Birkbeck
UNIVERSITY OF LONDON

BIROn - Birkbeck Institutional Research Online

Cameron, P.J. and Fairbairn, Ben and Gadoleau, M. (2014) Computing in permutation groups without memory. Technical Report. Birkbeck, University of London, London, UK.

Downloaded from: https://eprints.bbk.ac.uk/id/eprint/26716/

Usage Guidelines:
Please refer to usage guidelines at https://eprints.bbk.ac.uk/policies.html or alternatively contact lib-eprints@bbk.ac.uk.

Computing in Permutation Groups Without Memory

By

Peter J. Cameron, Ben Fairbairn & Maximilien Gadouleau

Birkbeck Pure Mathematics Preprint Series Preprint Number #1

www.bbk.ac.uk/ems/research/pure/preprints

Computing in Permutation Groups Without Memory

Peter J. Cameron, Ben Fairbairn & Maximilien Gadouleau

27 August 2014

Abstract

Memoryless computation is a modern technique to compute any function of a set of registers by updating one register at a time while using no memory. Its aim is to emulate how computations are performed in modern cores, since they typically involve updates of single registers. The memoryless computation model can be fully expressed in terms of transformation semigroups, or in the case of bijective functions, permutation groups. In this paper, we consider how efficiently permutations can be computed without memory. We determine the minimum number of basic updates required to compute any permutation, or any even permutation. The small number of required instructions shows that very small instruction sets could be encoded on cores to perform memoryless computation. We then start looking at a possible compromise between the size of the instruction set and the length of the resulting programs. We consider updates only involving a limited number of registers. In particular, we show that binary instructions are not enough to compute all permutations without memory when the alphabet size is even. These results, though expressed as properties of special generating sets of the symmetric or alternating groups, provide guidelines on the implementation of memoryless computation.

Keywords: memoryless computation, permutation groups, symmetric group, alternating group, generating sets, Boolean networks, sequential updates

1 Introduction

1.1 Memoryless computation

Typically, swapping the contents of two variables x and y requires a buffer t, and proceeds as follows (using pseudo-code):

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

However, a famous programming trick consists in using XOR (when x and y are sequences of bits), which we view in general as addition over a 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.

While the example described above (commonly referred to as the XOR swap) is folklore in Computer Science, the idea to compute functions without memory was developed by Burckel et. al. in [2, 3, 6, 7, 8, 4, 5], surveyed in [5], and expanded by Gadouleau and Riis in [12]. Amongst the results derived in the literature is the non-trivial fact that any function of n-bit input and n-bit ouput can be computed using memoryless computation. Moreover, only a number of updates linear in the number of registers is needed: any function of n variables can be computed in at most $4n-3$ updates [5, 12], and only $2n-1$ updates if the function is bijective [4].

Memoryless computation is a new topic, which is arguably not well known so far. It significantly differs from the many branches of "traditional" theoretical computer science, its peculiar setup yielding new phenomena, for instance the fact that binary instructions cannot always be used to compute a given function (shown for Boolean functions in [12] and developed in Theorem 4.1 of this paper). In fact, memoryless computation can be best studied via algebraic methods, especially as it can be completely recast in terms of permutation groups (when computing bijective functions) or transformation semigroups (in the general case).

It is this constant interplay between algebra and computer science that makes memoryless computation uniquely interesting. As such, the questions tackled in these papers both come from a computer science point of view and a mathematical point of view. Some questions are more natural with possible applications of memoryless computation in mind (for instance, the use of l-ary instructions), while others are natural from an algebraic point of view (such as the study of the alternating group).

Memoryless computation has the potential to speed up computations in two ways. First, it avoids time-consuming communication with the memory, thus easing concurrent execution of different programs. Second, unlike traditional computing which treats registers as "black boxes," memoryless computation effectively combines the values contained in those registers. Therefore, memoryless computation can be viewed as an analogue in computing to network coding [1, 23], an alternative to routing on networks. It is then proved in [12] that memoryless computation uses arbitrarily fewer updates than black-box computing for a certain class of manipulations of registers.

1.2 Model for computing in permutation groups without memory

Let us recall some notation and results from the literature in memoryless computation. Let A be a finite set, referred to as the *alphabet*, of cardinality $q := |A| \geq 2$ (usually, we assume $A = \mathbb{Z}_q$ or $A = \mathrm{GF}(q)$ if q is a prime power). Let $n \geq 2$ be an integer representing the number of registers $x_1, \ldots, x_n$. We denote $[n] = \{1, 2, \ldots, n\}$. The elements of A^n are referred to as *states*, and any state $a \in A^n$ is expressed as $a = (a_1, \ldots, a_n)$, where a_i is the i-th coordinate or register of a. For any $1 \leq k \leq n$, the k-th unit state is given by $e^k = (0, \ldots, 0, 1, 0, \ldots, 0)$ where the 1 appears in coordinate k. We also denote the all-zero state as e^0.

Although the model in [12] considered the computation of any transformation of A^n, in this paper we only consider permutations of A^n. For any $f \in \mathrm{Sym}(A^n)$, we denote its n coordinate functions as $f_1, \ldots, f_n : A^n \to A$, i.e. $f(x) = (f_1(x), \ldots, f_n(x))$ for all $x = (x_1, \ldots, x_n) \in A^n$. We say that the i-th coordinate function is *trivial* if it coincides with that of the identity: $f_i(x) = x_i$; it is nontrivial otherwise.

An *instruction* is a permutation g of A^n with exactly one nontrivial coordinate function:

$$g(x) = (x_1, \ldots, x_{j-1}, g_j(x), x_{j+1}, \ldots, x_n).$$

We say the instruction g *updates* the j-th register. We can represent this instruction as

$$y_j \leftarrow g_j(y)$$

where $y = (y_1, \ldots, y_n) \in A^n$ represents the contents of the registers. By convention, we also let the identity be an instruction, but we shall usually omit it. We denote the set of all instructions in $\mathrm{Sym}(A^n)$ as $\mathcal{I}(A^n)$ (or simply $\mathcal{I}$ when there is no ambiguity). For instance, $\mathcal{I}(\mathrm{GF}(2)^2)$ is given by

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

A *program* computing $f \in \mathrm{Sym}(A^n)$ is a sequence $g^{(1)}, \ldots, g^{(L)} \in \mathcal{I}$ such that

$$f = g^{(L)} \circ \cdots \circ g^{(1)}.$$

In other words, a program computes f by updating one register at a time, without any knowledge of the input. We remark that unless f is the identity, we can assume that none of the instructions in its program is the identity; furthermore, we can always assume that $g^{(k+1)}$ updates a different register to $g^{(k)}$. The shortest length of a program computing f is denoted as $\mathcal{L}(f)$ and referred to as the *complexity* of f.

With this notation, the swap of two variables can be viewed as computing the permutation f of A^2 defined as $f(x_1, x_2) = (x_2, x_1)$, and the program is given by

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

Thus, the complexity of the swap is three instructions.

Burckel [2] indicates that the instructions generate the symmetric group $\mathrm{Sym}(A^n)$: any permutation can be computed without memory (see [12] for a concise proof). Once this is established, the first natural problem is to determine how fast permutations can be computed. Theorem 2 in [12] shows that the maximum complexity of any permutation of A^n is exactly $2n - 1$ instructions. Moreover, the average complexity is above $2n - 3$ when the alphabet is binary and n is large enough (see [12]).

