper unitriangular. Therefore, N has full rank and there is a linear combination of rows vN satisfying

$$vN + (y_k)^k = \bar{e}^k,$$

where $(y_k)^k$ are the first k positions of y_k . Therefore, denoting all but the k -th row of M_{k-1} as N' , perform

$$y_k \leftarrow y_k + vN'.$$

We now prove the second step: any upper unitriangular matrix can be turned into the identity matrix in $n - 1$ instructions. Let us prove that we can obtain a matrix whose last k rows are equal to those of the identity matrix in $k - 1$ instructions ($1 \leq k \leq n$). For $k = 1$, this is trivial. Suppose it holds for $k - 1$ and denote $y_{n-k+1} = (0, \dots, 0, 1, a_{n-k+2}, \dots, a_n)$, then perform

$$y_{n-k+1} \leftarrow y_{n-k+1} - \sum_{i=n-k+2}^n a_i y_i.$$

We now consider matrices with rank $1 \leq \rho < n$. We prove the claim by induction on n , the claim being clear for $n = 1$. Assume it is true for up to $n - 1$ and let us compute the matrix $M \in A^{n \times n}$. Without loss of generality, let the first ρ rows of M be linearly independent and them as $(N|P) \in A^{\rho \times n}$, where N has ρ columns and P has $n - \rho$ columns.

By hypothesis, there is a program with length at most $2\rho - 1$ which can compute N . Suppose N has k rows equal to those of the identity matrix. Then it is easily shown that there exists a program which computes N in no more than $2(\rho - k) - 1$ instructions. This program can be appended by k trivial instructions $y_j \leftarrow y_j$ to obtain a program which computes N , which updates all rows, and which has no more than $2\rho - 1$ instructions in total.

We adapt this program so that it computes $(N|P)$ as follows. Suppose that $y_j \leftarrow f_j$ is the first final update therein. Then applying the program, it should yield the j -th row of N for the first ρ coordinates and the all-zero vector for the last $n - \rho$ coordinates. However, the j -th row of P , say v_j , can be expressed as a linear combination of the last $n - \rho$ rows of the $n \times n$ identity matrix. Therefore, simply replace $y_j \leftarrow f_j$ by

$$y_j \leftarrow f_j + v_j.$$

Subsequently, replace any occurrence of y_j by $y_j - v_j$ in the program. Do this operation for all rows, and we obtain a program which computes $(N|P)$ in at most $2\rho - 1$ instructions. Finally, all other $n - \rho$ rows of M can be expressed as linear combinations of the first ρ , so it only takes $n - \rho$ final updates. \square

4.2. Further results for nonsingular matrices. Let us characterise the set $\mathcal{M}(\text{GF}(q)^n)$ of invertible linear instructions. It is given by the set of nonsingular matrices with at most one nontrivial row:

$$\mathcal{M} = \{S(i, v) : 1 \leq i \leq n, v \in A^n(i)\},$$

where

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

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

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

$$G := \text{Cay}(\text{GL}(n, q), \mathcal{M}).$$

Our previous results imply that G is undirected and connected. Since it is a Cayley graph, it is vertex-transitive and in particular it is regular of valency $|\mathcal{M}| - 1 = n(q^n - q^{n-1} - 1)$. 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 the general linear group.

The diameter of G is of great interest as it gives the maximum procedural complexity $\mathcal{L}'(M)$ of computing a nonsingular matrix by updating one row at a time. We know that it is no more than $2n - 1$; we shall see that it is at least $\lfloor \frac{3n}{2} \rfloor$ (and hence it is equal to 3 when $n = 2$) but it remains unknown for $n \geq 3$. However, when the field A is large, then almost all $n \times n$ matrices can be computed in no more than n linear instructions.

PROPOSITION 4.2. *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 $\text{GF}(q)$ 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.

DEFINITION 4.3. *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.*

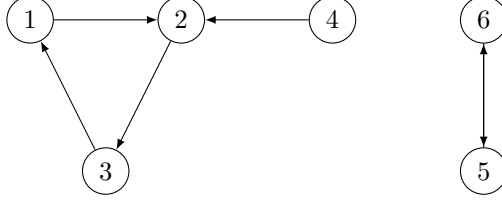


FIG. 4.1. Representing a transformation via a graph.

For instance, the identity matrix is the only scaled diagonal matrix. For any nonzero vector $v \in \text{GF}(q)^n$ with leading nonzero coordinate v_j , we can express

$$v = v_j v^*$$

for a unique scaled vector v^* . 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 3.8.

