A linear-time algorithm for the orbit problem over cyclic groups

Anthony W. Lin[1] and Sanming Zhou[2]

[1] Yale-NUS College, Singapore
[2] Department of Mathematics and Statistics, The University of Melbourne, Australia

Abstract. The orbit problem is at the heart of symmetry reduction methods for model checking concurrent systems. It asks whether two given configurations in a concurrent system (represented as finite sequences over some finite alphabet) are in the same orbit with respect to a given finite permutation group (represented by their generators) acting on this set of configurations. It is known that the problem is in general as hard as the graph isomorphism problem, which is widely believed to be not solvable in polynomial time. In this paper, we consider the restriction of the orbit problem when the permutation group is cyclic (i.e. generated by a single permutation), an important restriction of the orbit problem. Our main result is a linear-time algorithm for this subproblem.

1 Introduction

Since the inception of model checking, a key challenge in verifying concurrent systems has always been how to circumvent the state explosion problem due to the growth in the number of processes. Among others, symmetry reduction [10, 14, 18] has emerged to be an effective technique in combatting the state explosion problem. The essence of symmetry reduction is to identify symmetries in the system and avoid exploring states that are "similar" (under these symmetries) to previously explored states, thereby speeding up model checking.

Every symmetry reduction method has to deal with the following problems: (1) how to identify symmetries in the given system, and (2) how to check that two configurations are similar under these symmetries. For concurrent systems with n processes, Problem 1 amounts to searching for a group G of permutations on $[n] := \{1, \ldots, n\}$ such that the system behaves in an identical way under the action of permuting the indices of the processes by any $\pi \in G$. For example, for a distributed protocol with a ring topology, the group G could be a *rotation group* generated by the "cyclical right shift" permutation RS that maps $i \mapsto i+1$ mod n for each $i \in [n]$. The reader is referred to the recent survey [23] for more detailed discussions and techniques for handling Problem 1, a computationally difficult problem in general. Now the group G partitions the state space of the concurrent system (i.e. Γ^n for some finite set Γ) into equivalence classes called *(G-)orbits*. Problem 2 is essentially the *orbit problem (over finite permutation groups)*: given G and two configurations $\mathbf{v}, \mathbf{w} \in \Gamma^n$, determine whether $\mathbf{v}$ and $\mathbf{w}$

are in the same G-orbit. For example, if G is generated by RS with $n = 4$, the two configurations $(1, 1, 0, 0)$ and $(0, 0, 1, 1)$ are in the same orbit.

The orbit problem (OP) was first studied in the context of model checking by Clarke *et al.* [10] in which it was shown to be in NP but is as hard as the graph isomorphism problem, which is widely believed to be not solvable in polynomial time. The difficulty of the problem is due to the fact that the input group G is represented by a set S of generators and that the size of G can be exponential in $|S|$ in the worst case. There is also a closely related variant of OP called the *constructive orbit problem (COP)*, which asks to compute the lexicographically smallest element $\mathbf{w} \in \Gamma^n$ in the orbit of a given configuration $\mathbf{v} \in \Gamma^n$ with respect to a given group G. OP is easily reducible to COP, though the reverse direction is by no means clear. COP was initially studied in the context of graph canonisation by Babai and Luks [3], in which COP was shown to be NP-hard (in contrast, whether OP is NP-hard is open). In the context of model checking, COP was first studied by Clarke *et al.* [9], in which a number of "easy groups" for which COP becomes solvable in P are given including polynomial-sized groups (e.g. rotation groups), the full symmetry group S_n (i.e. containing all permutations on $[n]$), and disjoint/wreath products of easy groups (cf. [13]).

In this paper, we consider the orbit problem over *cyclic groups* (i.e. generated by a single permutation $\pi \in S_n$), which is an important subproblem of OP. Firstly, an algorithm for this subproblem has immediate applications for OP in the general case. For example, given a permutation group G with generators $\pi_1, \ldots, \pi_k$, we can check if the two configurations $\mathbf{v}$ and $\mathbf{w}$ are in the same orbit of the cyclic subgroup generated by any one of π_j. [If yes, then $\mathbf{v}$ and $\mathbf{w}$ are also in the same G-orbit.] It is also possible to combine cyclic groups with other easy groups from [9] via disjoint/wreath product operators. Secondly, it subsumes a commonly occurring class of symmetries for concurrent systems: the rotation groups. Unlike the case of rotation groups however, the size of cyclic groups can be exponential in n (see Proposition 3 below), which rules out a naive enumeration of the group elements. Finally, OP over cyclic groups is intimately connected to the classical orbit problem over rational matrices [19]: given a rational n-by-n matrix M and two rational vectors $\mathbf{v}, \mathbf{w} \in \mathbb{Q}^n$, determine if there exists $k \in \mathbb{N}$ such that $A^k\mathbf{v} = \mathbf{w}$. In fact, they coincide when M is restricted to *permutation matrices* [6], i.e., 0-1 matrices with precisely one column for each row with entry 1. To see this, given a permutation π on $[n]$, simply take an n-by-n 0-1 matrix $A = (A[i, j])_{1 \leq i,j \leq n}$ such that $A[i, j] = 1$ iff $\pi(j) = i$. The reverse direction is similar. That OP over cyclic groups is in P follows from Kannan and Lipton's celebrated result [19] that OP over rational matrices is in P.

Contributions. In this paper, we provide an algorithm for the orbit problem over cyclic groups that is simpler than Kannan-Lipton's algorithm [19] and moreover runs in linear-time in the standard RAM model. To this end, we provide a linear-time reduction to the problem of solvability of systems of linear congruence equations. The reduction is simple though it exploits subtle connections to the string searching problem and number-theoretic results like the Erdös-Graham Lemma [15] concerning solutions of Diophantine equations.

As for the solvability of systems of linear congruence equations, we start off with an algorithm that runs in linear-time assuming constant-time integer arithmetic operations. However, when we measure the number of bit operations (i.e. bit complexity model), it turns out that the algorithm runs in time cubic in the number of equations in the systems. To address this issue, we restrict the problem to input instances provided by our reduction from the orbit problem. We offer two solutions. Firstly, we show that the average-case complexity of the algorithm under the bit complexity model is $O(\log^5 n)$, which is sublinear. Secondly, we provide another algorithm that uses at most linearly many bit operations *in the worst case* (though on average it is worse than the first algorithm).

Organisation. Section 2 contains definitions and basic concepts. We provide our first algorithm for solving systems of linear congruence equations in Section 3 (Algorithm 1), while we provide our linear-time reduction from the orbit problem to equations solving in Section 4 (Algorithm 2). Thus far, we assume that arithmetic operations take constant time. We deal with the issue of bit complexity in Section 5. We conclude with future work in Section 6.

Acknowledgment. We thank the anonymous referees for their helpful feedback. Part of the work was done when Lin was at Oxford supported by EPSRC (H026878). Zhou was supported by ARC (FT110100629).

