Straight-line programs with memory and matrix Bruhat decomposition

Alice C. Niemeyer, Tomasz Popiel & Cheryl E. Praeger*

December 19, 2017

Abstract

We advocate that straight-line programs designed for algebraic computations should be accompanied by a comprehensive complexity analysis that takes into account both the number of fundamental algebraic operations needed, as well as memory requirements arising during evaluation. We introduce an approach for formalising this idea and, as illustration, construct and analyse straight-line programs for the Bruhat decomposition of $d \times d$ matrices with determinant 1 over a finite field of order q that have length $O(d^2 \log(q))$ and require storing only $O(\log(q))$ matrices during evaluation.

1 Introduction

We propose a comprehensive approach to the analysis of straight-line programs for use in algebraic computations. Our approach facilitates exhaustive complexity analyses which account for both the number of fundamental algebraic operations, as well as memory/storage requirements arising during evaluation. Our aim is to crystallise the ideas underpinning existing data structures, such as those used in GAP [6] and Magma [11], by introducing a data structure which we call a straight-line program with memory (MSLP). We demonstrate the effectiveness of our approach by constructing MSLPs for the Bruhat decomposition of $d \times d$ matrices g with determinant 1 over a finite field, namely the decomposition $g = u_1wu_2$ with u_1, u_2 lower-unitriangular matrices and w a monomial matrix. In particular, we prove the following.

Theorem 1.1 Let $q = p^f$ for some prime p and $f \ge 1$. Given a matrix $g \in \mathrm{SL}(d,q)$, there is a straight-line program to compute the Bruhat decomposition of g which has length $cd^2 \log(q)$ for some absolute constant c and requires storing at most 2f + 18 matrices in memory simultaneously during evaluation.

^{*}The University of Western Australia, 35 Stirling Highway, Crawley 6009 WA, Australia. Email: {alice.niemeyer, tomasz.popiel, cheryl.praeger}@uwa.edu.au.

Our MSLPs for this example are based on the algorithm described by Taylor [17, p. 29] and return the Bruhat decomposition in terms of the Leedham-Green-O'Brien [9] standard generators of SL(d,q), that is, in a form that can be readily used for evaluations in black box special linear groups. We hope that the concept of MSLPs will lead to more transparent complexity and memory usage analyses in a wider context, as motivated below.

Straight-line programs in computational algebra. Straight-line programs (SLPs) have long been used in computer science, as programs without branching or loops, as discussed by Bürgisser et al. [4, Section 4.7]. Nowadays they are a powerful tool in many areas, including genetics [3], compression algorithms [5], and in complexity analyses of algebraic computations [10]. In our context, an SLP is a sequence of instructions where each instruction either copies an element of a given (input) set or utilises only the results of previous instructions.

A seminal paper of Babai and Szemeredi [2] led to the use of SLPs in the analysis of algorithms for computational group theory, and eventually to practically efficient algorithms. The paper introduced SLPs to the computational group theory community, as well as other fundamental concepts that are now used widely in algorithms for groups, such as black box groups. One of the classical results proved by Babai and Szemeredi [2] is that every element of a finite black box group of order n can be reached by an SLP of length at most $(1 + \log(n))^2$.

The monograph by Kantor and Seress [7] led to SLPs playing a crucial role in the design and analysis of constructive group recognition algorithms; see also the book by Seress [14] or the paper by O'Brien [13] for an overview. They now form an integral part of the fundamental machinery in general group recognition algorithms, such as the recognition tree of Leedham-Green and O'Brien [8], which is implemented in the computational algebra package Magma [11], or that proposed by Neunhöffer and Seress [12] and implemented in GAP [6]. In this context, the role of SLPs in computational group theory has shifted. Intended originally as a compact way to encode long group computations to obtain a particular group element, they now need to allow efficient evaluation, often in a preimage under a homomorphism of the group in which they were constructed.

Evaluation of SLPs. The evaluation of an SLP amounts to executing its instructions, replacing the (input) set of elements by a set of elements of interest. An SLP encountered in a recognition algorithm might record a group computation to a particular element in the concretely represented group and then, at a later stage, need to be evaluated in a different group to yield a carefully crafted element. Evaluation in the second group might require a lot more memory, and computation in this group might be far less efficient. Hence, efficient evaluation of SLPs in such less favourable circumstances must be addressed in practice.

A bottleneck of the original concept of SLPs was the lack of a formal means for identifying which subset of the evaluated instructions would be required to evaluate subsequent instructions. Thus, when evaluating an SLP on a given set of group generators and input elements, it was hard to keep track of how many intermediately computed elements would no longer be required for the rest of the evaluation. Consequently, for the complexity analysis of an SLP of length ℓ , the upper bound for the memory requirement during evaluation was ℓ group elements. In many cases, ℓ is not a constant but an increasing function of the size of the input, and storing ℓ group elements may not be possible.

The implementation of SLPs in GAP (see straight.gi in the GAP library [6]) addresses this issue by allowing an already constructed SLP to be analysed via the function SLOTUSAGEPATTERN written by Max Neunhöffer. Information about how to evaluate the SLP efficiently is recorded, so that no unnecessary elements are stored during subsequent evaluations. This ensures the best possible use of memory for a given SLP in practice, but does not yield an upper bound for the memory requirement in a theoretical analysis. A different approach is taken by Bäärnhielm and Leedham-Green [1], who also identify the problem of storing too many intermediate elements. They are concerned with writing efficient SLPs to reach randomly generated group elements, and propose a data structure that contains an additional entry in each instruction to record how many times this instruction will be used, thus also ensuring that no unnecessary elements are stored during subsequent evaluations.

Straight-line programs with memory. While the approaches taken by Bäärnhielm and Leedham-Green and in GAP are tailored to yield efficient evaluation of existing SLPs, such as those produced by a random element generator, our purpose is different. We aim to design efficient SLPs for specific tasks from the start, and to include memory assignments as part of the data structure, with the goal to minimise storage. Our approach requires precise knowledge of the underlying computations for a given task to construct an SLP that can be evaluated in a possibly different, computationally less favourable algebraic structure, while storing only a small number of intermediate elements. In this sense, our *MSLPs* are custom built. Once an MSLP is constructed, we know how much memory is required during evaluation, making the evaluation process more transparent. This brings the construction and analysis of SLPs closer to the aforementioned issues faced in an implementation.

We hope that our approach will lead to SLPs that facilitate efficient evaluation for many important procedures, where most of the intermediate elements constructed are quite specific, for example, where particular elements must be reached from particular generators of the group. We demonstrate the concept by considering a case study, of the Bruhat decomposition of a $d \times d$ matrix with determinant 1 over a finite field of order q, analysing the length and the memory requirements as functions of d and q. Our example demonstrates that keeping track of memory requirements in the design of an SLP can lead to SLPs that are extremely efficient in terms of storage (see Theorem 1.1).

The paper is structured as follows. MSLPs are formally defined and their evaluation discussed in Section 2, where some simple examples are also given. The in-depth Bruhat decomposition example is investigated across Sections 3 and 4, with the latter concluding by drawing together the necessary results to prove Theorem 1.1. We then comment briefly on a GAP implementation of our

2 Straight-line programs with memory

2.1 Definitions

A straight-line program with memory (MSLP) is a sequence $S = [I_1, \ldots, I_n]$ such that each instruction I_r , $r \leq n$, is formally one of the following:

- (i) $m_k \leftarrow m_i$ with $i, k \in \{1, \ldots, b\}$;
- (ii) $m_k \leftarrow m_i * m_j$ with $i, j, k \in \{1, \ldots, b\}$;
- (iii) $m_k \leftarrow m_i^{-1} \text{ with } i, k \in \{1, ..., b\};$
- (iv) Show(A) where $A \subseteq \{1, \ldots, b\}$.

The positive integer b is called the *memory quota* of S, and S is said to be a b-MSLP. The positive integer n is called the length of S, and the empty sequence is also permitted, with length 0. The meaning of these instructions is revealed through evaluation of the MSLP, as follows.