1.3 Smaller instruction sets

In this paper, we investigate another natural problem for memoryless computation. The model introduced above allows us to update a register by any possible function of all the registers. In practice the very large number of possible instructions makes it hard to encode all of them on a core. Therefore, we must search for limited instruction sets which are easy to encode while still salvaging the advantages offered by memoryless computation. Before answering

this engineering problem, we will determine its theoretical limit, i.e. the size of the smallest instruction set which allows us to compute any function without memory.

Once recast in algebraic language, the design of instruction sets able to compute any transformation becomes a problem on generating sets, which has been extensively studied for transformation semigroups [15, 16]. It is well known that to generate the full transformation semigroup, one only needs a generating set of the symmetric group and one more transformation [13]. Since the last transformation can be an instruction [12], we only consider the symmetric group in this paper. In Theorem 3.1 (a), we prove that $\mathrm{Sym}(A^n)$ can be generated by only n instructions (unless $q = n = 2$, where three instructions are needed); we also prove a similar result for the alternating group in Theorem 3.1 (b). This result illustrates that memoryless computation could be practically implemented on cores.

An "efficient" instruction set must satisfy the following tradeoff: it should contain a relatively small number of instructions and yet yield short programs. Also, the XOR swap is seldom used in practice, mostly because the instructions it involves cannot be easily pipelined. We thus expect a good set of instructions to offer the possibility to easily pipeline instructions. We will then investigate a natural candidate for a possible generating set of instructions. We will try to compute functions using only "local" instructions, which only involve a limited number of registers, i.e. *l-ary* instructions.

Definition 1.1. *A coordinate function $f_j : A^n \to A$ is l-ary if it only involves at most l variables:*

$$f_j(x) = f_j(x_{k_1}, \ldots, x_{k_l})$$

for some $k_1, \ldots, k_l \in [n]$. For $l = 1, 2$, we say it is unary, binary respectively. A permutation whose coordinate functions are all l-ary is also referred to as l-ary.

Binary Boolean instructions are not sufficient to compute any Boolean function [12]. In Theorem 4.1, we settle the general case. In particular, an n-ary instruction is required to generate $\mathrm{Sym}(A^n)$ when $|A|$ is even.

Another type of search for a trade-off between small instruction sets and short programs is investigated in [10], where we compute linear functions only using linear updates. Analogous results to Theorem 3.1 and to the maximum complexity in [12] are determined for the general and special linear groups.

The rest of the paper is organised as follows. Section 2 gives some preliminary definitions and results. Among others, it proves that any even permutation can be computed by even instructions in most cases. Section 3 determines the minimum size of a generating set of instructions for the symmetric and alternating groups. Then, in Section 4 we determine the groups generated by l-ary instructions. Finally, Section 5 gives some open questions and perspectives on the topic.

2 Preliminaries

2.1 Action on the set of instructions by conjugation

We remark that the set of l-ary permutations forms a group if and only if $l \in \{1, n\}$. Indeed, for $l = 1$ the unary permutations form the group

$$U := \mathrm{Sym}(A) \text{ Wr } \mathrm{Sym}(n),$$

where Wr denotes the wreath product (see [9]), and for $l = n$ they form $\mathrm{Sym}(A^n)$. Conversely, if $2 \leq l \leq n-1$, then it is clear that the following program uses l-ary instructions and yet does not compute an l-ary permutation:

$$y_2 \leftarrow y_2 + y_{l+1}$$
$$y_1 \leftarrow \sum_{i=1}^{l} y_i.$$

A *permutation of variables* is a permutation f^π of A^n such that $f^\pi(x) = (x_{1\pi}, \ldots, x_{n\pi})$ for some $\pi \in \mathrm{Sym}(n)$; note that $(f^\pi)^{-1} = f^{\pi^{-1}}$. In other words, the permutations of variables represent an action of $\mathrm{Sym}(n)$ on A^n.

Proposition 2.1. The largest group acting by conjugation on the set of instructions $\mathcal{I}$ is the group U of unary permutations.

Proof. First, let us show that U acts on $\mathcal{I}$ by conjugation. Note that U is generated by the permutations of variables and the unary instructions. Let $h \in \mathrm{Sym}(A^n)$ be a unary instruction with nontrivial coordinate function $h_i(x_i)$, where $h_i \in \mathrm{Sym}(A)$; then $h^{-1}(x) = (x_1, \ldots, h_i^{-1}(x_i), \ldots, x_n)$. Let g be an instruction updating register j, then

$$h^{-1}gh(x) = \begin{cases} (x_1, \ldots, g_j(h(x)), \ldots, x_n) & \text{if } i \neq j, \\ (x_1, \ldots, h_i^{-1}(g_i(h(x))), \ldots, x_n) & \text{otherwise.} \end{cases}$$

In both cases, it is an instruction updating register j.

Let f^π be a permutation of variables, then it is easily checked that

$$f^{\pi^{-1}}gf^\pi(x) = (x_1, \ldots, g_j(f^\pi(x)), \ldots, x_n),$$

where the nontrivial term appears in coordinate $j\pi^{-1}$.

Second, let us prove that any non-unary permutation does not act on $\mathcal{I}$ by conjugation. Let $f \in \mathrm{Sym}(A^n)$ have a non-unary coordinate function (and hence f^{-1} also), then there is an x_i which appears more than once in the coordinate functions of f^{-1}, say in $f_1^{-1}(x_i, \ldots)$ and $f_2^{-1}(x_i, \ldots)$. Then there exist two pairs of states a^k, b^k $(k = 1, 2)$ in A^n only differing in register i such that $f_k^{-1}(a^k) \neq f_k^{-1}(b^k)$. Let g be an instruction updating the i-th register such that $g(a^1) = b^1, g(b^2) = a^2$. Then denoting $u = f^{-1}(a^1)$ and $v = f^{-1}(b^2)$, we obtain

$$\begin{aligned} f_1^{-1}gf(u) &= f_1^{-1}g(a^1) = f_1^{-1}(b^1) \neq u_1 \\ f_2^{-1}gf(v) &= f_2^{-1}g(b^2) = f_2^{-1}(a^2) \neq v_2. \end{aligned}$$

Therefore, $f^{-1}gf$ updates the first and second registers and as such is not an instruction. □

We remark that if g, g' are U-conjugates (i.e. $g = hgh^{-1}$ for some $h \in U$), then $\mathcal{L}(g) = \mathcal{L}(g')$.

2.2 Internally computable permutation groups

Let $G \leq \mathrm{Sym}(A^n)$ and $g \in G$. We say that g is *computable* in G if there exists a program computing g consisting only of instructions from G. The set of elements computable in G is hence given by the subgroup $\langle G \cap \mathcal{I} \rangle$. We say G is *internally computable* if all its elements are computable therein, i.e. if $G = \langle G \cap \mathcal{I} \rangle$.

In order to illustrate how subtle the question of internal computability is, let us consider cyclic permutation groups. If g is an instruction, then clearly $\langle g \rangle$ is an internally computable group. On the other hand, consider $\mathrm{Sym}(\{0,1,2\}^2)$; for convenience number the elements of $\{0,1,2\}^2$ in Gray code as $1 := 00$, $2 := 01$, $3 := 02$, $4 := 12$, $5 := 11$, $6 := 10$, $7 := 20$, $8 := 21$, $9 := 22$ and let $g := (123)(67)$. The only generating sets of size one for the group $\langle g \rangle$ are g and g^{-1} and clearly neither of these is an instruction. Simultaneously, the elements g^2 and g^3 are instructions and since $g = g^3(g^2)^{-1}$ it follows that $\langle g \rangle = \langle g^2, g^3 \rangle$ so the group $\langle g \rangle$ is indeed internally computable, despite the fact that g is not an instruction.