2 Preliminaries

General Notations: We use log (resp. ln) to denote the logarithm function in base 2 (resp. natural logarithm). We use the standard interval notations to denote a subset of integers within that interval. For example, $[i, j)$ denotes the set $\{k \in \mathbb{Z} : i \leq k < j\}$. Likewise, for each positive integer n, we use $[n]$ to denote the set $\{1, \ldots, n\}$. We shall also extend arithmetic operations to sets of numbers in the usual way: whenever $S_1, S_2 \subseteq \mathbb{Z}$, we define $S_1 + S_2 := \{s_1 + s_2 : s_1 \in S_1, s_2 \in S_2\}$ and $S_1 S_2 := \{s_1 \times s_2 : s_1 \in S_1, s_2 \in S_2\}$. In the context of arithmetic over $2^{\mathbb{Z}}$, we will treat a number $n \in \mathbb{N}$ as the singleton set $\{n\}$. That way, for $a, b \in \mathbb{N}$, the notation $a + b\mathbb{Z}$ refers to the *arithmetic progression* $\{a + bc : c \in \mathbb{Z}\}$, where a (resp. b) is called the *offset* (resp. *period*) of the arithmetic progression. Likewise, for a subset $S \subseteq \mathbb{N}$, we use $\gcd(S)$ to denote the greatest common divisor of S.

We will use standard notations from formal language theory. Let Γ be an *alphabet* whose elements are called *letters*. A word (or a string) w over Γ is a finite sequence of elements from Γ. We use Γ^* to denote the set of all words over Γ. The length of w is denoted by $|w|$. Given a word $w = a_1 \ldots a_n$, the notation $w[i, j]$ denotes the subword $a_i \ldots a_j$. For a sequence $\sigma = i_1, \ldots, i_k \in [n]^*$ of *distinct* indices of w, we write $w[\sigma]$ to denote the word $a_{i_1} \ldots a_{i_k}$. We also define $\mathrm{RS}(w)$ to be $a_n a_1 a_2 \ldots a_{n-1}$, i.e., the word w cyclically right-shifted.

Number Theory: We will use the following basic result (cf. [11]).

Proposition 1 (Chinese Remainder Theorem). *Let $n_1, \ldots, n_k$ be pairwise relatively prime positive integers, and $n = \prod_{i=1}^{k} n_i$. The ring $\mathbb{Z}_n$ and the direct*

product of rings $\mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}$ *are isomorphic under the function* $\sigma : \mathbb{Z} \to \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}$ *with* $\sigma(x) := (x \mod n_1, \ldots, x \mod n_k)$ *for each* $x \in \mathbb{Z}$.

Groups: We briefly recall basic concepts from group theory and permutation groups (cf. see [7]). A *group* G is a pair $(S, \cdot)$, where S is a set and $\cdot : (S \times S) \to S$ is a binary operator satisfying: (i) associativity (i.e. $g_1 \cdot (g_2 \cdot g_3) = (g_1 \cdot g_2) \cdot g_3$), (ii) the existence of a (unique) identity element $e \in S$ such that $g \cdot e = e \cdot g = g$ for all $g \in S$, and (iii) closure under inverse (i.e. for each $g \in G$, there exists $g^{-1} \in G$ such that $g \cdot g^{-1} = g^{-1} \cdot g = e$). When it is clear from the context, we will write $g \cdot g'$ as gg'. The *order* $\mathrm{ord}(G)$ of the group G is defined to be $|S|$. This paper concerns only finite groups, i.e., groups G with $\mathrm{ord}(G) = |S| \in \mathbb{N}$. For each $n \in \mathbb{N}$, we define g^n by induction: (i) $g^0 = e$, and (ii) $g^n = g^{n-1} \cdot g$. The *order* $\mathrm{ord}(g)$ of $g \in G$ is the least positive integer n such that $g^n = e$.

A *subgroup* H of $G = (S, \cdot)$ (denoted as $H \leq G$) is any group $(S', \cdot_H)$ such that $S' \subseteq S$ and $\cdot_H$ and $\cdot$ agree on S'. Given any subset $X \subseteq S$, the subgroup of G *generated* by X is defined to be the subgroup $\langle X \rangle := (S', \cdot_h)$ of G each of whose elements can be expressed as a finite product of elements of X and their inverses. If $H = \langle X \rangle$, then X is said to *generate* H. A *cyclic group* is a group generated by a singleton set $X = \{g\}$.

An *action* of a group $G = (S, \cdot)$ on a set Y is a function $\times : S \times Y \to Y$ such that for all $g, h \in S$ and $y \in Y$: (1) $(gh) \times y = g \times (h \times y)$, and (2) $e \times y = y$. The (G-)orbit containing y, denoted Gy, is the subset $\{g \times y : g \in G\}$ of Y. The action $\times$ partitions the set Y into G-orbits. When the meaning is clear, we shall omit mention of the operator $\times$, e.g, condition (2) above becomes $ey = y$.

Permutation Groups. A *permutation* on $[n]$ is any bijection $\pi : [n] \to [n]$. The set of all permutations on $[n]$ forms the *(nth) full symmetry group* S_n under functional composition. We shall use the notation Id to denote the identity element of each S_n. A word $w = a_0 \ldots a_{k-1} \in [n]^*$ containing distinct elements of $[n]$ (i.e. $a_i \neq a_j$ if $i \neq j$) can be used to denote the permutation that maps $a_i \mapsto a_{i+1 \mod k}$ for each $i \in [0, k)$ and fixes other elements of $[n]$. In this case, w is called a *cycle*, which we will often write in the standard notation $(a_0, \ldots, a_{k-1})$ so as to avoid confusion. Observe that w and $\mathrm{RS}(w)$ represent the same cycle c. We will however fix a particular ordering to represent c (e.g. the word provided as input to the orbit problem). For this reason, if $\mathbf{v} \in \Gamma^n$ for some alphabet Γ, the notation $\mathbf{v}[c]$ is well-defined (see General Notations above), which means projections of $\mathbf{v}$ onto elements with indices in c, e.g., if $\mathbf{v} = (1, 1, 1, 0)$ and $c = (1, 4, 2)$, then $\mathbf{v}[c] = (1, 0, 1)$. Any permutation can be written as a composition of disjoint cycles [7]. Each subgroup $G = (S, \cdot)$ of S_n acts on the set Γ^n (over any finite alphabet Γ) under the group action of permuting indices, i.e., for each $\pi \in S$ and $\mathbf{v} = (a_1, \ldots, a_n) \in \Gamma^n$, we define $\pi\mathbf{v} := (a_{\pi(1)}, \ldots, a_{\pi(n)})$.