The b-MSLP S may be evaluated with respect to an ordered list M of length b of elements of a group G. The idea is that the memory M will store those group elements that are needed in the evaluation process, with certain elements being overwritten as the evaluation proceeds, in order to minimise the number of elements stored at any given time. Let M[k], $1 \le k \le b$, denote the kth element, or memory slot, of M. The value of S at M from s to t, where either s = t = 0 or $1 \le s \le t \le n$, is denoted by Eval(S, M, s, t) and obtained as follows.

If s=t=0 then $\operatorname{Eval}(S,M,s,t)$ is a list of length 1 containing the identity element of G. If $1 \leq s \leq t \leq n$ then for each $r \in \{s,\ldots,t\}$ in order, we perform one of the following steps:

- (i) If I_r is the instruction $m_k \leftarrow m_i$ for some $i, k \in \{1, ..., b\}$ then store M[i] in memory slot M[k].
- (ii) If I_r is the instruction $m_k \leftarrow m_i * m_j$ with $i, j, k \in \{1, ..., b\}$ then store the product M[i]M[j] in M[k].
- (iii) If I_r is the instruction $m_k \leftarrow m_i^{-1}$ with $i, k \in \{1, \dots, b\}$ then store $M[i]^{-1}$ in M[k].
- (iv) If I_r is the instruction Show(A) where $A \subseteq \{1, \ldots, b\}$ then no action is required.

If for r = t we perform step (i), (ii) or (iii), and hence store an element in M[k], then Eval(S, M, s, t) is defined to be the element stored in M[k]. On the other hand, if the instruction I_t was Show(A) then Eval(S, M, s, t) is defined to be the list of elements stored in memory slots M[i] for $i \in A$.

r	I_r	M after applying I_r
	$m_3 \leftarrow m_2 * m_1$	M = [g, h, hg]
2	$m_3 \leftarrow m_3^{-1}$	$M = [g, h, g^{-1}h^{-1}]$
3	$m_3 \leftarrow m_3 * m_1$	$M = [g, h, g^{-1}h^{-1}g]$
4	$m_3 \leftarrow m_3 * m_2$	$M = [g, h, g^{-1}h^{-1}gh]$

Table 1: 3-MSLP for the commutator of two group elements. Eval $(S, M, 1, 4) = g^{-1}h^{-1}gh$ when S is evaluated with initial memory M = [g, h, 1].

Remark 2.1 Instruction type (i) above simply copies an element already in memory to a different memory slot. These instructions can arguably be disregarded for the purpose of determining the length of an MSLP, because in a practical implementation they could be handled via relabelling. Therefore, for simplicity we ignore instructions of the form (i) when determining the lengths of our MSLPs in Sections 3 and 4.

2.2 Basic examples

To demonstrate how MSLPs function in practice, we discuss MSLPs for some fundamental group operations that arise in our more involved examples in the subsequent sections.

(i) Commutators

Consider computing the commutator $[g,h] = g^{-1}h^{-1}gh$ of two group elements. Let us suppose that we wish to never overwrite the input elements g and h, which we store in memory slots 1 and 2, respectively. We begin by forming the product hg and storing it in a new (third) memory slot. The element hg is then overwritten by its inverse $g^{-1}h^{-1}$. This element is then overwritten by $g^{-1}h^{-1}g$, which is in turn overwritten by $g^{-1}h^{-1}gh = [g,h]$. A total of four MSLP instructions (group multiplications or inversions) are required, and only one memory slot is needed in addition to the two memory slots used to permanently store the input elements g,h. In other words, there exists an MSLP S with memory quota b=3 and length 4, such that when evaluated with input containing g and h, S returns output containing [g,h]. Specifically, if we take initial/input memory M=[g,h,1] then the instructions in Table 1 yield final/output memory with M[3] the commutator [g,h]. (Unfortunately we have here a clash of notation with $[\cdot,\cdot]$ denoting both the commutator and, potentially, a list of length 2, but the meaning should be clear.)

(ii) Powering via repeated squaring

Consider powering a group element g up to g^{ℓ} , say, via repeated squaring. This can be done by using one memory slot to store the powers $g^2, g^4, g^8, \ldots, g^{\lfloor \log_2(\ell) \rfloor}$, overwriting each g^i with g^{2i} while another memory slot stores a running product of g^2 -powers that is updated (if need be) upon the calculation of each g^i and

r	I_r	M after applying I_r
1	$m_2 \leftarrow m_1 * m_1$	$[g, g^2, 1, 1]$
2	$m_3 \leftarrow m_1 * m_2$	$[g, g^2, g^3, 1]$
3	$m_4 \leftarrow m_1 * m_2$	$[g, g^2, g^3, g^3]$
4	$m_2 \leftarrow m_2 * m_2$	$[g, g^4, g^3, g^3]$
5	$m_4 \leftarrow m_2 * m_4$	$[g, g^4, g^3, g^7]$
6	Show(3,4)	$[g, g^4, g^3, g^7]$

Table 2: 4-MSLP for computing the third and seventh powers of a group element. Eval $(S, M, 1, 6) = [g^3, g^7]$ when S is evaluated with initial memory M = [g, 1, 1, 1].

eventually becomes g^{ℓ} . This procedure is completed in at most $2\lfloor \log_2(\ell) \rfloor \leq 2\log_2(\ell)$ MSLP instructions (group multiplications). In other words, there exists a 3-MSLP of length at most $2\log_2(\ell)$ that when evaluated with memory containing g returns output containing g^{ℓ} (one memory slot permanently stores the input g while the other two slots are used for the intermediate computations). Table 2 details an explicit and slightly more complicated example, where two powers of g are computed simultaneously via a 4-MSLP, thereby also demonstrating the 'Show' instruction.

3 Bruhat decomposition: step 1

In Section 4 we describe an MSLP for computing the Bruhat decomposition $g = u_1wu_2$ of a matrix $g \in SL(d,q)$. The algorithm we use is along the lines of that given by Taylor [17, p. 29]. We also note the recent interesting account by Strang [16, Section 4] of the Bruhat decomposition and its history.

The construction in Section 4 outputs an MSLP which evaluates to the monomial matrix $w = u_1^{-1} g u_2^{-1}$. The lower-unitriangular matrices u_1 and u_2 are returned as words in the Leedham-Green–O'Brien [9] standard generators for $\mathrm{SL}(d,q)$ defined in Section 3.1 below. In this section we explain how to express an arbitrary monomial matrix $w \in \mathrm{SL}(d,q)$ as a word in the Leedham-Green-O'Brien standard generators, yielding an MSLP from the standard generators to w. When concatenated with the MSLP from Section 4, this gives an MSLP for the complete Bruhat decomposition of g.

3.1 Standard generators for SL(d, q)

Let q be a prime power, say $q = p^f$, and d a positive integer. Fix an ordered basis for the natural module of SL(d,q) and a primitive element $\omega \in \mathbb{F}_q$. Let

$$s_0 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad t_0 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad x_0 = \begin{pmatrix} 0 & I_3 \\ -1 & 0 \end{pmatrix}$$

and define the following matrices in SL(d, q):

$$\delta = \operatorname{diag}(\omega, \omega^{-1}, 1, \dots, 1), \quad s = \begin{pmatrix} s_0 & 0 \\ 0 & I_{d-2} \end{pmatrix}, \quad t = \begin{pmatrix} t_0 & 0 \\ 0 & I_{d-2} \end{pmatrix},$$

$$v = \begin{pmatrix} 0 & I_{d-2} \\ I_2 & 0 \end{pmatrix} \text{ or } \begin{pmatrix} 0 & 1 \\ -I_{d-1} & 0 \end{pmatrix} \text{ for } d \text{ even or odd, respectively,}$$

$$x = \begin{pmatrix} x_0 & 0 \\ 0 & I_{d-4} \end{pmatrix} \text{ or } I_d \text{ for } d \text{ even or odd, respectively.}$$

These are the *standard generators* of SL(d, q) as given by Leedham-Green and O'Brien [9].

3.2 Strategy

Let N be the subgroup of $\mathrm{SL}(d,q)$ whose matrices are monomial with respect to our chosen basis (namely, they have exactly one nonzero entry in each row and column). Let Ψ denote the homomorphism from N onto the symmetric group S_d such that if e_1,\ldots,e_d is our basis then $w\in N$ permutes the subspaces $\langle e_1\rangle,\ldots,\langle e_d\rangle$ in the same way as $\Psi(w)$ permutes $1,\ldots,d$.