The problem of determining whether a permutation group is internally computable can reveal some surprises. For instance, the alternating group $\mathrm{Alt}(\mathrm{GF}(2)^2)$ is not internally computable. Indeed, recall that the set of instructions is given by

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

Only the instructions $(x_1 + 1, x_2)$ and $(x_1, x_2 + 1)$ are even; those instructions only generate a group of order 4.

Proposition 2.2. The alternating group $\mathrm{Alt}(A^n)$ is internally computable unless $q = 2$ and $n = 2$ or $n = 3$.

Proof. The proof for $q = 2$ will follow Theorem 4.1 (the case for $q = 2$, $n = 3$ is checked by computer). For $q \geq 3$, let G be the group generated by even instructions. Define a relation $\sim_1$ on A^n by $x \sim_1 y$ if $x = y$ or there exists z such that the 3-cycle (x, y, z) is in G. This is an equivalence relation, and it has the property that, if $x \sim_1 y$ and $y \sim_1 z$, then $(x, y, z) \in G$. So if $\sim_1$ is the universal relation, then G contains every 3-cycle, and is the alternating group.

Let x, y differ in one position, i.e. $y = x + \lambda e^i$ for some $\lambda \neq 0$ and some $1 \leq i \leq n$. Then for any $\mu \notin \{0, \lambda\}$, (x, y, z) is an even instruction, where $z = x + \mu e^i$. Thus any two states are equivalent when they differ in one register, and hence $\sim_1$ is the universal relation. □

Given an internally computable group G, one may ask two "extreme" questions, similar to the problems for the symmetric group.

1. What is the maximum complexity of an element in G, i.e. the diameter of the Cayley graph $\mathrm{Cay}(G, G \cap \mathcal{I})$? This indicates how fast we can compute any element in G if we allow any instruction.

2. What is the smallest cardinality of a set of instructions in G which generates G? This indicates the minimum amount of space required to store the instructions needed to compute any element of G.

2.3 Fast permutations

For any set of instructions $\mathcal{J} \subseteq \mathcal{I}$ and any $g \in \langle \mathcal{J} \rangle$, we denote the shortest length of a program computing g using only instructions from $\mathcal{J}$ as $\mathcal{L}(g, \mathcal{J})$. (Note that we only look at instructions in $\mathcal{J}$, and not all instructions in $\langle \mathcal{J} \rangle$). If $\mathcal{J} \subseteq \mathcal{K}$, we say g is $(\mathcal{J}, \mathcal{K})$-*fast* if $\mathcal{L}(g, \mathcal{J}) = \mathcal{L}(g, \mathcal{K})$.

We have the following properties: let $\mathcal{J} \subseteq \mathcal{K} \subseteq \mathcal{M}$ and $g, h \in \langle \mathcal{J} \rangle$.

1. If $\mathcal{J}$ is symmetric and g is $(\mathcal{J}, \mathcal{K})$-fast, then so is g^{-1}.
2. If g, h are $(\mathcal{J}, \mathcal{K})$-fast and $\mathcal{L}(gh, \mathcal{K}) = \mathcal{L}(g, \mathcal{K}) + \mathcal{L}(h, \mathcal{K})$, then gh is also $(\mathcal{J}, \mathcal{K})$-fast.
3. If g is $(\mathcal{J}, \mathcal{M})$-fast, then g is $(\mathcal{J}, \mathcal{K})$-fast.
4. If g is $(\mathcal{J}, \mathcal{K})$-fast and $(\mathcal{K}, \mathcal{M})$-fast, then g is $(\mathcal{J}, \mathcal{M})$-fast.

If all elements of an internally computable group K are $(K \cap \mathcal{I}, G \cap \mathcal{I})$-fast for some $K \leq G$, we say that K is fast in G. We simply say that K is fast if it is fast in $\mathrm{Sym}(A^n)$. For instance, if K is the group of all instructions updating a given register, then K is clearly fast. On the other hand, the alternating group is never fast.

Proposition 2.3. The alternating group $\mathrm{Alt}(A^n)$ is not fast for any A and n.

Proof. Let $g = (e^0, e^1, e^2) = (e^0, e^2) \circ (e^0, e^1)$ be an even permutation of A^n. Then $\mathcal{L}(g, \mathrm{Sym}(A^n)) = 2$ since both transpositions are instructions. On the other hand, any program computing g of length 2 must begin with either its first or second coordinate function g_1 or g_2, i.e.

$$\begin{aligned} \text{either} \quad y_1 \leftarrow g_1(y) &= y_1 + \delta(y, e^0) - \delta(y, e^1) \\ \text{or} \quad y_2 \leftarrow g_2(y) &= y_2 + \delta(y, e^1) - \delta(y, e^2), \end{aligned}$$

where δ is the Kronecker delta function. However, the first corresponding instruction is the transposition (e^0, e^1), while the second is not even a permutation, for it maps both e^2 and e^0 to e^0. □

The problem of determining whether a group G is fast has three important special cases.

1. $G = \mathrm{GL}(n, q)$ for q a prime number. This indicates whether one can compute linear functions any faster by allowing nonlinear instructions.

   Conjecture 2.4. $\mathrm{GL}(n, q)$ is fast.

   Two partial results are already known. First, Theorem 4 in [12] shows that the permutation matrices are fast in the general linear group: it takes exactly $n - F + C$ instructions to compute a permutation of variables with F fixed points and C cycles, and this can be done via linear instructions. Secondly, Proposition 4 in [12] shows that for large q, almost all of $\mathrm{GL}(n, q)$ can be computed in n linear instructions and hence almost all of the general linear group is fast.

2. $G = \mathrm{Sym}(A^k) \times \mathrm{Sym}(A^{n-k})$ acting coordinatewise. The significance of this group can be explained as follows. Suppose we want to compute a function of k registers only, but we have n registers available. The additional $n-k$ registers can then be used as additional memory. It is known that using additional memory can yield shorter programs in some cases [12]. However, all the shortest programs using memory known so far "erase" the memory content and replace it with functions of the first k registers. If G is fast, then one cannot compute the original function of k registers any faster without erasing some knowledge of the last $n-k$ registers. More formally, G is fast if and only if the following conjecture is true.

   Conjecture 2.5. Let $g \in \mathrm{Sym}(A^k)$, $h \in \mathrm{Sym}(A^{n-k})$ and define $f \in \mathrm{Sym}(A^n)$ by $(f_1(x), \ldots, f_k(x)) = g(x_1, \ldots, x_k)$ and $(f_{k+1}(x), \ldots, f_n(x)) = h(x_{k+1}, \ldots, x_n)$. Then
   $$\mathcal{L}(f) = \mathcal{L}(g) + \mathcal{L}(h).$$

3. $G = \mathrm{Sym}(A^k)$ acting on the first k coordinates. This might be a simpler case than the previous one.

3 Smallest sets of generating instructions

We now turn to the problem of determining the smallest number of instructions generating the symmetric group and the alternating group. Clearly, one needs to update all n registers to generate a transitive group.

Theorem 3.1. *1. Unless $q = n = 2$, $\mathrm{Sym}(A^n)$ is generated by n instructions.*

2. If $q \geq 3$, then $\mathrm{Alt}(A^n)$ is generated by n instructions.

We recall a classical theorem of Jordan (see for instance [11, Theorem 3.3E]).