PROPOSITION 4.4. *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 :

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

4.3. Manipulating variables. We generalise the example of swapping two variables by considering any manipulation 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. Although we do not require that q should be a prime power, in such a case a manipulation of variables is indeed a linear transformation. Remark that $f^\phi \in \text{Sym}(A^n)$ if and only if $\phi \in \text{Sym}(n)$. We always use the postfix notation for ϕ , i.e. the image of i under ϕ is denoted as $i\phi$. For $\phi : [n] \rightarrow [n]$, ϕ^k does represent the k -th power of ϕ according to composition.

DEFINITION 4.5. *A manipulation of variables is a transformation f^ϕ of A^n such that there exists a transformation ϕ of $[n]$ for which*

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

for all $x \in A^n$.

The transformation ϕ can be represented using a directed graph on $[n]$ with n arcs $(i, i\phi)$ (see [11] 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 $0 \leq l \leq k-1$, 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 $0 \leq l \leq k-1$ 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 $\phi : [6] \rightarrow [6]$ 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 4.1.

Example. Let us first consider the case of a cyclic shift of three variables, i.e. $\pi = (1, 2, 3)$ and $f^\pi : A^3 \rightarrow A^3$ such that $f^\pi(x_1, x_2, x_3) = (x_2, x_3, x_1)$. This can be computed via linear combinations:

$$\begin{aligned} y_1 &\leftarrow y_1 + y_2 + y_3 & (= x_1 + x_2 + x_3) \\ y_3 &\leftarrow y_1 - y_2 - y_3 & (= x_1) \\ y_2 &\leftarrow y_1 - y_2 - y_3 & (= x_3) \\ y_1 &\leftarrow y_1 - y_2 - y_3 & (= x_2). \end{aligned}$$

However, it is impossible to perform this cyclic shift in four instructions by first updating y_1 and then updating y_2 instead of y_3 .

This is an example of the more general result below.

PROPOSITION 4.6. *Let $\kappa \in \text{Sym}(n)$ be a cyclic permutation, without loss $\kappa = (1, 2, \dots, n)$. Then the cyclic shift of n variables $f^\kappa : A^n \rightarrow A^n$ 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 $x_{n+1} = x_1, x_n, \dots, x_{n-i+1}$ 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

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

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, for all $1 \leq k \leq n$, the update of y_i must occur after that of y_{i+1} . Indeed, otherwise 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. The only order possible is hence y_1, y_n, \dots, y_1 . \square

We can then determine the procedural complexity of a manipulation of variables.

THEOREM 4.7. *Let $\phi : [n] \rightarrow [n]$ have F fixed points and D detached cycles. Then the procedural complexity of the manipulation of n variables $f^\phi : A^n \rightarrow A^n$ 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 4.6. 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. Then after the first such update $y_i \leftarrow x_{i\phi}$, 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 $i > j$ [1]. 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

$$y_{a_m} \leftarrow y_{a_m\phi^L}. \quad (4.1)$$

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 (4.1).

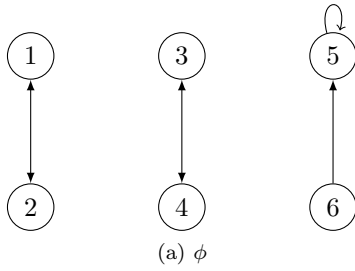
Conversely, it is clear that at least $n - F$ instructions are needed to compute f^ϕ . Furthermore, assume $D > 0$ and that there is a program computing f^ϕ in exactly $n - F$ instructions. Let i in the cycle c be the first coordinate belonging to a detached cycle to be updated. Then the program first 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. \square

Theorem 4.7 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 4.8. *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 4.7 is that when ϕ is not a permutation, we can obtain shorter programs by using some arithmetic than by adopting the ‘‘black box’’ approach used for the swap of two variables escribed in the very beginning of the paper. Figure 4.2 shows the smallest example: computing f^ϕ takes 6 instructions when using the program described in the proof of Theorem 4.7, while it takes 7 instructions when we do not combine variables. Clearly, this example can be generalized by adding more cycles, thus

(a) ϕ

With arithmetic		Without	
$y_6 \leftarrow y_1 + y_3$	$(= x_1 + x_3)$	$y_6 \leftarrow y_1$	$(= x_1)$
$y_1 \leftarrow y_2$	$(= x_2)$	$y_1 \leftarrow y_2$	$(= x_2)$
$y_2 \leftarrow y_6 - y_3$	$(= x_1)$	$y_2 \leftarrow y_6$	$(= x_1)$
$y_3 \leftarrow y_4$	$(= x_4)$	$y_6 \leftarrow y_3$	$(= x_3)$
$y_4 \leftarrow y_6 - y_2$	$(= x_3)$	$y_3 \leftarrow y_4$	$(= x_4)$
$y_6 \leftarrow y_5$	$(= x_5)$	$y_4 \leftarrow y_6$	$(= x_3)$
		$y_6 \leftarrow y_5$	$(= x_5)$

(b) Programs for f^ϕ

FIG. 4.2. The simplest manipulation of variables using a shorter program with arithmetic

yielding an arbitrarily large gap between the two approaches. The results are summarised in Proposition 4.9 and Corollary 4.10.