Suppose first that d is odd. Observe that $\Psi(v^{-1})$ is the d-cycle $(1 \dots d)$ and $\Psi(s)$ is the transposition (1 2). We first construct an MSLP that, when evaluated with input memory containing the generating set $\{(1 \dots d), (1 \ 2)\}$ for S_d , outputs, as a word in these generators, a permutation \bar{w} such that $\Psi(w) = \bar{w}$. This MSLP is described in Section 3.3. We then evaluate this same MSLP with input containing $\{v^{-1}, s\}$ to return, as a word in v and s, a monomial matrix $w' \in SL(d,q)$ such that $\Psi(w') = \Psi(w)$. Finally, we construct a second MSLP, described in Section 3.4, that writes the diagonal matrix $h := w(w')^{-1} \in SL(d,q)$ as a word in the standard generators of SL(q,d) (when evaluated with these generators as input). Combining the constructions in Sections 3.3 and 3.4 yields, as required, the monomial matrix

$$w = hw'$$

as a word in the Leedham-Green-O'Brien standard generators of SL(d, q).

The only difference for the case where d is even is that the standard generator v does not have the property $\Psi(v^{-1})=(1\ldots d);$ more to the point, $\Psi(v)=(1\ 3\ldots d-1)(2\ 4\ldots d)$ is not a d-cycle in this case. To overcome this issue, it suffices to replace v^{-1} in the above argument by a word z in v and s such that $\Psi(z)$ is a d-cycle, and then replace the d-cycle $(1\ldots d)$ in Section 3.3 by $\Psi(z)$ via a straightforward relabelling. For example, one can set z=sv, so that $\Psi(z)=(1\ 4\ 6\ldots d\ 2\ 3\ 5\ldots d-1).$

3.3 Permutations

As explained above, we consider being given a monomial matrix in SL(d, q) and hence, via the homomorphism $\Psi : N \to S_d$, a permutation

$$i \mapsto \pi(i), \quad i = 1, \dots, d.$$

We seek an MSLP that, when evaluated with input $(1\ 2)$ and (1...d), outputs π as a word in these generators of S_d . Let us assume that the inverse of (1...d) is also given as input. We use the Schreier-Sims algorithm [15]. Writing

$$s_1 := (1 \ 2)$$
 and $v_1 := (1 \ d \ d - 1 \ \dots \ 2) = (1 \ 2 \ \dots \ d)^{-1}$,

we have

$$\pi = \left(\prod_{i=1}^{d} v_i^{\pi(i)-1}\right)^{-1}, \quad \text{where } v_i := (i \ d \ d-1 \ \dots \ i+1). \tag{1}$$

That is, $\pi_1 := \pi v_1^{\pi(1)-1}$ fixes 1, $\pi_2 := \pi_1 v_2^{\pi(2)-1}$ fixes 1 and 2, and so on, with

$$\pi_d = \pi \prod_{i=1}^d v_i^{\pi(i)-1} = 1.$$

The v_i are computed recursively from s_1 and v_1 according to

$$v_i = s_{i-1}v_{i-1}, \text{ where } s_i := (i \ i+1),$$
 (2)

with the s_i computed recursively via

$$s_i = v_1 s_{i-1} v_1^{-1}. (3)$$

Proposition 3.1 Let $\lambda = 2d \log_2(d) + 4d$ and b = 8. There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing $\{s_1, v_1, v_1^{-1}\}$ then S outputs memory containing the permutation $\pi \in S_d$.

Proof. Computing the s_i , $2 \le i \le d-1$, via the recursion (3) costs 2(d-2) < 2d MSLP instructions, and computing the v_i , $2 \le i \le d$, via (2) costs d-1 < d instructions. Powering each v_i , $1 \le i \le d$, up to $v_i^{\pi(i)-1}$ costs at most

$$2\log_2(\pi(i) - 1) < 2\log_2(d)$$

instructions (see Section 2.2(ii)), and forming all of the $v_i^{\pi(i)-1}$ therefore costs at most $2d\log_2(d)$ instructions. Computing and inverting the product $\prod_{i=1}^d v_i^{\pi(i)-1}$ then costs a further d instructions. So the permutation π is computed as per (1) in at most $2d\log_2(d) + 4d$ MSLP instructions.

In addition to the three memory slots needed to store the input $\{s_1, v_1, v_1^{-1}\}$, the memory quota for the MSLP is determined as follows. Only one slot is required to store the s_i , because each s_i can overwrite s_{i-1} when computed via the recursion (3). Similarly, one slot suffices to store the v_i . Two slots (at most) are required for the purpose of powering the v_i up to $v_i^{\pi(i)-1}$ via repeated squaring (see Section 2.2(i)): once each $v_i^{\pi(i)-1}$ is computed, it can be multiplied by the product $v_1^{\pi(1)-1} \cdots v_{i-1}^{\pi(i-1)-1}$, which is stored in its own memory slot. So at most 3+1+1+2+1=8 memory slots are needed for the MSLP.

3.4 Diagonal matrices

We now explain how to write a diagonal matrix in SL(d,q) as an MSLP in the Leedham-Green-O'Brien standard generators. This is done by multiplying together powers of the particular diagonal matrices considered in the following lemma. Here, as per Section 3.1, ω denotes a primitive element of \mathbb{F}_q .

Lemma 3.2 For j = 1, ..., d - 1, write

$$h_j = \operatorname{diag}(\underbrace{1,\ldots,1}_{j-1},\omega,\omega^{-1},1,\ldots,1) \in \operatorname{SL}(d,q).$$

Then $h_1 = \delta$ and the other h_j are computed recursively as follows:

$$h_2 = x^{-1}h_1x \quad \text{if d is even,}$$

$$h_j = \begin{cases} vh_{j-1}v^{-1} & \text{if d is odd} \\ v^{-1}h_{j-2}v & \text{if d is even and } j \neq 2. \end{cases}$$

Proof. First suppose that d is odd. It suffices to show that given a diagonal matrix $h = \text{diag}(b_1, \ldots, b_d)$ we have $vhv^{-1} = \text{diag}(b_d, b_1, \ldots, b_{d-1})$. Write our basis for SL(d, q) as e_1, \ldots, e_d . Then

$$v: \begin{cases} e_1 \mapsto e_d \\ e_i \mapsto -e_{i-1}, \quad i = 2, \dots, d \end{cases}$$

and hence

$$e_1 vhv^{-1} = e_d hv^{-1} = b_d e_d v^{-1} = b_d e_1,$$

 $e_i vhv^{-1} = -e_{i-1} hv^{-1} = -b_{i-1} e_{i-1} v^{-1} = b_{i-1} e_i, \quad i = 2, \dots, d.$

That is, $vhv^{-1} = \text{diag}(b_d, b_1, \dots, b_{d-1})$ as required.

Now suppose that d is even, and again write our basis as e_1, \ldots, e_d . The fact that $x^{-1}\delta x = \operatorname{diag}(1,\omega,\omega^{-1},1,\ldots,1)$, namely $h_2 = x^{-1}h_1x$, is verified by noting that x fixes e_5,\ldots,e_d and $x:(e_1,e_2,e_3,e_4)\mapsto (e_2,e_3,e_4,-e_1)$. It then suffices to check that given a diagonal matrix $h=\operatorname{diag}(b_1,\ldots,b_d)$ we have $v^{-1}hv=\operatorname{diag}(b_{d-1},b_d,b_1,\ldots,b_{d-2})$. This is straightforward to verify on noting that

$$v: \begin{cases} e_i \mapsto e_{i+2}, & i = 1, \dots, d-2 \\ (e_{d-1}, e_d) \mapsto (e_1, e_2). \end{cases}$$

Lemma 3.3 Let $h \in SL(d,q)$, and write $h = diag(\omega^{\ell_1}, \ldots, \omega^{\ell_d})$ for some ℓ_1, \ldots, ℓ_d . Then

$$h = \prod_{j=1}^{d-1} h_j^{\lambda_j}, \quad \text{where } \lambda_j = \sum_{k=1}^j \ell_k.$$
 (4)

Proof. The matrix $g:=\prod_{j=1}^{d-1}h_j^{\lambda_j}$ is diagonal with determinant 1, being a product of diagonal matrices with determinant 1. Since each h_j has 1 in every diagonal entry except the jth and the (j+1)th, the first entry of g is $\omega^{\lambda_1}=\omega^{\ell_1}$ and for $i=2,\ldots,d-1$ the ith entry is the product of the ith entries of h_i and h_{i-1} , namely $\omega^{\lambda_i}\omega^{-\lambda_{i-1}}=\omega^{\ell_i}$. The dth entry of g is the dth entry of h_{d-1} , namely $u^{-\lambda_{d-1}}$, and $u^{-\lambda_{d-1}}=-\sum_{i=1}^{d-1}\ell_i\pmod{q}\equiv\ell_d\pmod{q}$ since u and u and u and u and u as a claimed.