Lemma 3.2. *Let $G \leq \mathrm{Sym}(m)$ be primitive and suppose that G contains a cycle of length p for some prime $p \leq m-3$. Then $G = \mathrm{Sym}(m)$ or $\mathrm{Alt}(m)$.*

We first deal with small alphabets.

Lemma 3.3. *The group $\mathrm{Alt}(\{0,1,2\}^n)$ is generated by n instructions.*

Proof. The case $n = 2$ is easily dealt with separately, so we shall assume that $n > 2$.

We order the elements of A^n lexicographically and number them accordingly (so for instance if $n = 3$ then the elements in order are 000=:1, 001=:2, 002=:3, 010=:4, 011=:5,...). We define the permutations $\pi_1, \ldots, \pi_n$ as follows. First $\pi_1 = (1,2,3)$, so π_i only updates the rightmost register. For $2 \leq i \leq n$ the permutation π_i is the unique instruction updating the i^{th} register that is a product of 3^{n-1} cycles of length 3 all but the last of which, when written in the usual cycle notation, lists states in lexicographic order. For example, if $n = 3$ these permutations are

$$\begin{aligned}
\pi_1 &= (1,2,3), \\
\pi_2 &= (1,4,7)(2,5,8)(3,6,9)(10,13,16)(11,14,17) \\
&\qquad (12,15,18)(19,22,25)(20,23,26)(27,24,21) \text{ and} \\
\pi_3 &= (1,10,19)(2,11,20)(3,12,21)(4,13,22)(5,14,23) \\
&\qquad (6,15,24)(7,16,25)(8,17,26)(27,18,9).
\end{aligned}$$

We claim that these permutations generate a 2-transitive group. It is easy to see that they generate a transitive group. We suppose $n \geq 4$ and proceed by induction, the case $n = 3$ be easily verified by computer. By hypothesis the group generated by $\pi_1, \ldots, \pi_{n-1}$ gives all permutations of the points the points $\{1, \ldots, q^{n-1}\}$. In particular we have the permutations $(i, i+1, i+2)$ for $1 \leq i \leq q^{n-1}-3$ and thus the permutations $(i, i+1, i+1)^{\pi_n^j}$ for $1 \leq i \leq q^{n-1}-4$ and $1 \leq j \leq q-1$. Furthermore, we also have the permutations $(1, 2, q^{n-1})^{\pi_n^j}$ for $1 \leq j \leq q-1$ which altogether gives us enough permutations to act transitively on the stabilizer of any point in $\{1, \ldots, q^{n-1}\}$.

We have shown that $\pi_1, \ldots, \pi_n$ generate a group that is 2-transitive and therefore primitive. Since π_1 is a 3-cycle we can now apply Jordan's Theorem to conclude that $\pi_1, \ldots, \pi_n$ generates the whole of $\mathrm{Alt}(A^n)$. □

Lemma 3.4. *The group* $\mathrm{Sym}(\{0,1\}^n)$ *is generated by* n *instructions.*

Proof. We define a set of permutations $\pi_1, \ldots, \pi_n$ as follows. We order words in $\{0,1\}^n$ in the usual Gray code ordering and label them $1, \ldots, 2^n$ (so for instance if $n = 3$ the elements in order are $000 =: 1$, $001 =: 2$, $011 =: 3$, $010 =: 4$, $110 =: 5$). We now define $\pi_1 := (1, 2)$. For $2 \leq i \leq n$ we construct π_i by taking the derangement that updates the i^{th} register and removing the first cycle that interchanges two adjacent words. For example, if $n = 3$ then

$$\begin{aligned}\pi_1 &= (1,2)\\ \pi_2 &= (1,4)(5,8)(6,7) \text{ and}\\ \pi_3 &= (1,8)(2,7)(3,6).\end{aligned}$$

A straightforward induction analogous to the proof of Lemma 3.3 now enables us to show that the above generate a primitive group containing a transposition and so we apply Jordan's Lemma to show that the above generate the whole of $\mathrm{Sym}(A^n)$. □

Lemma 3.5. *If* $q > 2$ *is odd then* $Sym(A^n)$ *is generated by* n *instructions.*

Proof. We define the permutation $\pi_1 : A^n \to A^n$ as follows. For $a := (a_1, a_2, \ldots, a_n)$ we have that

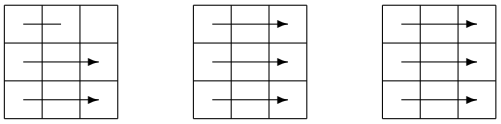
$$\pi_1 : a \mapsto \begin{cases} (1 - a_1, 0, \ldots, 0) & \text{if } a_1 \in \{0,1\} \text{ and } a_2 = \cdots = a_n = 0\\ a & \text{if } a_1 > 1 \text{ and } a_2 = a_3 = \cdots = a_n = 0\\ (0, a_2, \ldots, a_n) & \text{if } a_1 = q-1 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n\\ (a_1 + 1, a_2, \ldots, a_n) & \text{otherwise.}\end{cases}$$

For $2 \leq r \leq n$ we define the permutation $\pi_r : A^n \to A^n$ as follows.

$$\pi_r : a \mapsto \begin{cases} (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r\\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r \neq q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r\\ (a_1, \ldots, a_{r-1}, q-1, a_{r+1}, \ldots, a_n) & \text{if } a_r = 0 \text{ and } a_i = q-1 \text{ for all } i \neq r\\ (a_1, \ldots, a_{r-1}, a_r - 1, a_{r+1}, \ldots, a_n) & \text{otherwise.}\end{cases}$$

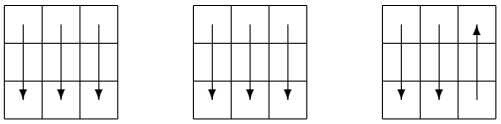
For example, if $q = 3$ and $n = 3$ then, writing the elements of A^n as a series of grids, we have that the permutation π_1 is

$a = (a_1, a_2, 0)$ $a = (a_1, a_2, 1)$ $a = (a_1, a_2, 2)$



whilst the permutation π_2 is

$a = (a_1, a_2, 0)$ $a = (a_1, a_2, 1)$ $a = (a_1, a_2, 2)$



and similarly for π_3.

We claim that $G := \langle \pi_1, \ldots, \pi_n \rangle$ is 2-transitive. It is easy to see that G acts transitively on A^n. We will show that the stabilizer of the state $(q-1, q-1, \ldots, q-1)$ is transitive on the remaining states.

Note that every cycle of π_1 apart from one has length q, which is odd, the remaining cycle being a transposition. It follows that $\tau := \pi_1^{q-1}$ is a transposition. It is easy to see that repeatedly conjugating τ by the various π_is we have enough elements for the stabilizer of $(q-1, q-1, \ldots, q-1)$ to act transitively on the remaining points, that is, G acts 2-transitively.