PROPOSITION 4.9. *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 instructions of the form $y_i \leftarrow y_j$ for any $i, j \in [n]$ if and only if ϕ is not a permutation (or is the identity). In that case, the shortest length of such a program is $n - F + D$.*

The proof calls arguments similar to those used above and is hence omitted.

COROLLARY 4.10. *If ϕ is not a permutation, then the ratio between the procedural complexity of f^ϕ over the minimum length of a program computing f^ϕ using instructions of the form $y_i \leftarrow y_j$ is always greater than $2/3$. Conversely, for any $\epsilon > 0$, there exists ϕ for which that ratio is between $2/3$ and $2/3 + \epsilon$.*

5. Using memory. Suppose we want to compute a transformation f of A^n using m memory cells storing values in A . By convention, we shall denote the content of the m memory cells as y_{n+1}, \dots, y_{n+m} . Then computing f using m memory cells is equivalent to computing some transformation $h(x_1, \dots, x_{n+m})$ 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 5.1 shows that our framework also considers the case of using memory. Indeed, 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.

PROPOSITION 5.1. *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 $h_{n+i}^e = e_i$ for $1 \leq i \leq m$. 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, $\mathcal{L}(h^e) \leq \mathcal{L}(f|m) + m$.

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 $j \leq n+k-1$) can be replaced by $y_j \leftarrow g'(y_1, \dots, y_{n+k-1})$ where $g' : A^{n+k-1} \rightarrow A$ 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 which computes some transformation in $D(f, m)$ and whose length is given by $\mathcal{L}(h^e) - m$. Thus $\mathcal{L}(f|m) \leq \mathcal{L}(h^e) - m$. \square

5.1. Shorter programs. We have shown in Theorem 2.4 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. In order to clarify notations, whenever $m = 1$, we denote the content of the memory cell as t .

We have shown in Theorem 3.5 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 3.2 and 5.2 show that these two results are simultaneously tight: there exists a permutation $f \in \text{Sym}(A^n)$ for which $\mathcal{L}(f) = 2n - 1$ while $\mathcal{L}(f|1) = n + 1$.

PROPOSITION 5.2. *The transposition (a, b) of two words $a, b \in A^n$ 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 us assume that a and b disagree on their first d coordinates. Then the following program computes (a, b) :

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

□

In Theorem 3.13, 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 5.3. *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 t_1, \dots, t_{n-1} and $2n - 1$ instructions:

$$\begin{aligned} t_1 &\leftarrow y_1 \\ &\vdots \\ t_{n-1} &\leftarrow y_{n-1} \\ y_1 &\leftarrow f_1(t_1, \dots, t_{n-1}, y_n) \\ &\vdots \\ y_n &\leftarrow f_n(t_1, \dots, t_{n-1}, y_n). \end{aligned}$$

□

Proposition 5.3 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.

This can be easily generalised to the case where f only has k nontrivial coordinate functions. In that case, using $k - 1$ memory cells yields a program of length at most $2k - 1$ instructions, and hence only $k - 1$ memory cells are needed.

The ideas behind Theorem 3.5 can be adapted to the case of using memory to yield a refinement of Proposition 5.3 for permutations.

THEOREM 5.4. *Any permutation of A^n can be computed in at most $3m$ instructions with m memory cells if $n = 2m$ is even and at most $3m + 3$ instructions with $m + 2$ memory cells if $n = 2m + 1$ is odd.*

Proof. Suppose $n = 2m$, let $f \in \text{Sym}(A^n)$ and let t_1, \dots, t_m denote the memory. By Proposition 3.4, there exist m functions $g_1, \dots, g_m : A^n \rightarrow A$ such that

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

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

- Step 1 (m instructions). For i from 1 to m , do $t_i \leftarrow g_i(x)$.
- 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 $(y_{m+1}, \dots, y_n, t_1, \dots, t_m)$.
- 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 $(y_1, \dots, y_m, t_1, \dots, t_m)$.

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. \square

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 5.4.