We now compute upper bounds for the length and memory quota of an MSLP for expressing an arbitrary diagonal matrix $h \in \mathrm{SL}(d,q)$ as a word in the Leedham-Green–O'Brien standard generators.

Proposition 3.4 Let $\lambda = 2d \log_2(d) + 2d \log_2(q) + 3d$ and b = 15. Let $h \in \operatorname{SL}(d,q)$ be a diagonal matrix given in the form $h = \operatorname{diag}(\omega^{\ell_1}, \ldots, \omega^{\ell_d})$. There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the set $X = \{s, s^{-1}, t, t^{-1}, \delta, \delta^{-1}, v, v^{-1}, x, x^{-1}\}$ then S outputs final memory containing h.

Proof. We want our MSLP to write h as the product (4), with the h_j computed in terms of the standard generators as per Lemma 3.2. Computing the h_j via Lemma 3.2 costs 2(d-1) < 2d MSLP instructions: h_1 is already one of the standard generators and each subsequent h_j is formed by conjugating either h_{j-1} or h_{j-2} by either v, v^{-1} or x^{-1} . Powering each h_j up to $h_j^{\lambda_j}$ via repeated squaring costs (see Section 2.2(ii)) at most

$$2\log_2(\lambda_j) = 2\log_2\left(\sum_{k=1}^j \ell_k\right) \le 2\log_2(j(q-1)) < 2\log_2(q) + 2\log_2(j)$$

MSLP instructions, and so computing all of the $h_j^{\lambda_j}$ requires at most

$$2d\log_2(d) + 2d\log_2(q)$$

instructions. Finally, d-2 < d instructions are required to multiply the $h_j^{\lambda_j}$ together, yielding the claimed maximum total length of the MSLP:

$$2d + (2d\log_2(d) + 2d\log_2(q)) + d = 2d\log_2(d) + 2d\log_2(q) + 3d.$$

In addition to storing the input X, the memory quota for the MSLP is as follows. At most two memory slots are needed to store all of the h_j because the recursion for the h_j refers back to h_{j-1} when d is odd (necessitating only one memory slot) and to h_{j-2} when d is even (necessitating two memory slots); that is, each h_j can overwrite the one from which it is computed. Two memory slots are needed for computing the $h_j^{\lambda_j}$ (see Section 2.2(ii)), and one final memory slot is needed to multiply these together (the product of the first j-1 is overwritten by the product of the first j upon multiplication by $h_j^{\lambda_j}$, which can itself then be discarded). Therefore, the MSLP requires at most 2+2+1+|X|=15 elements to be stored in memory at any given time.

Remark 3.5 In practice, given a diagonal matrix h, we would need to compute d discrete logarithms to be able to write h in the form $\operatorname{diag}(\omega^{\ell_1}, \ldots, \omega^{\ell_d})$.

4 Bruhat decomposition: step 2

4.1 Preliminaries

We recall from the book by Taylor [17, p. 29] an algorithm for obtaining the Bruhat decomposition of a matrix

$$g = (g_{ij}) \in SL(d,q),$$

namely for writing g in the form

$$g = u_1 w u_2$$
 with u_1 , u_2 lower unitriangular and w monomial. (5)

The algorithm reduces g to the monomial matrix w by multiplying g by certain transvections, namely matrices of the following form.

Definition 4.1 Let e_1, \ldots, e_d denote our basis for V(d, q). For $i = 2, \ldots, d$ and $j = 1, \ldots, i - 1$, and $\alpha \in \mathbb{F}_q$, define $t_{ij}(\alpha) \in SL(d, q)$ by

$$t_{ij}(\alpha): e_k \mapsto e_k + \alpha \Delta_{ki} e_j$$
 where $\Delta_{ki} = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise.} \end{cases}$

That is, the matrix for $t_{ij}(\alpha)$ has ones along the main diagonal, α in entry (i, j), and zeroes elsewhere.

Multiplying g on the left by the transvection $t_{ij}(\alpha)$ effects an elementary row operation that adds α times the jth row to the ith row. Similarly, right multiplication by $t_{ij}(\alpha)$ effects an elementary column operation, adding α times column i to row j. Using the row operations, one can reduce g to a matrix with exactly one nonzero entry in its dth column, say in row r. Then the elementary column operations can be used to reduce the other entries in row r to zero. Continuing recursively, g can be reduced to a matrix with exactly one nonzero entry in each row and each column. Moreover, at the end of the procedure, the products of transvections on the left- and right-hand sides of g both form unitriangular matrices, because j < i. In other words, one obtains a decomposition of the form (5). Note that the described procedure differs from the one in Taylor's book in that we multiply g on the left and right by lower-triangular transvections (j < i), as opposed to upper-triangular transvections (i < j).

Our aim is to determine the length and memory quota for an MSLP for the Bruhat decomposition of an arbitrary matrix $g \in SL(d,q)$ via the above method, with the matrices u_1, u_2, w returned as words in the Leedham-Green-O'Brien [9] standard generators s, t, v, δ, x of SL(d,q) given in the previous section. We need the following lemma.

Lemma 4.2 If $\alpha \in \mathbb{F}_q$ is written as a polynomial in the primitive element ω , say $\alpha = \sum_{\ell} a_{\ell} \omega^{\ell}$, then

$$t_{ij}(\alpha) = \prod_{\ell} t_{ij}(\omega^{\ell})^{a_{\ell}}.$$

The $t_{21}(\omega^{\ell})$ are given by

$$t_{21}(\omega^{\ell}) = \begin{cases} (\delta^{-\ell}v\delta^{-\ell}v^{-1})st^{-1}s^{-1}(\delta^{-\ell}v\delta^{-\ell}v^{-1})^{-1} & \text{if } d \text{ is odd} \\ (\delta^{-\ell}x^{-1}\delta^{-\ell}x)st^{-1}s^{-1}(\delta^{-\ell}x^{-1}\delta^{-\ell}x)^{-1} & \text{if } d \text{ is even.} \end{cases}$$
(6)

The other $t_{ij}(\alpha)$, and in particular the $t_{ij}(\omega^{\ell})$, are then computed recursively:

$$t_{32}(\alpha) = (xv^{-1})t_{21}(\alpha)(xv^{-1})^{-1}$$
 if d is even, (7)

$$t_{i(i-1)}(\alpha) = \begin{cases} vt_{(i-1)(i-2)}(\alpha)v^{-1} & \text{if } d \text{ is odd} \\ v^{-1}t_{(i-2)(i-3)}(\alpha)v & \text{if } d \text{ is even and } i \neq 3, \end{cases}$$
(8)

$$t_{ij}(\alpha) = [t_{i(j+1)}(1), t_{(j+1)j}(\alpha)] = [t_{i(i-1)}(\alpha), t_{(i-1)j}(1)] \quad \text{if } i - j > 1. \quad (9)$$