Since any 2-transitive action is primitive it follows that G acts primitively and since $\tau \in G$ is a transposition, Jordan's Theorem tells us that $G = \mathrm{Sym}(A^n)$. □

Lemma 3.6. *If $q > 2$ is even then Sym(A^n) is generated by n instructions.*

Proof. We define the permutation $\pi_1 : A^n \to A^n$ as follows. For $a := (a_1, a_2, \ldots, a_n)$ we have that

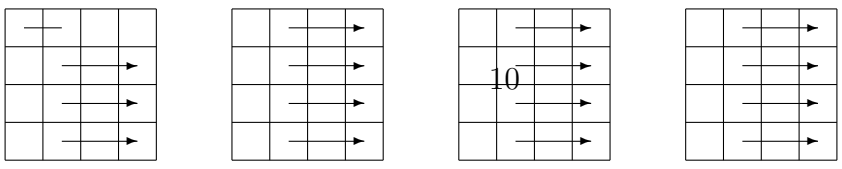
$$\pi_1 : a \mapsto \begin{cases} (1 - a_1, 0, \ldots, 0) & \text{if } a_1 \in \{0, 1\} \text{ and } a_2 = \cdots = a_n = 0 \\ a & \text{if } a_1 > 1 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ a & \text{if } a_1 = 0 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (1, a_2, \ldots, a_n) & \text{if } a_1 = q - 1 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (a_1 + 1, a_2, \ldots, a_n) & \text{otherwise.} \end{cases}$$

For $2 \leq r \leq n$ we define the permutation $\pi_r : A^n \to A^n$ as follows.

$$\pi_r : a \mapsto \begin{cases} (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r \neq q - 1 \text{ and } a_i \neq q - 1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r < q - 2 \text{ and } a_i = q - 1 \text{ for all } i \neq r \\ (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q - 1 \text{ and } a_i \neq q - 1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q - 2 \text{ and } a_i = q - 1 \text{ for all } i \neq r \\ a & \text{if } a_1 = a_2 = \ldots = a_n = q - 1 \end{cases}$$

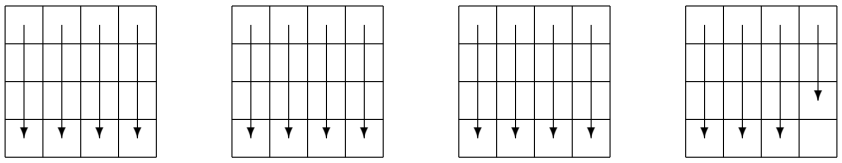
For example, if $q = 4$ and $n = 3$ then, writing the elements of A^n as a series of grids, we have that the permutation π_1 is

$a = (a_1, a_2, 0, 0)$ $a = (a_1, a_2, 1, *)$ $a = (a_1, a_2, 2, *)$ $a = (a_1, a_2, 3, *)$



where the star means that a_4 can take any value, whilst the permutation π_2 is

$a = (a_1, a_2, 0, *)$ $a = (a_1, a_2, 1, *)$ $a = (a_1, a_2, 2, *)$ $a = (a_1, a_2, 3, *)$



and similarly for π_3 and π_4.

The argument concludes in the same way as the previous lemma: these permutations generate a group that is 2-transitive and thus primitive and contains a transposition, so by Jordan's Theorem our result follows. □

We remark that $q = 2$ must naturally be handled separately since all cycles of every instruction in that case have length two.

More generally we ask the following.

Q 3.7. What do minimum generating sets of instructions look like?

As a partial answer to this question we have the following.

Lemma 3.8. *Let $X \subset Sym(A^n)$ be a set of n instructions which generates* $\mathrm{Sym}(A^n)$. *Then X does not contain a unary instruction.*

Proof. Let $\sim_r$ be the equivalence relation on A^n where $x \sim_r y$ if and only if $x_r = y_r$. Then any instruction updating any register other than r preserves $\sim_r$. Moreover, any unary instruction updating the register r also preserves $\sim_r$. Therefore, if X contains a unary instruction, it preserves $\sim_r$ for some r and hence cannot generate $\mathrm{Sym}(A^n)$. □

Comment Since we can order the states in A^n such that two consecutive states only differ in one register (it is called a (q, n)-Gray code [14]), the Coxeter generators according to that ordering are all instructions. Therefore, the maximum size of a minimal set of generating instructions (i.e., a generating set whose proper subsets are not generating) is exactly $q^n - 1$.

We proceed to discuss Theorem 3.1 (b). Essentially our argument is the same as the previous two lemmas replacing each of our transpositions with 3-cycles and resorting to Jordan's Theorem to deduce the final result. Unfortunately, in this case we need to split off into more cases than simply even and odd since we now need to keep track not only of the length of the cycles mod 3 but also of the parities of the permutations to ensure that the permutations we construct are in fact all even.

Lemma 3.9. *If $q \equiv 1$ or $5 \pmod 6$ and $q > 1$ then $Alt(A^n)$ is generated by n instructions.*

Proof. We define the permutation $\pi_1 : A^n \to A^n$ as follows. For $a := (a_1, a_2, \ldots, a_n)$ we have

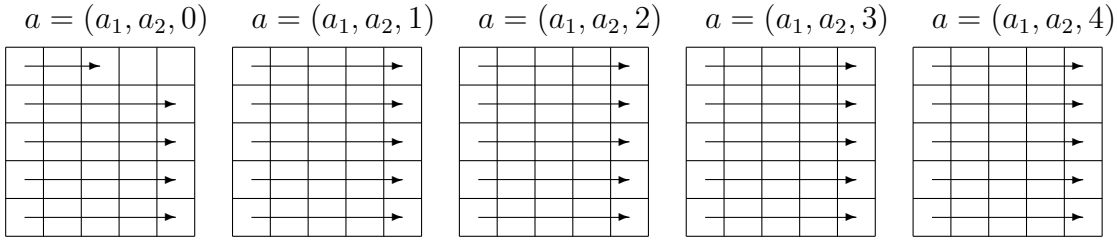
that

$$\pi_1 : a \mapsto \begin{cases} (1,0,\ldots,0) & \text{if } a_1 = a_2 = \cdots = a_n = 0 \\ (2,0,\ldots,0) & \text{if } a_1 = 1 \text{ and } a_2 = \cdots = a_n = 0 \\ (0,0,\ldots,0) & \text{if } a_1 = 2 \text{ and } a_2 = \cdots = a_n = 0 \\ a & \text{if } a_1 > 2 \text{ and } a_2 = \cdots = a_n = 0 \\ (0,a_2,\ldots,a_n) & \text{if } a_1 = q-1 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (a_1+1,a_2,\ldots,a_n) & \text{otherwise.} \end{cases}$$

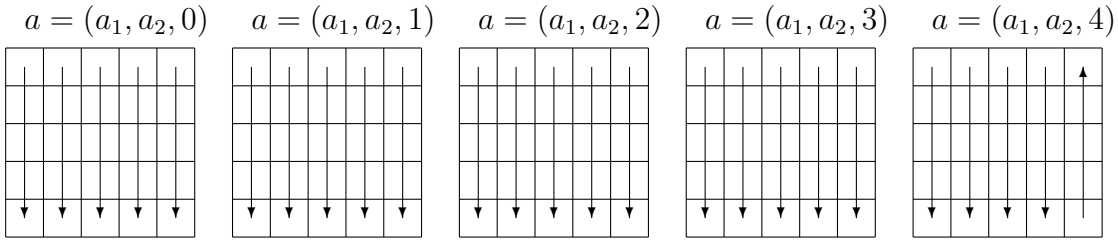
For $2 \leq r \leq n$ we define the permutation $\pi_r : A^n \to A^n$ as follows.

$$\pi_r : a \mapsto \begin{cases} (a_1,\ldots,a_{r-1},0,a_{r+1},\ldots,a_n) & \text{if } a_r = q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r \\ (a_1,\ldots,a_{r-1},a_r+1,a_{r+1},\ldots,a_n) & \text{if } a_r \neq q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r \\ (a_1,\ldots,a_{r-1},q-1,a_{r+1},\ldots,a_n) & \text{if } a_r = 0 \text{ and } a_i = q-1 \text{ for all } i \neq r \\ (a_1,\ldots,a_{r-1},a_r-1,a_{r+1},\ldots,a_n) & \text{otherwise.} \end{cases}$$

For example, if $q = 5$ and $n = 3$ then, writing the elements of A^n as a series of grids, we have that the permutation π_1 is



whilst the permutation π_2 is



and similarly for π_3.