Complexity Analysis: We will assume that permutations will be given in the input as a composition of disjoint cycles. It is easy to see that permutations can be converted back and forth in linear time from such representations and the representations of permutations as functions. The size $\|n\|$ of a number $n \in \mathbb{N}$ is defined to be the length of the binary representation of n, which is $\lfloor \log n \rfloor + 1$. The size $\|c\|$ of a cycle $c = (a_1, \ldots, a_k)$ on $[n]$ is defined to be $\sum_{i=1}^{k} \|a_i\|$ (in

contrast, the length $|c|$ of c is k). For a permutation $\pi = c_1 \cdots c_m$ where each c_i is a cycle, the size $\|\pi\|$ of π is defined to be $\sum_{i=1}^{m} \|c_i\|$. We will use standard asymptotic notations from analysis of algorithms (big-O and little-o), cf. [11]. We also use the standard $\sim$ notation: $f(n) \sim g(n)$ iff $\lim_{n\to\infty} f(n)/g(n) = 1$. We will use the standard RAM model that is commonly used when analysing the complexity of algorithms (cf. [11]). In Sections 3 and 4, we will assume that integer arithmetic takes constant time. Later in Section 5, we will use the *bit complexity model* (cf. [11]), wherein the running time is measured in the number of bit operations.

3 Solving a system of modular arithmetic equations

Recall that a linear congruence equation is a relation of the form $x \equiv a \pmod{b}$, where $a, b \in \mathbb{N}$, whose solution set is denoted by $[\![x \equiv a \pmod{b}]\!] = a + b\mathbb{Z}$. A system of linear congruence equations is a relation of the form $\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}$. The set of solutions $x \in \mathbb{Z}$ to this system is denoted by $[\![\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}]\!]$, which equals $\bigcap_{i=1}^{m} [\![x \equiv a_i \pmod{b_i}]\!]$. The system is *soluble / solvable* if the solution set is nonempty. We use FALSE to denote $x \equiv 0 \pmod{2} \wedge x \equiv 1 \pmod{2}$, which is not solvable. The following proposition provides a fast symbolic method for computing solutions to systems of linear congruences.

Proposition 2. *For any solvable system of linear congruence equations* $\varphi(x) := \bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}$*, we have* $[\![\varphi(x)]\!] = [\![x \equiv a \pmod{b}]\!]$ *for some* $a, b \in \mathbb{Z}$*. Furthermore, there exists an algorithm which computes* a, b *in linear time.*

This proposition is in fact a rather easy corollary of the following result in algorithmic number theory about solving more general linear congruence equations of the form $ax \equiv b \pmod{n}$.

Lemma 1 (Linear Congruence Theorem; see [11, Chapter 31.4]). *The equation* $ax \equiv b \pmod{n}$ *is solvable for the unknown* x *iff* $d|b$*, where* $d = \gcd(a, n)$*. Furthermore, if it is solvable, then the set of solutions equals* $x_0 + (n/d)\mathbb{Z}$*, for some* $x_0 \in [0, n/d)$ *that can be computed in time* $O(\log n)$*.*

This algorithm made use of the Extended Euclidean algorithm, which explains the $O(\log n)$ time complexity (see [11]). Algorithm 1 witnesses the linear-time algorithm claimed in Proposition 2. The algorithm sequentially goes through each equation $x \equiv a_i \pmod{b_i}$, while keeping the solution to the subsystem $\bigwedge_{i=1}^{j} x \equiv a_i \pmod{b_i}$ at jth iteration as an arithmetic progression $a + b\mathbb{Z}$, for some $a, b \in \mathbb{Z}$. Before we go through any equation, the set of solutions to the empty system of equations is $a + b\mathbb{Z}$ with $a = 0$ and $b = 1$. At the jth iteration, we assume that $[\![\bigwedge_{i=1}^{j-1} x \equiv a_i \pmod{b_i}]\!] = a + b\mathbb{Z}$ for some $a, b \in \mathbb{Z}$. We replace x in the equation $x \equiv a_j \pmod{b_j}$ by $a + by$ for an unknown y, which results in the new equation $\varphi(x) := by \equiv a_i - a \pmod{b_i}$. Lemma 1 gives an answer to $[\![\varphi]\!]$ as either $\emptyset$ or $a' + b'\mathbb{Z}$, for some $a' \in [0, b_i)$ and $b' \in [1, b_i]$. We substitute this solution set back to x, which gives $[\![\bigwedge_{i=1}^{j} x \equiv a_i \pmod{b_i}]\!] = (a'b + a) + bb'\mathbb{Z}$, which justifies the assignments $a := a'b + a$ and $b := bb'$.

Algorithm 1 Solving a system of modular arithmetic equations

Input: A system of modular arithmetic equations $\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}$
Output: Solution set $[\![\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}]\!]$ as $\emptyset$ or an arithmetic progression $a+b\mathbb{Z}$.

```
a := 0; b := 1;
for i = 1, ..., m do
    φ(y) := by ≡ a_i − a (mod b_i);
    Apply algorithm from Lemma 1 on φ returning either ∅ or a′ + b′ℤ for [[φ]];
    if [[φ]] = ∅ then return NO else a := a′b + a; b := bb′ end if
end for
return a + bℤ;
```

As for the time complexity of the algorithm, at jth iteration the algorithm invokes the algorithm from Lemma 1, which runs in time $O(\log b_j)$. Therefore, the total running time of our algorithm is $O(\sum_{j=1}^{m} \log b_j)$, i.e., linear in the size $\sum_{j=1}^{m}(\log a_j + \log b_j)$ of the input.

Remark 1. The number of bits that is used to maintain a and b in the worst case is linear in the size $\sum_{j=1}^{m}(\log a_j + \log b_j)$ of the input. This justifies treating a single arithmetic operation as a constant-time operation. We will address the issue of bit complexity in Section 5.

4 Reducing to solving a system of linear congruence equations

In this section, we prove the main result of the paper.

Theorem 1. *There is a linear-time algorithm for solving the orbit problem when the acting group is cyclic.*

This algorithm is a linear-time reduction from the orbit problem over cyclic groups to solving a system of linear congruence equations, which will allow us to use results from the previous section.

Before we proceed to the algorithm, the following proposition shows why the naive algorithm that checks whether $g^i(\mathbf{v}) = \mathbf{w}$, for a given permutation $g \in S_n$ and for each $i \in [0, \mathrm{ord}(g))$, actually runs in exponential time.

Proposition 3. *There exists a sequence $\{G_i\}_{i=1}^{\infty}$ of cyclic groups $G_i = \langle g_i \rangle$ such that ord(g_i) is exponential in the size $\|g_i\|$ of the permutation g_i.*