Proof. The first assertion, namely that $t_{ij}(\sum_{\ell} a_{\ell}\omega^{\ell}) = \prod_{\ell} t_{ij}(\omega^{\ell})^{a_{\ell}}$, is clear. The final assertion (9) follows immediately from a result in Taylor's book [17, Lemma 5.7]. Let us now justify the claimed expression (6) for $t_{21}(\omega^{\ell})$. First note that $\delta^{-\ell} = \operatorname{diag}(\omega^{-\ell}, \omega^{\ell}, 1, \dots, 1)$. If d is odd, we know from the proof of Lemma 3.2 that $v\delta^{-\ell}v^{-1} = \operatorname{diag}(1, \omega^{-\ell}, \omega^{\ell}, 1, \dots, 1)$. So $\delta^{-\ell}v\delta^{-\ell}v^{-1} = \operatorname{diag}(\omega^{-\ell}, 1, \omega^{\ell}, 1, \dots, 1)$, and the expression for $t_{21}(\omega^{\ell})$ can be verified directly by matrix multiplication, with the upper-left 3×3 block of

$$(\delta^{-\ell} v \delta^{-\ell} v^{-1}) s t^{-1} s^{-1} (\delta^{-\ell} v \delta^{-\ell} v^{-1})^{-1}$$

taking the form

$$\left(\begin{array}{ccc} \omega^{-\ell} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \omega^{\ell} \end{array} \right) \left(\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right) \left(\begin{array}{ccc} \omega^{\ell} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \omega^{-\ell} \end{array} \right) = \left(\begin{array}{ccc} 1 & 0 & 0 \\ \omega^{\ell} & 1 & 0 \\ 0 & 0 & 1 \end{array} \right).$$

For d even we have $x^{-1}\delta^{-\ell}x = \operatorname{diag}(1,\omega^{-\ell},\omega^{\ell},1,\ldots,1)$ from the proof of Lemma 3.2. So in this case $\delta^{-\ell}x^{-1}\delta^{-\ell}x = \operatorname{diag}(\omega^{-\ell},1,\omega^{\ell},1,\ldots,1)$, and the claimed expression for $t_{21}(\omega^{\ell})$ follows as in the odd d case.

Next we check the expression (8) for $t_{i(i-1)}$ in the case where d is odd. We must check that $vt_{i(i-1)}(\alpha)v^{-1} = t_{(i+1)i}(\alpha)$ (for $i \neq d$), namely that the left-hand side maps e_k to $e_k + \alpha \Delta_{k(i+1)}e_i$. Recall that for d odd,

$$v: \begin{cases} e_1 \mapsto e_d \\ e_j \mapsto -e_{j-1}, \quad j = 2, \dots, d. \end{cases}$$

If $k \neq i+1$ then $e_k v \notin \text{span}\{e_i\}$, so $t_{i(i-1)}(\alpha)$ fixes $e_k v$ and hence $e_k v t_{ij}(\alpha) v^{-1} = e_k v v^{-1} = e_k$. For k = i+1 we have (remembering that by assumption here $i \neq d$)

$$e_{i+1}vt_{i(i-1)}(\alpha)v^{-1} = -e_it_{i(i-1)}(\alpha)v^{-1} = (-e_i - \alpha e_{i-1})v^{-1} = e_{i+1} + \alpha e_i.$$

We now verify (8) for d even. For i even (and $i \neq d$) we need to check that $v^{-1}t_{i(i-1)}(\alpha)v = t_{(i+2)(i+1)}(\alpha)$, namely that the left-hand side maps e_{i+2} to $e_{i+2} + \alpha e_{i+1}$ and fixes e_k when $k \neq i+2$. Recall that, for d even,

$$v: \begin{cases} e_j \mapsto e_{j+2}, & j = 1, \dots, d-2 \\ (e_{d-1}, e_d) \mapsto (e_1, e_2). \end{cases}$$

If $k \neq i+2$ then $e_k v^{-1} \notin \text{span}\{e_i\}$, so $t_{i(i-1)}(\alpha)$ fixes $e_k v^{-1}$ and hence $v^{-1}t_{i(i-1)}(\alpha)v$ fixes e_k , while

$$e_{i+2}v^{-1}t_{i(i-1)}(\alpha)v = e_it_{i(i-1)}(\alpha)v = (e_i + \alpha e_{i-1})v = e_{i+2} + \alpha e_{i+1}.$$

For i odd we again use the property $v^{-1}t_{i(i-1)}(\alpha)v = t_{(i+2)(i+1)}(\alpha)$ (note that the above proof does not depend on the parity of i), together with equation (7). To verify (7), recall that x fixes e_5, \ldots, e_d and maps (e_1, e_2, e_3, e_4) to $(e_2, e_3, e_4, -e_1)$. We have $e_3(xv^{-1}) = e_4v^{-1} = e_2$ and hence

$$e_3(xv^{-1})t_{21}(\alpha)(xv^{-1})^{-1} = e_2t_{21}(\alpha)vx^{-1} = (e_2 + \alpha e_1)vx^{-1}$$
$$= (e_4 + \alpha e_3)x^{-1} = (e_3 + \alpha e_2) = e_3t_{32}(\alpha),$$

while $e_k(xv^{-1}) \notin \text{span}\{e_2\}$ if $k \neq 3$, so in this case $e_k(xv^{-1})t_{21}(\alpha)(xv^{-1})^{-1} = e_k(xv^{-1})(xv^{-1})^{-1} = e_k = e_kt_{32}(\alpha)$.

As we run through the algorithm described by Taylor, we deal with the columns of g in reverse order, beginning with column d. Suppose we have reached column c, for some $c \in \{1, \ldots, d\}$. At this stage, g has been reduced to a matrix in which columns $c-1, \ldots, d$ have exactly one nonzero entry (and these entries are in different rows). This has been achieved by multiplying g on the left and right by certain transvections, the products of which are lower-unitriangular matrices.

We identify the first row, call it the rth row, in which the cth column contains a nonzero entry; that is, we set

$$r = r(c) := \min_{1 \le i \le d} \{g_{ic} \ne 0\}.$$

The idea is to 'clear' the cth column, namely to apply elementary row operations to make the entries in rows $i = r + 1, \ldots, d$ zero. Specifically, g is multiplied on the left by the transvections

$$t_{ir}(-g_{ic}g_{rc}^{-1}),$$
 (10)

which subtract $g_{ic}g_{rc}^{-1}$ times the rth row of h from the ith row. Having cleared column c, we clear the corresponding rth row by multiplying h on the right by the transvections

$$t_{cj}(-g_{rj}g_{rc}^{-1}), (11)$$

for j = 1, ..., c - 1, which subtract $g_{rj}g_{rc}^{-1}$ times the cth column of h from the jth column.

For the purposes of costing Taylor's algorithm in terms of matrix operations, namely determining the length of an MSLP for the algorithm, we assume that the field elements $-g_{ic}g_{rc}^{-1}$ in (10) (and similarly in (11)) are given to us as polynomials of degree at most f-1 in the primitive element ω , where $q=p^f$ for some prime p. So we need to know (i) the number of matrix operations required to compute $t_{ij}(\alpha)$ for an arbitrary polynomial $\alpha = \sum_{\ell=0}^{f-1} a_{\ell} \omega^{\ell}$, for arbitrary i, j, and (ii) the maximum number of group elements that need to be kept in computer memory simultaneously during this computation.

Lemma 4.3 Let $\lambda = 2\log_2(q) + f - 1$ and b = f + 3, and let $\alpha = \sum_{\ell=0}^{f-1} a_\ell \omega^\ell$, with $\alpha_0, \ldots, \alpha_{f-1} \in \mathbb{F}_p$. There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the transvections $t_{ij}(\omega^\ell)$ for $\ell = 0, \ldots, f - 1$, then S returns memory containing the transvection $t_{ij}(\alpha) = \prod_{\ell=0}^{f-1} t_{ij}(\omega^\ell)^{a_\ell}$.

Proof. Powering each $t_{ij}(\omega^{\ell})$ up to $t_{ij}(\omega^{\ell})^{a_{\ell}}$ via repeated squaring costs at most $2\log_2(a_{\ell}) \leq 2\log_2(p-1) < 2\log_2(p)$ MSLP instructions whilst storing at most two elements (see Section 2.2(ii)). So computing all of the $t_{ij}(\omega^{\ell})^{a_{\ell}}$ costs at most $2f\log_2(p) = 2\log_p(q)\log_2(p) = 2\log_2(q)$ instructions, and then an extra $f-1 = \log_p(q)-1$ instructions are needed to obtain $t_{ij}(\alpha)$. The memory quota increases by only one element, since each $t_{ij}(\omega^{\ell})^{a_{\ell}}$ can be computed in turn, multiplied by the product of the previously computed transvections, and then discarded. So the total memory quota is b=f+3.