Since all of these permutations are products of cycles of odd length they are all even permutations.

We claim that $G := \langle \pi_1, \ldots, \pi_n \rangle$ is 2-transitive. It is easy to see that G acts transitively on A^n. We will show that the stabilizer of the state $(q-1, q-1, \ldots, q-1)$ is transitive on the remaining states.

Note that all but one cycle of π_1 has length q, which is coprime to 3, the remaining cycle being a 3-cycle. It follows that $\tau := \pi_1^q$ is a 3-cycle. It is easy to see that repeatedly conjugating τ by the various π_is we have enough elements for the stabilizer of $(q-1, q-1, \ldots, q-1)$ to act transitively on the remaining points, that is, G acts 2-transitively.

Since any 2-transitive action is primitive it follows that G acts primitively and since $\tau \in G$ is a 3-cycle, Jordan's Theorem tells us that $G = \mathrm{Alt}(A^n)$. □

Lemma 3.10. *If $q \equiv 0$ or 2 (mod 6) and $q > 2$ then Alt(A^n) is generated by n instructions.*

Proof. We define the permutation $\pi_1 : A^n \to A^n$ as follows. For $a := (a_1, a_2, \ldots, a_n)$ we have that

$$\pi_1 : a \mapsto \begin{cases} (1, 0, \ldots, 0) & \text{if } a_1 = a_2 = \cdots = a_n = 0 \\ (2, 0, \ldots, 0) & \text{if } a_1 = 1 \text{ and } a_2 = \cdots = a_n = 0 \\ (0, 0, \ldots, 0) & \text{if } a_1 = 2 \text{ and } a_2 = \cdots = a_n = 0 \\ a & \text{if } a_1 > 2 \text{ and } a_2 = \cdots = a_n = 0 \\ a & \text{if } a_1 = 0 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (1, a_2, \ldots, a_n) & \text{if } a_1 = q-1 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (a_1 + 1, a_2, \ldots, a_n) & \text{otherwise.} \end{cases}$$

For $2 \leq r \leq n$ we define the permutation $\pi_r : A^n \to A^n$ as follows.

$$\pi_r : a \mapsto \begin{cases} (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r \neq q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r < q-3 \text{ and } a_i = q-1 \text{ for all } i \neq r \\ (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q-3 \text{ and } a_i = q-1 \text{ for all } i \neq r \\ a & \text{otherwise.} \end{cases}$$

By this stage we believe that the reader has seen sufficiently many examples and their corresponding diagrams for the reader to be able draw these themselves.

Since π_1 is a product of cycles of odd length it is an even permutation, whilst the permutations π_r for $2 \leq r \leq n$ are products of an even number of even cycles and are therefore also even.

The argument now concludes in the same way as Lemma 3.9. □

Lemma 3.11. *If $q \equiv 3 \pmod 6$ then $Alt(A^n)$ is generated by n instructions.*

Proof. We define the permutation $\pi_1 : A^n \to A^n$ as follows. For $a := (a_1, a_2, \ldots, a_n)$ we have that

$$\pi_1 : a \mapsto \begin{cases} (1, 0, \ldots, 0) & \text{if } a_1 = a_2 = \cdots = a_n = 0 \\ (2, 0, \ldots, 0) & \text{if } a_1 = 1 \text{ and } a_2 = \cdots = a_n = 0 \\ (0, 0, \ldots, 0) & \text{if } a_1 = 2 \text{ and } a_2 = \cdots = a_n = 0 \\ a & \text{if } a_1 > 2 \text{ and } a_2 = \cdots = a_n = 0 \\ a & \text{if } a_1 \in \{0, 1\} \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (2, a_2, \ldots, a_n) & \text{if } a_1 = q-1 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (a_1 + 1, a_2, \ldots, a_n) & \text{otherwise.} \end{cases}$$

For $2 \leq r \leq n$ we define the permutation $\pi_r : A^n \to A^n$ as follows.

$$\pi_r : a \mapsto \begin{cases} (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r \neq q-1 \text{ and } a_i \neq q-1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, q-1, a_{r+1}, \ldots, a_n) & \text{if } a_r = 0 \text{ and } a_i = q-1 \text{ for all } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r - 1, a_{r+1}, \ldots, a_n) & \text{otherwise.} \end{cases}$$

Since all of these are products of cycles of odd length they are all even permutations.

The argument now concludes in the same way as Lemma 3.9. □

Lemma 3.12. *If $q = 4$ then $Alt(A^n)$ is generated by n instructions.*

Proof. We define the permutation $\pi_1 : A^n \to A^n$ as follows. For $a := (a_1, a_2, \ldots, a_n)$ we have that

$$\pi_1 : a \mapsto \begin{cases} (1, 0, \ldots, 0) & \text{if } a_1 = 0 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ (2, 0, \ldots, 0) & \text{if } a_1 = 1 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ (0, 0, \ldots, 0) & \text{if } a_1 = 2 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ a & \text{if } a_1 = 3 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ (4 - a_1, a_2, \ldots, a_n) & \text{otherwise.} \end{cases}$$

For $2 \leq r \leq n$ we define the permutation $\pi_r : A^n \to A^n$ as follows.

$$\pi_r : a \mapsto \begin{cases} (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = 3 \text{ and } a_i \neq 3 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r \neq 3 \text{ and } a_i \neq 3 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, 1 - a_r, a_{r+1}, \ldots, a_n) & \text{if } a_r \in \{0, 1\} \text{ and } a_i = 3 \text{ for all } i \neq r \\ a & \text{otherwise.} \end{cases}$$

Since π_1 is a product of a three cycle and an even number of disjoint transpositions it is an even permutation. For $2 \leq r \leq n$ we have that π_r is a product of an odd number of cycles of length 4 and a transposition, so they are also even permutations.