Proof. Let p_n denote the nth prime. The *Prime Number Theorem* states that $p_n \sim n \log n$ (cf. [17]). For each $i \in \mathbb{Z}_{>0}$, we define a cycle c_i of length p_i by induction on i. For $i = 1$, let $c_1 = (1, 2)$. Suppose that $c_{i-1} = (j, \ldots, k)$. In this case, we define c_i to be the cycle $(k+1, \ldots, k+p_i)$. To define the sequence $\{g_i\}_{i=1}^{\infty}$ of permutations, simply let $g_i = \Pi_{j=1}^{i} c_i$. For example, we have $g_3 = (1, 2)(3, 4, 5)(6, 7, 8, 9, 10)$. Since c_i's are disjoint, the order $\mathrm{ord}(g_i)$ of g_i is the smallest positive integer k such that $c_j^k = \mathrm{Id}$ for all $j \in [i]$. If S_j denotes the set of integers k satisfying $c_j^k = \mathrm{Id}$, then $\mathrm{ord}(g_i)$ is precisely the smallest positive

integer in the set $\bigcap_{j=1}^{i} S_j$. It is easy to see that $S_j = p_j\mathbb{Z}$, which is the set of solutions to the linear congruence equation $x \equiv 0 \pmod{p_j}$. Therefore, by the Chinese Remainder Theorem (cf. Propositon 1), the set $\bigcap_{j=1}^{i} S_j$ coincides with the arithmetic progression $t_i\mathbb{Z}$ with $t_i := \prod_{j=1}^{i} p_j$. This implies that $\mathrm{ord}(g_i) = t_i$. Now the number t_i is also known as the *ith primorial number* [1] with $t_i \sim e^{(1+o(1))i \log i}$, which is a corollary of the Prime Number Theorem. On the other hand, the size of g_i is $\sum(i) := \sum_{j=1}^{i} p_i$, which is known to be $\sim \frac{1}{2}i^2 \ln i$ (cf. [4]). Therefore, $\mathrm{ord}(g_i)$ is exponential in $\|g_i\|$ as desired. □

Algorithm 2 Reduction to system of modular arithmetic equations

Input: A permutation $g = c_1 \cdots c_m \in S_n$, a finite alphabet Γ, and $\mathbf{v}, \mathbf{w} \in \Gamma^n$.
Output: A system of modular arithmetic equations, which is satisfiable iff $\exists i \in \mathbb{N} : g^i(\mathbf{v}) = \mathbf{w}$.

```
// First solve for each individual cycle
for all i = 1, ..., m do
    Compute the length |c_i| of the cycle c_i;
    Compute an ordered list S'_i ⊆ [0, |c_i|) of numbers r with c_i^r(v[c_i]) = w[c_i];
    if S'_i = ∅ then return FALSE end if
    if |S'_i| = 1 then let a_i be the member of S_i; b_i := |c_i|; end if
    if |S'_i| > 1 then a_i := min(S'_i); a'_i := min(S'_i \ {a_i}); b_i := a'_i − a_i; end if
end for
// Now for each i ∈ [1, m] we have a modular arithmetic equation x ≡ a_i (mod b_i)
return YES iff there exists x ∈ ℕ satisfying ⋀_{i=1}^{m} x ≡ a_i (mod b_i)
```

Our linear-time reduction that witnesses Theorem 1 is given in Algorithm 2. In this algorithm, the acting group is $G = \langle g \rangle$ with $g \in S_n$, expressed as a composition of disjoint cycles in a standard way, say, $g = c_1c_2 \cdots c_m$ where each c_i is a cycle. Also part of the input is two strings $\mathbf{v} = v_1 \ldots v_n, \mathbf{w} = w_1 \ldots w_n \in \Gamma^n$ over a finite alphabet Γ. The orbit problem is to check whether $f(v) = w$ for some $f \in G$, i.e., $f = g^r$ for some $r \in \mathbb{N}$. Since c_i's are pairwise disjoint cycles, the question reduces to checking if there exists $r \in \mathbb{N}$ such that

$$\forall i \in [1, m] : (c_i^r \mathbf{v})[c_i] = \mathbf{w}[c_i] \qquad (*)$$

In other words, for each $i \in [1, m]$, applying the action c_i^r to $\mathbf{v}$ gives us $\mathbf{w}$ when restricted to the indices in c_i. Essentially, Algorithm 2 sequentially goes through each cycle c_i and computes the set S_i of solutions r to $(c_i^r \mathbf{v})[c_i] = \mathbf{w}[c_i]$ as the set of solutions to the linear congruence equation $x \equiv a_i \pmod{b_i}$. Therefore, the set of solutions to (*) is precisely the set of solutions to the system of congruence equations $\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}$. In the following, we will provide the details of each individual step of Algorithm 2. We will also use the following running example to illustrate the algorithm: $c = (6, 5, 7, 3, 2, 1)$, $\mathbf{v} = \underline{010}0\underline{11}11$, and $\mathbf{w} = \underline{101}\underline{100}01$, where the positions in $\mathbf{v}$ and $\mathbf{w}$ that are modified by c are underlined.

Step 1: Computing the length of cycles. This is the same as how to compute the length of a list. Therefore, computing the length $|c_i|$ can be done in time $O(\|c_i\|)$.

Step 2: Computing representatives $S'_i \subseteq [0, |c_i|)$ for S_i. During this step, we collect a subset of numbers $h \in [0, |c_i|)$ such that $c_i^h(\mathbf{v}[c_i]) = \mathbf{w}[c_i]$. A quadratic algorithm for this is easy to come up with: sequentially go through $h \in [0, |c_i|)$ while computing the current c_i^h, and save h if $c_i^h(\mathbf{v}[c_i]) = \mathbf{w}[c_i]$ holds. One way to obtain a linear-time algorithm is to reduce our problem to the *string searching problem*: given a "text" $T \in \Sigma^*$ (over some finite alphabet Σ) and a "pattern" $P \in \Sigma^*$, find all positions i in T such that $T[i, i + |P|] = P$. This problem is solvable in linear-time by Knuth-Morris-Pratt (KMP) algorithm (e.g. see [11]).

We now show how to reduce our problem to the string searching problem in linear time. Suppose that $c := c_i = (j_1, \ldots, j_k)$. We have $\mathbf{v}[c] = v_{j_1} \ldots v_{j_k}$ and $\mathbf{w}[c] = w_{j_1} \ldots w_{j_k}$.

Lemma 2. $(c\boldsymbol{v})[c] = \mathrm{RS}(\boldsymbol{v}[c])$.

In other words, if $\textsc{Dom}(c) = \{j_1, \ldots, j_k\}$, the effect of c on $\mathbf{v}$ when restricted to $\textsc{Dom}(c)$ coincides with applying a cyclical right shift on the string $[c]$. Following our running example, it is easy to check that $[c] = 101010$ and $(c\mathbf{v})[c] = \mathrm{RS}(\mathbf{v}[c]) = 010101$.