4.2 Pseudocode and analysis for d odd

Pseudocode for Taylor's algorithm for obtaining the Bruhat decomposition of an arbitrary matrix $g \in \mathrm{SL}(d,q)$ is presented in Algorithm 1 for the case where d is odd. The case where d is even is very similar, but requires a few changes that would complicate the pseudocode. So, for the sake of presentation, we analyse the d odd case here and then explain the differences for the d even case in the next subsection. To aid the exposition and analysis, Algorithm 1 refers to several subroutines, namely Algorithms 2–5.

We determine upper bounds for the length and memory quota of an MSLP for Algorithm 1. Recall that we use the Leedham-Green–O'Brien [9] standard generators δ, s, t, v, x of $\mathrm{SL}(d,q)$; although x=1 when d is odd, we include x here for consistency with the d even case in the next subsection. Define the (ordered) list

$$Y = Y(w, u_1, u_2) := [s, s^{-1}, t, t^{-1}, \delta, \delta^{-1}, v, v^{-1}, x, x^{-1}, w, u_1, u_2].$$
(12)

The idea is that our MSLP, when evaluated with initial memory containing Y(g, 1, 1), should output final memory containing $Y(w, u_1, u_2)$ with $g = u_1wu_2$ the Bruhat decomposition of g. In other words, our algorithm initialises w := g, $u_1 := 1$ and $u_2 := 1$ and multiplies w, u_1 and u_2 by the transvections necessary to render $g = u_1wu_2$ with w monomial and u_1, u_2 lower unitriangular.

As for the simpler examples considered in the previous section, here for the sake of presentation we do not write down explicit MSLP *instructions*, but instead determine the cost of Algorithm 1 while keeping track of the number of elements that an MSLP for this algorithm would need to keep in memory at any given time. We prove the following.

Proposition 4.4 Let $q = p^f$ with p prime and $f \ge 1$, let d be an odd integer with $d \ge 3$, and set b = f + 18 and

$$\lambda = d^2(2\log_2(q) + 5f + 10) + 4d(\log_2(q) + 1) + (5f + 1).$$

There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the list Y(g,1,1) as in (12) then S outputs memory containing $Y(w,u_1,u_2)$ with $g=u_1wu_2$ the Bruhat decomposition of g.

Let us also write

$$T_i := \{t_{i(i-1)}(\omega^{\ell}) \mid \ell = 0, \dots, f-1\} \text{ for } i = 2, \dots, d.$$

As noted in the previous subsection, these sets of transvections are needed to construct the transvections by which q is to be multiplied.

In practice, the MSLP should be constructed in such a way that the 'input' of each of the subroutines (Algorithms 2–5) is stored in memory when the subroutine is called and the 'output' is kept in memory for the subsequent stage of Algorithm 1. The cost of the subroutines is determined with this in mind; that is, for each subroutine we determine the maximum length and memory requirement for an MSLP that returns the required output when evaluated with an initial memory containing the appropriate input.

(i) Computing T_2 from the standard generators

The first step of the algorithm is the one-off computation of T_2 from the Leedham-Green-O'Brien standard generators of SL(d,q). The length and memory requirement of an MSLP for this step is as follows.

Lemma 4.5 Let $\lambda = 5f - 1$ and b = f + 14. There exists a b-MSLP, S, of length λ such that if S is evaluated with memory containing the list $Y(w, u_1, u_2)$ as in (12) then S outputs memory containing T_2 and $Y(w, u_1, u_2)$.

Proof. Recall the expression (6) for the $t_{21}(\omega^{\ell})$ in the d odd case, namely

$$t_{21}(\omega^{\ell}) = (\delta^{-\ell}v\delta^{-\ell}v^{-1})st^{-1}s^{-1}(\delta^{-\ell}v\delta^{-\ell}v^{-1})^{-1}.$$

Computing $t_{21}(1) = st^{-1}s^{-1}$ costs two matrix multiplications of elements from Y, and thus adds two instructions to our MSLP. One new memory slot, in addition to the |Y| = 13 initial slots, is needed to store $t_{21}(1)$, If $f \ge 2$ then we continue by observing that, for $\ell = 1, \ldots, f - 1$,

$$t_{21}(\omega^{\ell}) = z_{\ell} \hat{t} z_{\ell}^{-1},$$

Algorithm 1:

```
BruhatDecompositionOdd[g] Input: the list Y in (12) with w := g
          and u_1, u_2 := 1;
Output: a monomial matrix w \in SL(d,q) and two lower unitriangular
            matrices u_1, u_2 \in SL(d, q) such that g = u_1wu_2;
compute T_2 from the standard generators;
for c = d, \ldots, 1 do
    if r := \min\{i \mid h_{ic}\} \le d - 1 then
        call subroutine FIRSTTRANSVECTIONS[r] (Algorithm 2);
        if r \leq d-2 then
            for i=r+2,\ldots,d do
             call subroutine LeftUpdate[i] (Algorithm 3);
     end
    \mathbf{end}
    if c \geq 2 then
        call subroutine LastTransvections[c] (Algorithm 4);
        if c \geq 3 then
             \begin{aligned} & \textbf{for} \ j = c-2, \dots, 1 \ \textbf{do} \\ & \big| \ \ \text{call subroutine RightUpdate}[j] \ (\text{Algorithm 5}); \end{aligned} 
        end
    end
\mathbf{end}
u_1 := u_1^{-1};
u_2 := u_2^{-1};
return w, u_1, u_2;
```

where $\hat{t} := v^{-1}t_{21}(1)v^{-1}$ and the z_{ℓ} are given recursively by

$$z_{\ell} = \delta^{-1} z_{\ell-1} \delta^{-1}$$
 with $z_1 = \delta^{-1} v \delta^{-1}$.

Computing \hat{t} requires two MSLP instructions (matrix multiplications) and one new memory slot. Computing the z_{ℓ} requires 2(f-1) instructions, but also only one new memory slot because each z_{ℓ} can overwrite $z_{\ell-1}$ as the algorithm proceeds recursively. Similarly, computing the z_{ℓ}^{-1} takes (f-1) instructions but only one new memory slot. Forming the $t_{21}(\omega^{\ell}) = z_{\ell}\hat{t}z_{\ell}^{-1}$ costs 2(f-1) instructions, and f-2 new memory slots because each $t_{21}(\omega^{\ell})$ needs to be returned by the algorithm but $t_{21}(\omega^{f-1})$ can overwrite \hat{t} . In total, we require

$$2+2+2(f-1)+(f-1)+2(f-1)=5f-1$$

instructions, and

$$[3 + (f - 2)] + |Y| = f + 14$$

Algorithm 2:

memory slots.

(ii) Calls to Algorithm 2

Having computed the T_2 , we begin the main 'for' loop of Algorithm 1, running through the columns of g in reverse order. Observe that r takes each value $1, \ldots, d$ exactly once as we run through the columns of g, because we are reducing the nonsingular matrix g to a monomial matrix. If we are in the (unique) column where r = d then there is no 'column clearing' to do and we skip straight to the row clearing stage. For each other column, we start by calling the subroutine FIRSTTRANSVECTIONS[r] (Algorithm 2). The role of this subroutine is to multiply the matrix g on the left by the transvection

$$t_{(r+1)r}(-g_{(r+1)c}g_{rc}^{-1}),$$

thereby making the (r+1,c)-entry of g zero. If $r \geq 2$, it is necessary to first compute the $t_{(r+1)r}(\omega^{\ell})$ (if r=1 then these are already stored in memory). The cost of an MSLP for calls to FIRSTTRANSVECTIONS[r] is as follows.

Lemma 4.6 Let $\lambda = f(2r-1) + 2\log_2(q) + 2$ and b = f + 17. There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the input of Algorithm 2 then S returns memory containing the output of Algorithm 2.