The argument now concludes in the same way as Lemma 3.9. □

Lemma 3.13. *If $q \equiv 4 \pmod 6$ and $q > 4$ then $Alt(A^n)$ is generated by n instructions.*

Proof. We define the permutation $\pi_1 : A^n \to A^n$ as follows. For $a := (a_1, a_2, \ldots, a_n)$ we have that

$$\pi_1 : a \mapsto \begin{cases} (1 - a_1, a_2, \ldots, a_n) & \text{if } a_1 \in \{0, 1\} \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ (4, 0, \ldots, 0) & \text{if } a_1 = 3 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ (5, 0, \ldots, 0) & \text{if } a_1 = 4 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ (3, 0, \ldots, 0) & \text{if } a_1 = 5 \text{ and } a_2 = a_3 = \cdots = a_n = 0 \\ a & \text{if } a_1 > 5 \text{ and } a_2 = \cdots = a_n = 0 \\ (0, a_2, \ldots, a_n) & \text{if } a_1 = q - 1 \text{ and } a_i \neq 0 \text{ for some } 2 \leq i \leq n \\ (a_1 + 1, a_2, \ldots, a_n) & \text{otherwise.} \end{cases}$$

For $2 \leq r \leq n$ we define the permutation $\pi_r : A^n \to A^n$ as follows.

$$\pi_r : a \mapsto \begin{cases} (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q - 1 \text{ and } a_i \neq q - 1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r \neq q - 1 \text{ and } a_i \neq q - 1 \text{ for some } i \neq r \\ (a_1, \ldots, a_{r-1}, q - 1, a_{r+1}, \ldots, a_n) & \text{if } a_r = 0 \text{ and } a_i = q - 1 \text{ for all } i \neq r \\ (a_1, \ldots, a_{r-1}, a_r + 1, a_{r+1}, \ldots, a_n) & \text{if } a_r < q - 3 \text{ and } a_i = q - 1 \text{ for all } i \neq r \\ (a_1, \ldots, a_{r-1}, 0, a_{r+1}, \ldots, a_n) & \text{if } a_r = q - 1 \text{ and } a_i = q - 3 \text{ for all } i \neq r \\ a & \text{otherwise.} \end{cases}$$

Since π_1 is a product of a cycle of length 3, an odd number of cycles of length q and a transposition it is an even permutation. For $2 \leq r \leq n$ we have that π_r is a product of even cycles and is therefore an even permutation.

The argument now concludes in the same way as Lemma 3.9. □

4 l-ary instructions

Theorem 6 in [12] shows that the set of binary instructions only generates the affine group if $A = \mathrm{GF}(2)$. We characterise in Theorem 4.1 below what l-ary instructions generate in general.

Theorem 4.1. *For all $2 \leq l \leq n$, let G_l be the group generated by all l-ary instructions. We have the following:*

- *For any n, $G_n = \mathrm{Sym}(A^n)$.*

- *If $q = 2$, $G_2 = \mathrm{Aff}(n, 2)$ for all n and $G_3 = G_{n-1} = \mathrm{Alt}(A^n)$ for $n \geq 4$.*
- *If $n \geq 3$ and $q \geq 3$ is odd, then $G_2 = G_{n-1} = \mathrm{Sym}(A^n)$.*
- *If $n \geq 3$ and $q \geq 4$ is even, then $G_2 = G_{n-1} = \mathrm{Alt}(A^n)$.*

Proof. Note that, any $(n-1)$-ary instruction has the property that all its cycle lengths have multiplicities which are multiples of q, since 1 variable has no effect. So if q is even, every instruction is an even permutation, and the group G_{n-1} is contained in the alternating group.

When $q = 2$, G_2 was settled in [12, Theorem 6] and it directly follows from [17] that $G_3 = \mathrm{Alt}(\mathrm{GF}(2)^n)$.

We now assume $q \geq 3$ and it suffices to prove the result for G_2.

We use the following principles. Let G be a permutation group. (a) Define a relation $\sim$ by $x \sim y$ if $x = y$ or the transposition (x, y) is in G. Then $\sim$ is an equivalence relation. If it is the universal relation, then G contains every transposition, and is the symmetric group. (b) Define a relation $\sim_1$ by $x \sim_1 y$ if $x = y$ or there exists z such that the 3-cycle (x, y, z) is in G. Again $\sim_1$ is an equivalence relation, and it has the property that, if $x \sim_1 y$ and $y \sim_1 z$, then $(x, y, z) \in G$. So if $\sim_1$ is the universal relation, then G contains every 3-cycle, and is the alternating group.

Consider $n = 2$. There are two kinds of instruction, those that update y_1 and those that update y_2. Thinking of the points being permuted as forming a square grid, instructions of the first time form a group which is the direct product of symmetric groups on the columns; instructions of the second type form a group which is the direct product of symmetric groups on the rows. So the relation $\sim$ is non-trivial, and any two points in the same row or column are equivalent. Thus $\sim$ is the universal relation, and G is the symmetric group.

Now suppose that $n = 3$ and $q \geq 3$. By the $n = 2$ case, we see that every permutation fixing the first coordinate of all triples is in G. In particular, the permutation transposing $(x, 1, 1)$ with $(x, 1, 2)$ for all x belongs to G. Similarly the permutation transposing $(1, y, 1)$ and $(1, y, 0)$ for all y is in G. (Here we use $q \geq 3$.) Now in these permutations, the cycles containing $(1, 1, 1)$ intersect; the other cycles are disjoint. So the commutator of the two permutations is a 3-cycle. Now applying the argument about 3-cycles shows that G is symmetric or alternating. If q is even, it is alternating; if q is odd, since we have a product of q transpositions (an odd permutation), G is symmetric.

Finally, assume $n \geq 4$ and $q \geq 5$. Argue as before but using 3-cycles rather than transpositions; the condition $q \geq 5$ allows us to have 3-cycles meeting in a single point, so their commutator is a 3-cycle.

For $q = 3$, by induction we get the symmetric group at the previous stage, and so we have transpositions, and can play the usual game.

For $q = 4$, the commutator trick gives us a permutation t which interchanges, say, $(1, \ldots, 1, 1, 1)$ with $(1, \ldots, 1, 1, 2)$, and $(1, \ldots, 1, 1, 3)$ with $(1, \ldots, 1, 1, 0)$. Now there is an instruction g involving the last two coordinates which fixes three of these points and maps the fourth to $(1, \ldots, 1, 2, 0)$. Conjugating t by g gives a permutation which is the product of two transpositions, the first the same as in t, the second swapping $(1, \ldots, 1, 1, 3)$ with $(1, \ldots, 1, 2, 0)$. Now the commutator of t and t^g is a 3-cycle on $(1, \ldots, 1, 1, 3)$, $(1, \ldots, 1, 1, 0)$ and $(1, \ldots, 1, 2, 0)$. So the relation $\sim_1$ is non-trivial, and points which differ in only one coordinate are equivalent. So it is the universal relation, and we are done. □

We would like to emphasise the importance of Theorem 4.1 for the possible implementation of memoryless computation. Recall that any Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by using binary gates (NAND suffices, in fact). This shows that any function can be computed "locally". However, when we want to compute an odd Boolean permutation $g : \{0,1\}^n \to \{0,1\}^n$ without memory, Theorem 4.1 implies that we need an n-ary instruction to compute it. In other words, g cannot be computed "locally".

The main impact of this result (more specifically, its generalisation to any alphabet of even cardinality) is about the design of instruction sets. A CPU core cannot perform any possible operation on all its registers. The typical approach is to use assembly instructions which only work on two or three registers at once (i.e., binary or ternary instructions). However, in any implementation of a core which computes without memory, the instruction set must contain at least one instruction which uses all registers at once. The typical approach to the design of instruction sets is thus inappropriate to memoryless computation.

We close this section by a simple remark on how fast one can compute (even) permutations using only binary instructions, for $q \geq 3$. Suppose that any permutation in G_2 can be computed by a binary program of length L. Any binary instruction is of the form $y_i \leftarrow f_i(y_i, y_j)$, such that f_i induces a permutation of A for any choice of values of all registers but y_i. There are $q!$ choices for each permutation, and q different values of y_j, hence $q(q!)$ choices for f and at most $B := n(n-1)q(q!)$ binary instructions. We then have $B^L \geq |G_2| \geq (q^n)!/2$ and hence

$$L \geq \frac{\log((q^n)!/2)}{\log(n(n-1)q(q!))} \geq q^n \frac{n \log q - 1}{2 \log n + q^2 \log q},$$

thus yielding programs of exponential length.