Proof (of Lemma 2). Let $\mathbf{u} = u_1 \ldots u_k := (c\mathbf{v})[c]$ and $\mathbf{u} = u'_1 \ldots u'_k := \mathrm{RS}(\mathbf{v}[c])$. It suffices to show that $u_t = u'_t$ for all $t \in \mathbb{Z}_k$. By definition of RS, it follows that $u'_t = v_{j_{t-1}}$. Now suppose that $\mathbf{v}' = v'_1 \ldots v'_n := c\mathbf{v}$. Then

$$v'_j := \begin{cases} v_j & \text{if } j \notin \textsc{Dom}(c) \\ v_{j'} & \text{if } j \in \textsc{Dom}(c) \text{ and, for some } t \in \mathbb{Z}_k,\ j = j_{t+1} \text{ and } j' = j_t. \end{cases}$$

So, we have $u_t = ((c\mathbf{v})[c])[t] = (\mathbf{v}'[c])[t] = v'_{j_t} = v_{j_{t-1}}$. This proves that $u_t = u'_t$. □

Lemma 3. *For each $r \in \mathbb{N}$, we have $(c^r\boldsymbol{v})[c] = \mathrm{RS}^r(\boldsymbol{v}[c])$.*

Lemma 3 can easily be proven by induction using Lemma 2 (see full version). Lemma 3 implies that the set $S := S_i \subseteq \mathbb{N}$ of solutions r to the equation $(c_i^r\mathbf{v})[c_i] = \mathbf{w}[c_i]$ is a finite union of arithmetic progressions of the form $a + k\mathbb{Z}$, where $k = |c_i|$ and $a \in [0, k)$. This is simply because $\mathrm{RS}^{r+k}(\mathbf{v}[c_i]) = \mathrm{RS}^r(\mathbf{v}[c_i])$. We will finitely represent S by the offsets a's and the unique period k in these arithmetic progressions.

We now show how to compute the offsets for S in linear time by a linear-time reduction to the string searching problem. Define the text $T := \mathbf{v}[c]\mathbf{v}[c]$ and the pattern $P := \mathbf{w}[c]$. Observe that, for each $r \in [0, k)$, P is matched at position r in T iff $\mathrm{RS}^{r-1}(\mathbf{v}[c]) = \mathbf{w}[c]$. Therefore, after running the KMP algorithm with the solution set S', the offsets for S will be $\{r - 1 : r \in S'\}$. Solvability for each individual equation amounts to checking that, for each cycle c_i, the set S_i of solutions for the corresponding equation is nonempty.

Example 1. Continuing with our running example, it follows that $T = \mathbf{v}[c]\mathbf{v}[c] = 101010101010$ and $P = \mathbf{w}[c] = 010101$. We see that P matches T at positions $S' = \{2, 4, 6\}$. This implies that the set S of solutions $r \in \mathbb{Z}$ to the equation $(c^r\mathbf{v})[c] = \mathbf{w}[c]$ is $(1 + 6\mathbb{Z}) \cup (3 + 6\mathbb{Z}) \cup (5 + 6\mathbb{Z})$. ■

Observe that, for each c_i, this step takes time $O(\|c_i\|)$. Therefore, going through all the c_i's, this step takes time $\sum_{i=1}^{m} O(\|c_i\|) = O\left(\sum_{i=1}^{m} \|c_i\|\right) = O(\|g\|)$, i.e., linear in input size.

Step 3: Representing S_i as a single arithmetic progression. In the previous step, we have computed the representatives for S_i in $[0, |c_i|)$. This only shows that S_i is a finite union of arithmetic progressions, which cannot in general be expressed as the set of solutions to a linear congruence equation. In this step, we show that S_i can be represented as a single arithmetic progression and furthermore justify why the last three lines in Algorithm 2 computes S_i.

Lemma 4 (Normal Form). *For each $i = 1, \ldots, m$, either $S_i = \emptyset$ or $S_i = a_i + b_i\mathbb{Z}$ for some $a_i, b_i \in [0, |c_i|)$ where b_i divides $|c_i|$. In the case when $|S_i'| > 1$, we have $a_i = p_1$ and $b_i = p_2 - p_1$, where $p_1 < p_2$ are the smallest numbers in S_i'. Furthermore, we may compute the pair (a_i, b_i) of numbers in time $O(\|c_i\|)$.*

To prove this lemma, we will use the following number-theoretic result by Erdös and Graham [15]. [Also see the formulation in [8, 22], in which the result was applied in automata theory.]

Proposition 4. *Let $0 < p_1 < \ldots < p_s \leq k$ be natural numbers. Then, the set $X := \{\sum_{i=1}^{s} p_i x_i : x_1, \ldots, x_s \in \mathbb{N}\} \subseteq \mathbb{N}$ coincides with the set $S \cup (a + b\mathbb{N})$, where $S \subseteq \mathbb{N}$ contains no numbers bigger than k^2, and a is the least integer bigger than k^2 that is a multiple of $b := \gcd(p_1, \ldots, p_s)$.*

Proof (of Lemma 4). We use the shorthand S (resp. c) for S_i (resp. c_i). From Step 2, we know that S is a union of arithmetic progressions $\bigcup_{j=1}^{s} (p_j + k\mathbb{Z})$, for some $p_j \in [0, k)$ and $k = |c|$. Without loss of generality, we assume that $p_1 < \cdots < p_s$. If $s \in \{0, 1\}$, then we are done. Suppose now that $s > 1$. Let $\mathbf{v}[c] = d_1 \ldots d_k$ and $\mathbf{w}[c] = d_1' \ldots d_k'$. In this case, thanks to Lemma 3, it is the case that for each $j \in [1, s]$ and $l \in [1, k]$, we have $d_{l+p_j \mod k} = d_l'$. Let $\Delta := \{p_{h'} - p_h : \forall h < h' \in [1, s]\} \cup \{k\}$ be the set of all differences in the offsets of the arithmetic progressions union the set $\{k\}$ containing the common period. By transitivity of '=', it follows that $d_{l \mod k} = d_{l+\delta \mod k}$ for each $l \in [0, k)$ and $\delta \in \Delta$. Again, by transitivity of '=', it follows that $d_{l \mod k} = d_{l+\sigma \mod k}$ for each $l \in [0, k)$ and each number σ in the set $X := \{(\sum_{i=1}^{s} p_i x_i) + k x_{s+1} : x_1, \ldots, x_{s+1} \in \mathbb{N}\}$. By Proposition 4, we have $X = S \cup (a + b\mathbb{N})$ where $S \subseteq [0, k^2]$ and a is the least integer bigger than k^2 that is a multiple of $b := \gcd(\Delta)$. Observe also that b divides all numbers in S and so we have $d_l = d_{l'}$ for each $l, l' \in [0, k)$ with $l \equiv l' \pmod{b}$. In other words, we have $\mathbf{v} = \underbrace{\mathbf{v}' \ldots \mathbf{v}'}_{k/b \text{ times}}$, where $\mathbf{v}' = d_1 \ldots d_b$. Since $\mathrm{RS}^{p_1}(\mathbf{v}[c]) = \mathbf{w}[c]$, it follows that, for each $q \in \mathbb{N}$, $\mathrm{RS}^{p_1 + bq}(\mathbf{v}[c]) = \mathrm{RS}^{p_1}(\mathrm{RS}^{bq}(\mathbf{v}[c])) = \mathrm{RS}^{p_1}(\mathbf{v}[c]) = \mathbf{w}[c]$. Therefore, we have $S \subseteq p_1 + b\mathbb{N}$. On the other hand, since b divides k and each number in $\{p_j - p_1 : j \in [2, s]\}$, we also have $S \supseteq p_1 + b\mathbb{N}$. This gives us $S = p_1 + b\mathbb{N}$.