PROPOSITION 5.5. *Any linear permutation of A^n can be computed in at most $3m$ linear instructions with m memory cells if $n = 2m$ is even and at most $3m + 3$ linear instructions with $m + 2$ memory cells if $n = 2m + 1$ is odd.*

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 $\text{GF}(q)^{2m}$, there exists a third subspace in the same Grassmannian at subspace distance $2m$ from both [14] (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 [10], we only have to check for the row space of J and one subspace at distance $2d$ for each $0 \leq d \leq m$. 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

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

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

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

\square

For manipulations of variables, we can completely determine the gain offered by using memory.

Example. Let $\pi = (1, 2)(3, 4) \in \text{Sym}(4)$ and let $f^\pi : A^4 \rightarrow A^4$ be the corresponding permutation of variables. By Corollary 4.8, two disjoint transpositions of variables must be computed in at least 6 instructions when no memory is used. However, adjoining one memory cell t leads to a program with only 5 instructions, as seen below.

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

More generally, we can show that using only one memory cell is sufficient to compute any manipulation of variables.

PROPOSITION 5.6. *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 4.7, 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 4.7, 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} . \square

By comparing with Theorem 4.7, we see that using only one memory cell reduces the length of the program from $n - F + C$ to $n - F + 1$ for permutations. In particular, for a disjoint product of m transpositions, the complexity goes down from $3m$ to only $2m + 1$.

5.2. Binary instructions. Since the number of instructions is very large, 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 5.7. *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 5.8. *If $A = \text{GF}(2)$, then the set of all permutations of A^n which can be computed using binary instructions is the affine group $\text{Aff}(n, 2)$. 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 $A = \text{GF}(2)$ and $n = 2$, then it is well known that $\text{Sym}(\text{GF}(2)^2) = \text{Aff}(2, 2)$. If $n > 2$, then any instruction of the form $y_i \leftarrow g(y_i, y_j)$ must correspond to a binary instruction for $\text{GF}(2)^2$ 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 the memory cell t is used, we claim that the instructions in Theorem 2.4 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:

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

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

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

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

6. Acknowledgment. The authors would like to thank Peter J. Cameron, Ben Fairbairn, Peter Keevash and Rasmus Petersen for stimulating discussions.

REFERENCES

- [1] J.A. BONDY AND U.S.R. MURTY, *Graph Theory*, vol. 244 of Graduate Texts in Mathematics, Springer, 2008.

- [2] S. BURCKEL, *Closed iterative calculus*, Theoretical Computer Science, 158 (1996), pp. 371–378.
- [3] ———, *Elementary decompositions of arbitrary maps over finite sets*, Journal of Symbolic Computation, 37 (2004), pp. 305–310.
- [4] S. BURCKEL, E. GIOAN, AND E. THOMÉ, *Mapping computation with no memory*, in Proc. International Conference on Unconventional Computation, Ponta Delgada, Portugal, 2009, pp. 85–97.
- [5] S. BURCKEL AND M. MORILLON, *Three generators for minimal writing-space computations*, Theoretical Informatics and Applications, 34 (2000), pp. 131–138.
- [6] ———, *Quadratic sequential computations of boolean mappings*, Theory of Computing Systems, 37 (2004), pp. 519–525.
- [7] ———, *Sequential computation of linear boolean mappings*, Theoretical Computer Science, 314 (2004), pp. 287–292.
- [8] P.J. CAMERON, *Combinatorics: Topics, Techniques, Algorithms*, Cambridge University Press, Cambridge, 1994.
- [9] P.J. CAMERON, M. GADOLEAU, AND S. RIIIS, *Combinatorial representations*, submitted, (2011). Available at <http://arxiv.org/abs/1109.1216>.
- [10] P. DELSARTE, *Association schemes and t -designs in regular semilattices*, Journal of Combinatorial Theory A, 20 (1976), pp. 230–243.
- [11] O. GANYUSHKIN AND V. MAZORCHUK, *Classical Finite Transformation Semigroups: An Introduction*, vol. 9 of Algebra and Applications, Springer-Verlag, London, 2009.
- [12] C.D. GODSIL AND G. ROYLE, *Algebraic Graph Theory*, vol. 207 of Graduate Texts in Mathematics, Springer-Verlag, 2001.
- [13] D.-J. GUAN, *Generalized Gray codes with applications*, Proc. Natl. Sci. Coun. ROC(A), 22 (1998), pp. 841–848.
- [14] R. KÖTTER AND F.R. KSCHISCHANG, *Coding for errors and erasures in random network coding*, IEEE Transactions on Information Theory, 54 (2008), pp. 3579–3591.