Efficient Algorithmic Learning of the Structure of Permutation Groups by Examples

S. AZHAR AND J. H. REIF*
Computer Science Department, Duke University
Durham, NC 27706, U.S.A.

Abstract—This paper discusses learning algorithms for ascertaining membership, inclusion, and equality in permutation groups. The main results are randomized *learning algorithms* which take a random generator set of a fixed group $G \leq S_n$ as input. We discuss randomized algorithms for learning the concepts of group membership, inclusion, and equality by representing the group in terms of its strong sequence of generators using random examples from G. We present $O(n^3 \log n)$ time sequential learning algorithms for testing membership, inclusion and equality. The running time is expressed as a function of the size of the object set. ($G \leq S_n$ can have as many as $n!$ elements.) Our bounds hold for all input groups. We also introduce limited parallelism, and our lower processor bounds make our algorithms more practical.

Finally, we show that learning two-groups is in class NC by reducing the membership, inclusion, and inequality problems to solving linear systems over $GF(2)$. We present an $O(\log^3 n)$ time learning algorithm using n^ω processors for learning two-groups from examples (where $n \times n$ matrix product takes logarithmic time using n^ω processors).

1. INTRODUCTION

1.1. Motivation

In mathematical discussions, we often employ a set of illustrative examples to demonstrate general principles as well as to supplement the "theorem-proof" discourse. The use of examples to exhibit creative mathematical ideas dates back to ancient Greek philosophers. Effective examples can be gainfully engaged to appeal to the audience's intuition, and can be successfully used to introduce creative theorems. Examples are also used as a research tool to cultivate a deeper appreciation of, and to draw inference about, virgin ideas. As Halmos said: "the heart of mathematics consists of concrete examples and concrete problems".

This paper is primarily concerned with learning mathematical concepts from examples. Since algebraic groups find applications in a diverse arena of topics, we specialize our investigations to mathematical concepts involving algebraic groups. Of special interest to us are the permutation groups. The problems in group theory (see [2,3]) are not only interesting on their own accord, but they also find applications in several areas of computer science, physics, chemistry, and engineering. For example, computer scientists working in graphics and vision are interested in computing symmetries of intricate multidimensional solids.

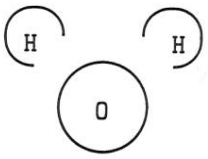
*Supported by Grants NSF/DARPA CCR-9725021, CCR-96-33567, NSF IRI-9619647, and ARO Contract DAAH-04-96-1-0448. An extended abstract of this paper appeared in [1].

Preprint of paper appearing in Computers & Mathematics with Applications, Volume 37, Issue 10, May 1999, pp 105-132.

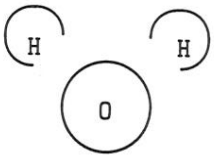
See [3–10] for efficient algorithms (and also parallel algorithms [11,12]) for various group theoretic problems given the generators. There also has been work using a group-theoretic approach to the graph isomorphism problem [13–15]. This approach involves reducing the isomorphism to a membership test in the group of all automorphisms (bijective mapping onto itself) of the graph.

Our methods rely on a source of examples of the group elements, and they demonstrate how succinct representation of the group structure can be efficiently computed, and used to efficiently ascertain membership, inclusion, and equality in groups. We say that we have *learned* the structure of a group, if we have a polynomial time algorithm to construct a representation of the group, such that we can answer the membership question in polynomial time. The complexity is measured in the terms of the size of the underlying object set. Our algorithms can be employed to efficiently *learn* a feasible representation of symmetries of such multidimensional solids. For a more empirical example, consider the Rubik's cube. The set of all permutations of the cube form a group under the function composition. Our results imply that it is possible to generate the entire group (with arbitrarily high probability) from a very small number of examples. Furthermore, our algorithms can be used to verify (with arbitrarily high probability) the correctness of a "solution method" to the cube, by applying the "solution method" to a small number of permutations.

Some practical applications of our algorithms lie in peripheral fields of physics and chemistry, in particular, solid state physics and molecular symmetry. Suppose that you are examining a model of a water molecule (which consists of an oxygen atom bonded to two hydrogen atoms). The model is so well constructed that it is impossible to distinguish between the two hydrogen atoms. Now, if you were to momentarily close your eyes, someone can rotate the model so that both the oxygen atoms have moved but it is impossible to distinguish between the initial and final positions of the molecule. See Figure 1.



Configuration 1.



Configuration 2.

Figure 1. Both configurations appear similiar after swapping two H atoms.

The symmetry of a molecule is characterized by the fact that it is possible to perform operations, which while interchanging the positions of some of the atoms, give arrangements of atoms which are indistinguishable from the initial arrangement. The set of symmetry operations (on a molecule or a crystal) form a group under the function composition. In a simple structure like the water molecule, it is not very difficult to deduce all the symmetries. Nevertheless, when one studies more complex structures, with an increasing number of symmetries, it becomes exceedingly difficult to deduce all symmetries of the molecules or the crystals. Our results can be used to learn the structure of the entire symmetry group by analyzing relatively few examples of symmetry operations. Furthermore, symmetry groups find important applications in theory of angular momenta, which is a topic in itself.

1.2. Learnability

The ability to learn new concepts is an essential manifestation of intelligent behavior. We say that a concept has been acquired by learning if the method for identifying the concept has been developed without explicitly programming the concept itself. Human beings are capable of grasping a fresh concept with the aid of their sensory perceptions and reasoning faculties. The five sensory abilities serve as a protocol for collecting information. This information is used by the

reasoning faculty to learn the concept by associating it with some features. Similarly, a *learning machine* consists of a *learning protocol* and a *deduction procedure*. The protocol accumulates information, which is used by the deduction procedure to assimilate the concept. Learning in our world implies deduction of a recognition procedure by which a machine can correctly classify a given input as being a positive or a negative example of a concept.