5 Perspectives

First of all, the reader is reminded of the conjectures on fast groups in Section 2. More generally, memoryless computation, despite some previous work, remains a dawning topic. Many problems, especially of computational nature, arise in this area. For instance, we know that any permutation can be computed by a program with at most $2n-1$ instructions. However, how hard is it to determine such a program computing a given permutation? This problem has a linear analogue: any matrix can be computed in $\lfloor 3n/2 \rfloor$ instructions [10], but the complexity of determining those instructions remains unknown.

Also, in this paper we started the investigation of smaller instruction sets which could still provide the advantages of memoryless computation. In particular, we determined the smallest cardinality of a generating set of instructions. Clearly, using such instruction set will yield very long programs. Can we derive results on the complexity of permutations when the smallest generating sets of instructions are used? We also studied the use of l-ary permutations and showed that they were not sufficient in general. Can we bound the complexity of permutations when binary instructions are sufficient? Furthermore, which other generating sets of instructions could be proposed for memoryless computation, and how good are they in terms of size and complexity?

Finally, an interesting application of our results on memoryless computation is in bioinformatics, more precisely in the modelling of gene regulatory networks, introduced in [19] and [22]

(see [18] and [21] for reviews on this topic). A network of n genes interacting with one another is typically modelled as follows. To each gene i ($1 \leq i \leq n$) are associated the following:

- Firstly, a variable x_i, called its state, taking a value in a finite alphabet A of size q, which indicates the level of activation of the gene (usually, $q = 2$, hence the common notation of a Boolean network).
- Secondly, an update function $f_i : A^k \to A$ which depends on the values of some genes $j_1, \ldots, j_k$ that influence its level of activation: $f_i(x_{j_1}, \ldots, x_{j_k})$.

In general, the order in which the genes update their state, referred to as the update scheme, is unknown. In some models, all the genes are assumed to update their state synchronously, i.e. all at the same time (this is the so-called parallel update scheme). In other cases, the updates can be done asynchronously, in particular, they can be assumed to be updated one after the other (this is the so-called serial update scheme). It is clear that memoryless computation corresponds to the serial update scheme, also called a sequential dynamic system [20], where an update of gene i corresponds to the instruction

$$y_i \leftarrow f_i(y_{j_1}, \ldots, y_{j_k}).$$

One major question is to determine whether any generality is lost by considering one kind of update schedule over another. In this paper, we have proved in Theorem 3.1 (a) the universality of the serial update for permutations. Indeed, for any A and any n (apart from the degenerate case $q = n = 2$), there exists a gene regulatory network $f_1, \ldots, f_n$ which can generate any possible permutation of A^n in its serial update scheme, i.e. for any $g = (g_1, \ldots, g_n) \in \mathrm{Sym}(A^n)$, there exists a word $(i_1, i_2, \ldots, i_L) \in \{1, \ldots, n\}^L$ such that successively updating the states of genes $i_1, i_2, \ldots, i_L$ eventually yields the state $(g_1(x), \ldots, g_n(x))$.

References

[1] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung: Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, July 2000.

[2] Serge Burckel: Closed iterative calculus. *Theoretical Computer Science*, 158:371–378, May 1996.

[3] Serge Burckel: Elementary decompositions of arbitrary maps over finite sets. *Journal of Symbolic Computation*, 37(3):305–310, 2004.

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

[5] Serge Burckel, Emeric Gioan, and Emmanuel Thomé: Computation with no memory, and rearrangeable multicast networks. *Discrete Mathematics and Theoretical Computer Science*, 16:121–142, 2014.

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

[7] Serge Burckel and Marianne Morillon: Quadratic sequential computations of boolean mappings. *Theory of Computing Systems*, 37(4):519–525, 2004.

[8] Serge Burckel and Marianne Morillon: Sequential computation of linear boolean mappings. *Theoretical Computer Science*, 314:287–292, February 2004.

[9] Peter J. Cameron: *Permutation Groups*. Volume 45 of *London Mathematical Society Student Texts*. Cambridge University Press, 1999.

[10] Peter J. Cameron, Ben Fairbairn, and Maximilien Gadouleau: Computing in matrix groups without memory. *Chicago Journal of Computer Science*, to appear.

[11] John D. Dixon and Brian Mortimer: *Permutation Groups*. Number 163 in Graduate Texts in Mathematics. Springer, 1996.

[12] Maximilien Gadouleau and Søren Riis: Memoryless computation: new results, constructions, and extensions. *Theoretical Computer Science*, To appear. Available at `http://www.sciencedirect.com/science/article/pii/S0304397514007300`.

[13] Olexandr Ganyushkin and Volodymyr Mazorchuk: *Classical Finite Transformation Semigroups: An Introduction*. Volume 9 of *Algebra and Applications*. Springer-Verlag, London, 2009.

[14] Dah-Jyn Guan: Generalized Gray codes with applications. *Proc. Natl. Sci. Counc. ROC(A)*, 22(6):841–848, 1998.

[15] Peter M. Higgins: *Techniques of Semigroup Theory*. Oxford University Press, 1992.

[16] John M. Howie: *Fundamentals of Semigroup Theory*. Oxford Science Publications, 1995.

[17] W. M. Kantor and T. P. McDonough: On the maximality of $\mathrm{PSL}(d+1,q)$, $d \geq 2$. *J. London Math. Soc.*, 8:426, 1974.

[18] Guy Karlebach and Ron Shamir: Modelling and analysis of gene regulatory networks. *Nature*, 9:770–780, October 2008.

[19] S. A. Kauffman: Metabolic stability and epigenesis in randomly connected nets. *Journal of Theoretical Biology*, 22:437–467, 1969.

[20] Henning S. Mortveit and Christian M. Reidys: *An Introduction to Sequential Dynamical Systems*. Universitext. Springer, 2008.

[21] Loïc Paulevé and Adrien Richard: Static analysis of boolean networks based on interaction graphs: A survey. *Electronic Notes in Theoretical Computer Science*, 284:93–104, 2012.

[22] R. Thomas: On the relation between the logical structure of systems and their ability to generate multiple steady states or sustained oscillations. *Spriner Series in Synergies*, 9:180–193, 1980.

[23] Raymond W. Yeung, Shuo-Yen Robert Li, Ning Cai, and Zhen Zhang: *Network Coding Theory*. Volume 2 of *Foundation and Trends in Communications and Information Theory*. now Publishers, Hanover, MA, 2006.

Peter J. Cameron
Professor of Mathematics and Statistics
School of Mathematics and Statistics
University of St Andrews, UK
pjc20@st-andrews.ac.uk
http://www-circa.mcs.st-andrews.ac.uk/~pjc/
Emeritus Professor of Mathematics
School of Mathematical Sciences
Queen Mary, University of London, UK
p.j.cameron@qmul.ac.uk
http://www.maths.qmul.ac.uk/~pjc/

Ben Fairbairn
Lecturer in Mathematics
Department of Economics, Mathematics and Statistics
Birkbeck, University of London, UK
b.fairbairn@bbk.ac.uk
http://www.bbk.ac.uk/ems/faculty/fairbairn

Maximilien Gadouleau
Lecturer in Computer Science
School of Engineering and Computing Sciences
Durham University, Durham, UK
m.r.gadouleau@durham.ac.uk
http://community.dur.ac.uk/m.r.gadouleau/