Proof. The nested 'for' loop in FIRSTTRANSVECTIONS[r] computes the set of transvections T_{r+1} in 2f(r-1) instructions (matrix operations). It does not increase the size

$$|T_2| + |Y(w, u_1, u_2)| = f + 13$$

of the input memory, because the recursive formula for the T_i allows each T_i to overwrite T_{i-1} . If we assume that $-g_{(r+1)c}g_{rc}^{-1}$ is known as a polynomial of

Algorithm 3:

```
LEFTUPDATE[i] Input: T_{i-1} \cup \{t_{(i-1)r}(1)\} \cup Y(w, u_1, u_2);

Output: T_i \cup \{t_{ir}(1)\} \cup Y(t_{ir}(-g_{ic}g_{rc}^{-1})w, t_{ir}(-g_{ic}g_{rc}^{-1})u_1, u_2);

for \ell = 0, ..., f-1 do

\mid \text{ compute } t_{i(i-1)}(\omega^{\ell}) = vt_{(i-1)(i-2)}(\omega^{\ell})v^{-1};

end

compute t_{i(i-1)}(-g_{ic}g_{rc}^{-1});

compute t_{ir}(-g_{ic}g_{rc}^{-1}) = [t_{i(i-1)}(-g_{ic}g_{rc}^{-1}), t_{(i-1)r}(1)];

compute t_{ir}(1) = [t_{i(i-1)}(1), t_{(i-1)r}(1)];

w := t_{ir}(-g_{ic}g_{rc}^{-1})w;

u_1 := t_{ir}(-g_{ic}g_{rc}^{-1})u_1;
```

degree (at most) f-1 in the primitive element $\omega \in \mathbb{F}_p$, then we can construct the transvection $t_{(r+1)r}(-g_{(r+1)c}g_{rc}^{-1})$ from T_{r+1} in at most $2\log_2(q)+f$ instructions, according to Lemma 4.3. A further two instructions are then required to multiply w and u_1 by this transvection to return the required final memory. In addition to the existing f+13 memory slots already in use, a further three slots are required while computing $t_{(r+1)r}(-g_{(r+1)c}g_{rc}^{-1})$, according to Lemma 4.3 (because the transvections to be powered up and multiplied together are already in memory), and then one more slot is needed to store this transvection prior to multiplying it by w and u_1 . So the MSLP requires at most

$$2f(r-1) + (2\log_2(q) + f) + 2 = f(2r-1) + 2\log_2(q) + 2$$

instructions and at most (f + 13) + 4 = f + 17 memory slots.

(iii) Calls to Algorithm 3

At this point in each pass of the main 'for' loop of Algorithm 1, we call the subroutine Leftupdate[i] for $i=r+2,\ldots,d$, unless $r\geq d-1$, in which case the current column will have already been cleared. The role of this subroutine is to effect the elementary row operations necessary to clear the rest of the current column (one entry of g is made zero with each call to the subroutine). The cost of an MSLP for Algorithm 3 is as follows.

Lemma 4.7 Let $\lambda = 2\log_2(q) + 3f + 10$ and b = f + 18. There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the input of Algorithm 3 then S returns memory containing the output of Algorithm 3.

Proof. The 'for' loop computes T_i in 2f instructions, and does not increase the input memory requirement of f+1+|Y|=f+14 slots because T_i can overwrite T_{i-1} . With $-g_{ic}g_{rc}^{-1}$ assumed given as a polynomial in ω , the transvection $t_{i(i-1)}(-g_{ic}g_{rc}^{-1})$ is constructed from T_i in at most $2\log_2(q)+f$ matrix operations,

Algorithm 4:

with an additional memory requirement of three slots plus the one slot required to store $t_{i(i-1)}(-g_{ic}g_{rc}^{-1})$ for the next step. So, by this point, the subroutine has needed at most

$$(f+14)+3+1=f+18 (13)$$

memory slots at any one time. The transvection $t_{ir}(-g_{ic}g_{rc}^{-1})$ is formed by computing the commutator of $t_{i(i-1)}(-g_{ic}g_{rc}^{-1})$ and $t_{(i-1)r}(1)$. According to Section 2.2(i), this takes four instructions, but does not increase the memory requirement (13) because one of the slots used to compute $t_{i(i-1)}(-g_{ic}g_{rc}^{-1})$ can now be overwritten while computing the commutator, which only requires one slots (plus the two inputs). The commutator $t_{ir}(1) = [t_{(i-1)r}(1), t_{i(i-1)}(1)]$ is then computed and replaces $t_{(i-1)r}(1)$ in memory, taking another four instructions but again not increasing the memory requirement (13). Finally, the matrices w and u_1 are multiplied by $t_{ir}(-g_{ic}g_{rc}^{-1})$ in two instructions without adding to (13). So our claimed value for b is given by (13), and our b-MSLP has maximal length $2f + (2\log_2(q) + f) + 4 + 4 + 2 = 2\log_2(q) + 3f + 10$.

Note that on the first pass of LeftUpdate[i], namely when i = r + 2, the element $t_{(i-1)r}(1) = t_{(r+1)r}(1)$ is already contained in $T_{i-1} = T_{r+1}$. A copy of this element would be made at this point to form the required input.

(iv) Calls to Algorithms 4 and 5

Once Algorithm 3 has been called the required number of times, the cth column of g is clear and main 'for' loop of Algorithm 1 moves on to the row clearing stage for the r(c)th row. This is accomplished by Algorithms 4 and 5. The former makes the (c-1,r)-entry of g zero by multiplying g on the right by $t_{c(c-1)}(-g_{r(c-1)}g_{rc}^{-1})$, after first computing the transvections comprising T_c (if $c \neq d$). The latter clears the rest of the rth row by multiplying g by the appropriate transvections. The costs of MSLPs for these subroutines are evidently the same as for Algorithms 2 and 3, respectively. We summarise:

Algorithm 5:

```
RIGHTUPDATE[j] Input: T_{j+2} \cup \{t_{c(j+1)}(1)\} \cup Y(w, u_1, u_2);

Output: T_{j+1} \cup \{t_{cj}(1)\} \cup Y(wt_{cj}(-g_{rj}g_{rc}^{-1}), u_1, u_2t_{cj}(-g_{rj}g_{rc}^{-1}));

for \ell = 0, \dots, f-1 do

\int t_{(j+1)j}(\omega^{\ell}) := v^{-1}t_{(j+2)(j+1)}(\omega^{\ell})v for \ell = 0, \dots, f-1;

end

compute t_{(j+1)j}(-g_{rj}g_{rc}^{-1});

t_{cj}(-g_{rj}g_{rc}^{-1}) := [t_{c(j+1)}(1), t_{(j+1)j}(-g_{rj}g_{rc}^{-1})];

t_{cj}(1) := [t_{c(j+1)}(1), t_{(j+1)j}(1)];

w := wt_{cj}(-g_{rj}g_{rc}^{-1});

u_2 := u_2t_{cj}(-g_{rj}g_{rc}^{-1});
```

Lemma 4.8 Let $\lambda = f(2r-1) + 2\log_2(q) + 2$ and b = f + 17. There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the input of Algorithm 4 then S returns memory containing the output of Algorithm 4.

Lemma 4.9 Let $\lambda = 2\log_2(q) + 3f + 10$ and b = f + 18. There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the input of Algorithm 5 then S returns memory containing the output of Algorithm 5.

(v) Total length and memory quota for Algorithm 1

The subroutine FIRSTTRANSVECTIONS[r] (Algorithm 2) is run whenever $r \neq 1$, namely d-1 times, with each value $r \in \{2,\ldots,d\}$ occurring exactly once (as explained earlier). According to Lemma 4.6, each call to FIRSTTRANSVECTIONS[r] contributes

$$f(2r-1) + 2\log_2(q) + 2$$

instructions to our MSLP for Algorithm 1. Hence, in total, calls to this subroutine contribute

$$\sum_{r=2}^{d} (f(2r-1) + 2\log_2(q) + 2) = (d^2 - 1)f + (d-1)(2\log_2(q) + 2)$$
 (14)

instructions.

For each column c with $r(c) \leq d-2$, the subroutine LeftUpdate[i] is called for $r+2 \leq i \leq d$. Each call to this subroutine requires $2\log_2(q)+3f+10$ MSLP instructions, according to Lemma 4.7. So

$$(d-r-1)(2\log_2(q)+3f+10)$$

instructions are needed per column, yielding a total cost of

$$\sum_{r=1}^{d-2} (d-r-1)(2\log_2(q)+3f+10) = \frac{1}{2}(d-1)(d-2)(2\log_2(q)+3f+10)$$
(15)

instructions.