From Step 2, we have computed the set $S' := S \cap [0, |c|)$. If $S' = \emptyset$, we also knew that $S_i = \emptyset$. If $S' = \{p\}$ is a singleton, we have $S = p + k\mathbb{Z}$. If $|S'| > 1$, we find the two smallest numbers $p_1 < p_2$ in S'. It follows that $S = p_1 + (p_2 - p_1)\mathbb{Z}$.

Observe that this takes time $O(\|c\|)$. [In fact, it is only linear in the size of the two smallest numbers since we ignore the rest of the members of S'.] □

Example 2. Continuing with our running example, we have $S = (1 + 6\mathbb{Z}) \cup (3 + 6\mathbb{Z}) \cup (5 + 6\mathbb{Z}) = 1 + 2\mathbb{Z}$. ■

The last three lines in Algorithm 2 runs in constant time since determining whether $|S_i| = 0$, $|S_i| = 1$, or $|S_i| > 1$ requires the algorithm to explore only a constant number of elements in S_i.

Summing up. To sum up, the time spent computing the linear congruence equation $x \equiv a_i \pmod{b_i}$ for each $i \in [1, m]$ is $O(\|c_i\|)$. Therefore, our reduction runs in time $O(\sum_{i=1}^{m} \|c_i\|) = O(\|g\|)$, which is linear in input size. Therefore, invoking Proposition 2 on the resulting system of linear congruence equations, we obtain the set of solutions to (*) in linear time.

Example 3. Let us continue with our running example. Let

$$g_1 := c(4, 8) = (6, 5, 7, 3, 2, 1)(4, 8), \quad g_2 := c(4, 8, 9) = (6, 5, 7, 3, 2, 1)(4, 8, 9).$$

Then, running Algorithm 2 on g_1 yields the system $x \equiv 1 \pmod 2 \wedge x \equiv 1 \pmod 2$, which is equivalent to $x \equiv 1 \pmod 2$. Running Algorithm 2 on g_2 yields the system $x \equiv 1 \pmod 2 \wedge x \equiv 1 \pmod 3$. Both systems are solvable. ■

Remark 2. At this stage, the reader might wonder whether the Normal Form Lemma (cf. Lemma 4) is necessary. For example, without this lemma one could directly convert the orbit problem over cyclic groups into satisfiability of positive boolean formulas (i.e. involving both disjunctions and conjunctions) where each proposition is interpreted as a linear congruence equation. [This can be construed as adding the power of disjunction to systems of linear congruence equations.] Unfortunately, it is not difficult to show that the resulting satisfiability problem is NP-complete using the techniques of Gödel numbering (cf. [16, 21]).

5 Making do with linearly many bit operations

Thus far, we have assumed that arithmetic operations take constant time. In this section, since Algorithm 1 makes a substantial use of basic arithmetic operations, we will revisit this assumption. It turns out that, although our reduction (Algorithm 2) to solving a system of linear congruence equations runs in linear time in the bit complexity model, the algorithm for solving the system of equations (Algorithm 1) uses at least a cubic number of arithmetic operations. The main results in this section are two-fold: (1) on inputs given by our reduction, Algorithm 1 runs in sublinear time (more precisely, $O(\log^5 n)$) *on average* in the bit complexity model, and (2) there exists another algorithm for solving a system of linear congruence equations (with numbers in the input represented in unary) that runs in linear time in the bit complexity model in the worst case.

We begin with two lemmas that provide the running time of Algorithm 2 and Algorithm 1 in the bit complexity model.

Lemma 5. *Algorithm 2 runs in linear time in the bit complexity model.*

Proof. On ith iteration, the number $|c_i|$ is stored in binary counter and can be computed by counting upwards from 0 and incrementing by 1 as we go through the elements in c_i. Although a single increment by 1 might take $O(|c_i|)$ bit operations in the worst case (since we have to propagate the carry bit), it is known (e.g. see [11, Chapter 17, p. 454]) that the entire sequence of operations actually takes time $O(|c_i|)$. Finally, since addition and substraction of two numbers can easily be performed in $O(\beta)$ time on numbers that use at most β bits, the operation $b_i := a'_i - a_i$ on the last line of the iteration takes at most $O(\log |c_i|)$ time. Therefore, accounting for all the cycles, the algorithm takes $\sum_{i=1}^{m} O(\|c_i\|) = O(\sum_{i=1}^{m} \|c_i\|) = O(\|g\|)$, which is linear in the input size. □

Lemma 6. *On an input* $\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}$ *with* $N = \max\{b_i : i \in [1, m]\}$, *Algorithm 1 uses at most* $m \log N$ *bits to store any numeric variables. Furthermore, the algorithm runs in time* $O(m^3 \log^2 N)$ *in the bit complexity model.*

Proof. On ith iteration, the number of bits used to store a and b grow by at most $\log b_i$. On the other hand, the invariant that $a', b' \in [0, b_i)$ is always maintained on the ith iteration and so they only need at most $\log N$ bits to represent throughout the algorithm. Hence, the algorithm uses $M = O(m \log N)$ bits to store a, b, a', and b'. Extended Euclidean Algorithm runs in time $O(M^2)$ on inputs where each number uses at most M bits (cf. [11, Problem 31-2]), which also bounds the time it takes on each iteration. Therefore, the algorithm takes at most $O(mM^2) = O(m^3 \log^2 N)$ in the bit complexity model. □

We now provide an average case analysis of the running time of Algorithm 1 on system of linear congruence equations given by our reduction. The input to the orbit problem over cyclic groups includes a permutation $g \in S_n$ and two vectors $\mathbf{v}, \mathbf{w} \in \Gamma^n$. We briefly recall the setting of average-case analysis (cf. [20]). Let Π_N be the set of all inputs to the algorithm of size N. Likewise, let Σ_N be the sum of the *costs* (i.e. running time) of the algorithm on *all* inputs of size N. Hence, if $\Pi_{N,k}$ is the cost of the algorithm on input of size N, then $\Sigma_N = \sum_k k\Pi_{N,k}$. The *average case complexity of the algorithm* is defined to be Σ_N / Π_N.

Theorem 2. *The expected running time of Algorithm 1 in the bit complexity model on inputs provided by Algorithm 2 is* $O(\log^5 n)$.

Proof. The size of a single permutation $g \in S_n$ is $O(n)$ and additionally $\Pi_n = |S_n| = n!$. Suppose that g has k cycles (say, $g = c_1 \cdots c_k$). Then, Algorithm 2 produces a system of equations $\bigwedge_{i=1}^{k} x \equiv a_i \pmod{b_i}$, where $a_i, b_i \in [0, |c_i|)$. By Lemma 6, Algorithm 1 takes $O(k^3 \log^2 n)$ time in the bit complexity model, since $N := \max\{b_i : i \in [1, m]\} \leq n$. In addition, the number of permutations in S_n with k cycles is precisely the definition of the *unsigned Stirling number of the first kind* $\begin{bmatrix} n \\ k \end{bmatrix}$. Therefore, we have $\Sigma_n = O\left(\sum_{k=1}^{n} (k^3 \log^2 n) \begin{bmatrix} n \\ k \end{bmatrix}\right) =$

$O\left(\log^2 n \sum_{k=1}^{n} k^3 \begin{bmatrix} n \\ k \end{bmatrix}\right)$. Therefore, it suffices to show that $\frac{1}{n!}\sum_{k=1}^{n} k^3 \begin{bmatrix} n \\ k \end{bmatrix} \sim c\log^3 n$ for a constant c. The proof can be found in the full version. □

Finally, we will now give our final main result of this section.