The total cost of all column clearing stages of Algorithm 1 is the sum of the expressions (14) and (15). For simplicity, let us note that this sum is less than

$$d^{2}\left(\log_{2}(q) + \frac{5f}{2} + 5\right) + 2d(\log_{2}(q) + 1). \tag{16}$$

As noted above, the row clearing stages, namely calls to Algorithms 4 and 5, contribute (at most as) much as the column clearing stages. So *twice* the quantity (16) is contributed to the maximum length of an MSLP for Algorithm 1. In addition, we must also include the cost of the initial computation of T_1 given by Lemma 4.5, namely 5f-1 instructions, and then two additional instructions in order to invert u_1 and u_2 before returning them (second- and third-last lines of Algorithm 1). So we see that we can construct an MSLP for Algorithm 1 with length at most

$$d^{2}(2\log_{2}(q) + 5f + 10) + 4d(\log_{2}(q) + 1) + (5f - 1) + 2.$$

The maximum memory quota b for such an MSLP is just the maximum of all the b values in Lemmas 4.5–4.9. This maximum is b = f + 18, from both Lemma 4.7 and Lemma 4.9. This completes the proof of Proposition 4.4.

4.3 Modifications for d even

Let us now explain the changes required when d is even. The main issue is that the formula (8) used to compute the sets of transvections T_i recursively throughout our implementation of the algorithm described by Taylor looks two steps back instead of one when d is even. That is, each T_i is computed from either T_{i-2} (while clearing a column) or T_{i+2} (while clearing a row), not from $T_{i\pm 1}$ as was the case for d odd. Therefore, an MSLP for the d even case needs to hold two of the sets T_i in memory at any given time, because a T_i cannot be overwritten until until it has been used to compute $T_{i\pm 2}$. This adds (at most) f memory slots to the maximum memory quota b = f + 18 of Proposition 4.4.

The other main change is that initially, namely for the first column clearing stage, the set of transvections T_3 must be computed from T_2 via the formula (7), whereas in the d odd case it is computed via (8). This adds only one extra MSLP instruction, in order to form and store the element xv^{-1} needed in the conjugate on the right-hand side of (7) (this element can later be overwritten and so does not add to the overall maximum memory quota; recall also that x is no longer the identity when d is odd). Observe that the formula (6) differs from the d odd case

only in the sense that v is replaced by x^{-1} , and hence the initial computation of T_2 requires the same number of instructions and memory slots as before. The sets T_2 and T_3 are computed as described above in preparation for the first column clearing stage, but are subsequently computed via the recursion (8) (with increased memory quota relative to the d odd case, as already explained).

Although the described modifications are not complicated in and of themselves, they would introduce noticeable complications into our pseudocode and hence we have chosen to separate the d even case for the sake of clearer exposition, opting to simply point out and explain the changes instead of writing them out in detail. Since the d even case is slightly more costly than the d odd case in terms of both the length (one extra instruction) and required memory (f extra slots) for our MSLP, we conclude by extending Proposition 4.4 thusly:

Proposition 4.10 Let $q = p^f$ with p prime and $f \ge 1$, let d be an integer with $d \ge 2$, and set b = 2f + 18 and

$$\lambda = d^2(2\log_2(q) + 5f + 10) + 4d(\log_2(q) + 1) + (5f + 2).$$

There exists a b-MSLP, S, of length at most λ such that if S is evaluated with memory containing the list Y(g, 1, 1) in (12) then S outputs memory containing $Y(w, u_1, u_2)$ with $g = u_1wu_2$ the Bruhat decomposition of g.

Proof of Theorem 1.1. Combing Proposition 4.10 with Propositions 3.1 and 3.4 yields the theorem: the maximum memory quota at any point is the 2f + 18 memory slots needed in Proposition 4.10, while the lengths of the MSLPs in each case are $O(d^2 \log(q))$.

5 Implementation

Our algorithm has been implemented in the computer algebra system GAP [6]. We have tested our implementation on matrices of various sizes over finite fields.

For example, computing the Bruhat decomposition of a random matrix in GL(250,2) resulted in an SLP of length 525 394. During the evaluation, our MSLP required 25 memory slots and it was easily possible to evaluate this MSLP on the standard generators of GL(250,2). However, it was not possible to evaluate this SLP directly in GAP by storing 525 394 matrices in GL(250,2), as this required too much memory. (We remark that the number of memory slots used, 25, is slightly higher than the maximum of 2f + 18 = 22 slots asserted in Theorem 1.1, because we stored a few extra group elements for convenience.)

We note that after applying the function SLOTUSAGEPATTERN, the resulting SLP only required 12 memory slots and could be evaluated in the same time as our MSLP. This is due to the fact that SLOTUSAGEPATTERN was handed a well-designed SLP. When faced with an SLP not designed to be memory efficient, one might not expect such drastic improvements.

Acknowledgements

We thank Max Neunhöffer for his assistance and advice on the SLP functionality in GAP.

References

- [1] H. Bäärnhielm and C. R. Leedham-Green, "The product replacement prospector", J. Symb. Comput. 47 (2012) 64–75.
- [2] L. Babai and E. Szemerédi, "On the complexity of matrix group problems I", in: 25th Annual Symposium on Foundations of Computer Science, 1984, (IEEE, 1984),229–240.
- [3] C. E. Borges, C. L. Alonso, J. L. Montana, M. De La Cruz Echeandía, and A. O. De La Puente, "Coevolutionary architectures with straight line programs for solving the symbolic regression problem", in: *ICEC 2010 Proceedings of the International Conference on Evolutionary Computation 2010*, 41–50.
- [4] P. Bürgisser, M. Clausen and M. A. Shokrollahi, Algebraic complexity theory, volume 315 of Grundlehren der Mathematischen Wissenschaften, (Springer-Verlag, Berlin, 1997).
- [5] F. Claude and G. Navarro, "Self-indexed text compression using straight-line programs", *Proc. MFCS'09* (2009) 235–246.
- [6] The GAP Group, GAP Groups, Algorithms, and Programming, Version 4.6.2; 2013, (\protect\vrule width0pt\protect\href{http://www.gap-system.org}{http://www.gap-system.org}
- [7] W. M. Kantor and Á Seress, Black box classical groups, volume 149 of Memoirs of the American Mathematical Society (AMS, 2001).
- [8] C. R. Leedham-Green, "The computational matrix group project", in: Groups and computation, III (Columbus, OH, 1999), volume 8 of Ohio State Univ. Math. Res. Inst. Publ., (de Gruyter, Berlin, 2001), 229–247.
- [9] C. R. Leedham-Green and E. A. O'Brien, "Constructive recognition of classical groups in odd characteristic", J. Algebra **322** (2009) 833–881.
- [10] N. A. Lynch, "Straight-line program length as a parameter for complexity analysis", *J. Comput. Syst. Sci.* **21** (1980) 251–280.
- [11] W. Bosma, J. Cannon and C. Playoust, "The Magma algebra system. I. The user language", J. Symbolic Comput. 24 (1997) 235–265.
- [12] M. Neunhöffer and Á Seress, "A data structure for a uniform approach to computations with finite groups", in: *ISSAC 2006*, (ACM, New York, 2006), 254–261.

- [13] E. A. O'Brien, "Algorithms for matrix groups", in: *Groups St. Andrews* 2009 in Bath. Volume 2, volume 388 of London Math. Soc. Lecture Note Ser., (Cambridge University Press, Cambridge, 2011), 297–323.
- [14] Á Seress, Permutation group algorithms, volume 152 of Cambridge Tracts in Mathematics (Cambridge University Press, Cambridge 2003).
- [15] C. S. Sims, "Computational methods in the study of permutation groups", in: Computational Problems in Abstract Algebra, (Pergamon, Oxford, 1970), pp. 169–183.
- [16] G. Strang, "Triangular factorizations: The algebra of elimination", preprint, 2011.
- [17] D. E. Taylor, The geometry of the classical groups, volume 9 of Sigma Series in Pure Mathematics (Heldermann Verlag, Berlin, 1992).