Theorem 3. *There exists a linear-time algorithm in the bit complexity model for solving a system of linear congruence equations when the input numbers are represented in unary.*

We now provide an algorithm that witnesses the above theorem. Let $\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}$ be the given system of equations. With unary representation of numbers, the size N_i of the equation $x \equiv a_i \pmod{b_i}$ is $a_i + b_i$. We use n to denote the total number of bits in the system of equations. Initially, we compute a binary representation of all the numbers a_i's, b_i's, and n as in the proof of Lemma 5, which takes linear time. Next we factorise all the numbers b_i into a product of distinct prime powers $p_{j_{i1}}^{e_{i1}} \cdots p_{j_{it_i}}^{e_{it_i}}$, where p_j stands for the jth prime and all e_{ij}'s are positive integers. This can be done in time $O(\sqrt{N_i}\log^2 N_i)$. To obtain this time bound, we can use any *unconditional*[3] deterministic factorisation methods like Strassen's algorithm, whose complexity was shown in [5] (cf. also see [12]) to be $O(f(N^{1/4}\log N))$ for factoring a number N, where $f(M)$ is the number of bit operations required to multiply two numbers with M bits. The standard (high-school) multiplication algorithm runs in quadratic time giving us $f(M) = O(M^2)$, which suffices for our purposes. This shows that Strassen's algorithm runs in time $O(N^{1/2}\log^2 N)$. [In practice, do factoring using the general number field sieve (cf. [11]), which performs extremely well in practice, though its complexity requires some unproven number-theoretic assumptions.]

Next, following Chinese Remainder Theorem (CRT), we compute $z_{ij} := a_i \bmod p_{ij}^{e_{ij}}$ for each $j \in [1, t_i]$. Let us analyse the time complexity for performing this. Each z_{ij} can be computed by a standard algorithm (e.g. see [11]) in time quadratic in the number of bits used to represent a_i and $p_{ij}^{e_{ij}}$. Since each of these numbers use at most $\log N_i$ bits, each z_i can be computed in time $O(\log^2 N_i)$, which is $o(N_i)$. In addition, since $e_{ij} > 1$ for each $j \in [1, t_i]$, it follows that $t_i = O(\log N_i)$. This means that the total time it takes to compute $\{z_{ij} : j \in [1, t_i]\}$ is $O(\log^3 N_i)$, which is also $o(N_i)$. So, computing this for all $i \in [1, m]$ takes time $O(\sum_{i=1}^{m} \log^3 N_i)$, which is at most linear in the input size.

In summary, for each $i \in [1, m]$, we obtained the following system of equations, which is equivalent to $x \equiv a_i \pmod{b_i}$ by CRT:

$$x \equiv z_{i1} \pmod{p_{i1}^{e_{i1}}} \quad \wedge \quad \cdots\cdots \quad \wedge \quad x \equiv z_{it_i} \pmod{p_{it_i}^{e_{it_i}}} \qquad (E_i)$$

The final step is to determine if there exists a number $x \in \mathbb{N}$ that satisfies *each* (E_i), for all $i \in [1, m]$. Loosely, we will go through all the equations and makes sure that there is no conflict between any two equations whose periods are powers of the same prime number, i.e., $x \equiv a \pmod{b}$ and $x \equiv a' \pmod{b'}$

[3] This means that the bound does not depend on any number-theoretic assumptions.

such that $b = p^i$ and $b' = p^{i'}$ for some prime p and $i, i' \in \mathbb{Z}_{>0}$. In order to achieve this in linear-time in the bit complexity model, one has to store these equations in the memory (in the form of lookup tables) and carefully perform the lookup operations while looking for a conflict. To this end, we first compute $p_{\max} = \max\{p_{ij} : i \in [1, m], j \in [1, t_i]\}$ and $e_{\max} = \max\{e_{ij} : i \in [1, m], j \in [1, t_j]\}$.

Lemma 7. *$p_{\max}$ and $e_{\max}$ can be computed using $O(n)$ many bit operations.*

Proof. The algorithm for computing $p_{\max}$ and $e_{\max}$ is a slight modification of the standard algorithm that computes the maximum number in a list, which sequentially goes through the list $n_1, \ldots, n_m$ while keeping the maximum number $n_{\max}$ in the sublist explored so far. To ensure linear-time complexity, we have to make sure that when comparing the values of n_i and $n_{\max}$, we explore at most n_i bits of $n_{\max}$ (since $n_{\max}$ is possibly much larger than n_i). This is easily achievable by assuming binary representation of these numbers *without redundant leading 0s*, e.g., the number 5 will be represented as 101, not 0101 or 00000101. That way, we will only need to inspect $\log(n_i)$ bits from $n_{\max}$ on the ith iteration, which will give a total running time of $O(\sum_{i=1}^{m} \log(n_i))$, which is linear in input size. □

Next, keep one 1-dimensional array A and one 2-dimensional array B:

$$A[1, \ldots, p_{\max}] \qquad\qquad B[1, \ldots, p_{\max}][1, \ldots, e_{\max}].$$

$A[k]$ and $B[k][e]$ will not be defined when k is not a prime number. We will use $A[k]$ as a flag indicating whether some equation of the form $x \equiv z \pmod{k^e}$ has been visited, in which case $A[k]$ will contain (z, e). In this case, we will use $B[k][e']$ (with $e' \leq e$) to store the value of $z \mod k^{e'}$.

We now elaborate how A and B are used when iterating over the equations in the system. Sequentially go through each system (E_i) of equations. For each $i \in [1, m]$, sequentially go through each equation $x \equiv z_{ij} \pmod{p_{ij}^{e_{ij}}}$, for each $j \in [1, t_i]$, and check if $A[p_{ij}]$ is defined. If it is not defined, set $A[p_{ij}] := (z_{ij}, e_{ij})$ and compute $B[p_{ij}][l] = z_{ij} \mod p^l$ for each $l \in [1, e_{ij}]$. If it is defined (say, $A[p_{ij}] = (z, e)$), then we analyse the constraints $x \equiv z \pmod{p_{ij}^e}$ and $x \equiv z_{ij} \pmod{p_{ij}^{e_{ij}}}$ simultaneously. We compare e and e_{ij} resulting in three cases:

Case 1. $e = e_{ij}$. In this case, make sure that $z = z_{ij}$ otherwise the two equations (and, hence, the entire system) cannot be satisfied simultaneously.

Case 2. $e < e_{ij}$. In this case, make sure that $z_{ij} \equiv z \pmod{p_{ij}^e}$ (otherwise, unsatisfiable) and assign $A[p_{ij}] := (z_{ij}, e_{ij})$. For each $l \in [1, e_{ij}]$, update $B[p_{ij}][l] := z_{ij} \mod p_{ij}^l$.

Case 3. $e > e_{ij}$. In this case, make sure that $z_{ij} \equiv z \pmod{p_{ij}^{e_{ij}}}$ (otherwise, unsatisfiable).

We now analyse the running time of this final step (i.e. when scanning through the subsystem (E_i)). To this end, we measure the time it takes to process each equation $x \equiv z_{ij} \pmod{p_{ij}^{e_{ij}}}$. There are two cases, which we will analyse in turn.

(Case I): when $A[p_{ij}]$ is not defined. In this case, setting $A[p_{ij}]$ takes constant time, while setting $B[p_{ij}][l]$ for all $l \in [1, e_{ij}]$ takes $O(e_{ij} \times (\log z_{ij} + \log p_{ij}^{e_{ij}})^2)$ since computing $a \mod b$ can be done in time quadratic in $\log(a) + \log(b)$. Since $e_{ij} \leq \log N_i$ and $z_{ij}, p_{ij} \leq N_i$, this expression can be simplified to $O(\log N_i \times \log^2(z_{ij} N_i p_{ij})) = O(\log^3 N_i)$.

(Case II): when $A[p_{ij}]$ is already defined, e.g., $A[p_{ij}] = (z, e)$. In this case, we will compare the values of e and e_{ij}. To ensure linear-time complexity, we will make sure that at most $\log(e_{ij})$ bits from e are read by using the trick from the proof of Lemma 7. For Case 1, we will need extra $O(\log z_{ij}) = O(\log N_i)$ time steps. For Case 2, we have $0 \leq z \leq p^{e_{ij}}$ and computing $z_{ij} \mod p_{ij}^{e}$ can be done in time $O(\log^2 N_i)$ as before. Updating $B[p_{ij}][l]$ for all $l \in [1, e_{ij}]$ takes $O(\log^3 N_i)$ as in the previous paragraph. For Case 3, since $e > e_{ij}$, we may access the value of $z \mod p_{ij}^{e_{ij}}$ from $B[p_{ij}][e_{ij}]$ in constant time and compare this with the value of z_{ij}. Since $z \in [0, p_{ij}^{e_{ij}})$, this takes time $O(\log N_i)$.

In summary, either case takes time at most $O(\log^3 N_i)$. Therefore, accounting for the entire subsystem (E_i), the algorithm incurs $O(\sum_{j=1}^{t_i} \log^3 N_i) = O(\log^4 N_i)$ time steps. Hence, accounting for *all* of the subsystems E_i ($i \in [1, m]$) the algorithm takes time $O(\sum_{i=1}^{m} \log^4 N_i)$, which is linear in the size of the input. This completes the proof of Theorem 3.

Remark 3. The purpose of the 2-dimensional array B above is to avoid superlinear time complexity for Case 3. We can imagine a system of linear equations $\bigwedge_{i=1}^{m} x \equiv a_i \pmod{b_i}$, where a_1 and b_1 are substantially larger than the other a_i's and b_i's ($i \in [2, m]$). In this case, without the lookup table B, checking whether $a_i \equiv a_1 \pmod{b_i}$ in Case 3 will require the algorithm to inspect the entire value of a_1, which prevents us from bounding the time complexity in terms of a_i and will yield a superlinear time complexity for our algorithm.

6 Future work

We mention several future research avenues. Firstly, can we extend polynomial-solvability to any fixed number $k \in \mathbb{Z}_{>0}$ of group generators? The polynomial-time reduction in [10] from the graph isomorphism problem to the orbit problem requires an unbounded number of generators. In addition, the generalisation of the orbit problem over rational matrices to any fixed number k of matrices viewed as generators of (semi)groups is undecidable even when $k = 3, 4$, though results on polynomial-time solvability (hence, decidability) exist when the matrices commute (see [2] and references therein). So, polynomial-time solvability does not follow from the corresponding problem over matrices. The second problem concerns the constructive orbit problem over cyclic groups. Due to the lack of a target configuration $\mathbf{w} \in \Gamma^n$, our technique does not seem to apply directly in this case. In particular, we cannot simply use $\mathbf{w} \in \Gamma^n$ that is derived from the input configuration $\mathbf{v} \in \Gamma^n$ by separately finding the lexicographically minimum parts for each cycle in the given permutation, since this might render the system of equations insoluble.

References

1. `http://oeis.org/A002110`. Primorial Numbers (The On-Line Encyclopedia of Integer Sequences).
2. L. Babai, R. Beals, J.-Y. Cai, G. Ivanyos, and E. M. Luks. Multiplicative equations over commuting matrices. In *SODA*, pages 498–507, 1996.
3. L. Babai and E. M. Luks. Canonical labeling of graphs. In *STOC*, pages 171–183, 1983.
4. E. Bach and J. Shallit. *Algorithmic Number Theory*, volume 1 of *Foundations of Computing*. The MIT Press, 1996.
5. A. Bostan, P. Gaudry, and É. Schost. Linear Recurrences with Polynomial Coefficients and Application to Integer Factorization and Cartier-Manin Operator. *SIAM J. Comput.*, 36(6):1777–1806, 2007.
6. R. A. Brualdi. *Combinatorial matrix classes*. Encyclopedia of Mathematics and Its Applications 108. Cambridge University Press, 2006.
7. P. J. Cameron. *Permutation Groups*. London Mathematical Society Student Texts. Cambridge University Press, 1999.
8. M. Chrobak. Finite automata and unary languages. *Theor. Comput. Sci.*, 47(3):149–158, 1986.
9. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions inmodel checking. In *CAV*, pages 147–158, 1998.
10. E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.
11. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
12. E. Costa and D. Harvey. Faster deterministic integer factorization. *CoRR*, abs/1201.2116, 2012.
13. A. F. Donaldson and A. Miller. On the constructive orbit problem. *Ann. Math. Artif. Intell.*, 57(1):1–35, 2009.
14. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.
15. P. Erdös and R. L. Graham. On a linear diophantine problem of Frobenius. *Acta Arithm.*, 21:399–408, 1972.
16. S. Göller, R. Mayr, and A. W. To. On the computational complexity of verifying one-counter processes. In *LICS*, pages 235–244, 2009.
17. G. H. Hardy and E. M. Wright. *An Introduction to The Theory of Numbers*. OUP Oxford, 6 edition, 2008.
18. C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.
19. R. Kannan and R. J. Lipton. Polynomial-time algorithm for the orbit problem. *J. ACM*, 33(4):808–821, 1986.
20. R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Professional, 2 edition, 2013.
21. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC*, pages 1–9, 1973.
22. A. W. To. Unary finite automata vs. arithmetic progressions. *Inf. Process. Lett.*, 109(17):1010–1014, 2009.
23. T. Wahl and A. F. Donaldson. Replication and abstraction: Symmetry in automated formal verification. *Symmetry*, 2:799–847, 2010.