The learning protocol can be thought of as a data collector for the deduction procedure. We employ the most commonly used learning protocol[1]: a source of examples (which will be referred to as EXAMPLE, henceforth). Generally, an example can be a completely or an incompletely specified vector. A request to this source, EXAMPLE(G), produces a positive example of the concept G. Our preliminary analysis is based on the assumption that a call to EXAMPLE(G) produces every possible positive example of the concept G with a probability $1/|G|$. This facilitates our developing a thorough comprehension of the methods. In the strict definition of learnability, we are not allowed to place any restriction on the distribution of EXAMPLE(G). Consequently, we extend the analysis of our algorithms to other fixed distributions.

Our goal is to devise sequential and parallel algorithms which would learn to represent any group in such a way that we are able to 'efficiently' answer the membership query. For our models of computation, we assume the unit cost Random Access Machine (RAM) for sequential computation and the CRCW parallel RAM (PRAM) for parallel computation (see also the texts of [16,17]).

The succinct representation of an arbitrary algebraic group is the concept we desire to learn, and the members of the group are the examples of the concept (which would enable us to learn the concepts).

Suppose, we are learning a class C of concepts, where each element in the class consists of a representation of some group (which is a subgroup of the the group of all bijective mappings from a set of n elements onto itself, i.e., S_n). A single concept c is selected from the class C, and we are given a set of examples of this concept. Now, a learning algorithm for class C is a function which takes the relevant examples, and returns as its hypothesis some concept in class C. This hypothesis is a suitable representation of some subgroup G of S_n (symbolically $G \leq S_n$). Formally, a concept is defined as follows.

DEFINITION 1.2.1. CONCEPT. *Concept is defined in terms of its component features which in turn may themselves be concepts or primitive sensory inputs. A concept consists of a domain of n relevant features, where each of the features may take a range of values. A concept G is a subset of all possible vectors.*

If we are interested in answering group theoretic questions, the group representation scheme we deploy is of paramount importance. We are concerned with space conservation as well as time efficiency. When we talk about representation of groups as concepts, there are several conceivable ways to describe them in terms of their component features. For example, we may represent a group by enumerating all its members. Such a naive approach has its obvious shortcomings because a permutation group on n elements can have size $n!$ (which is $O(n^n)$). Even though the membership question can be answered with some degree of efficiency, this representation is forbidden due to its space consumption and other notable shortcomings. Instead, we can choose to represent the groups in terms of their generators. This representation is easy to compute, but it does not lend itself to efficient algorithms for ascertaining group membership, and answering other group-theoretic concepts. Nevertheless, there are some merits to representing groups by their generators, and several fundamentally productive concepts become evident in the development of this idea. However, we will see that it is preferable to depict them in terms of their *strong sequence of generators* because it is more computationally efficient representation. This technique involves recursively representing a group by elements of its cosets.

[1]Another commonly used protocol is ORACLE(x) which determines if the input x is a possible example of the concept.

There are several learning paradigms, and most of these paradigms (like failure-driven or exploratory) are dependent on examples to formulate and strengthen their hypothesis for the concept they are learning.

DEFINITION 1.2.2. EXAMPLES. *If $\vec{x} \in G$, then $\vec{x}$ is a positive example of the concept G, otherwise it is a negative example of the concept G. For our applications, the positive examples are elements of the group $G \leq S_n$, and the negative examples are the elements of S_n which are not in G.*

We treat representation of groups as concepts, and the members of the groups are examples of the concept. Valiant formalized the notion of complexity-based learning from examples. In this paper, we extend and specialize Valiant's model to learning concepts defined by group structures.

In learning a class C of concepts from examples, a single concept is selected from C, and we are given a finite set of positive examples of this concept (which would be members of the group at hand).

DEFINITION 1.2.3. LEARNING. *Learning is a recognition procedure by which a machine can correctly classify a given input as being a positive or a negative example of a concept.*

A learning algorithm for a class C is a function that returns a representation of $G \leq S_n$, which is its hypothesis for the given concept. A learning algorithm operates by drawing m examples of the concept to be identified, and then forms a hypothesis of the concept. The bounds on the number of examples drawn, in order to *learn* the concept, are required to be independent of the underlying probability distribution of the examples. The model of a learning machine has the following properties.

- For sufficiently large m, the learning machine should be capable of achieving arbitrarily small error with arbitrarily high probability. Let U be the set of all concepts, and consider $G, H \in U$. Let $\Delta(G, H)$ represent the symmetric difference between the two concepts. So $\Delta(G, H)$ is the set of all vectors for which G and H disagree:

$$\{\vec{x} \mid \vec{x} \in (G - H) \cup (H - G)\}.$$

  Then

$$d(G, H) = \sum_{x \in \Delta(G,H)} \text{Prob}(x)$$

  is the probability that a call to EXAMPLE(G) will produce an element of $\Delta(G, H)$ with respect to a fixed distribution D. For our purposes, $d(G, H)$ is a measure of difference between concepts G and H. Given parameters ϵ, δ, where $0 < \{\epsilon, \delta\} < 1$, we require that $\text{Prob}(d(L_o, L_h) \geq \epsilon) \leq \delta$, where L_o is the actual concept to be learned, and L_h is the deduced concept (which is the output of the learning algorithm).
- It is capable of learning a whole "class" of concepts. By a "class" of concepts, we mean a set of concepts which have some common properties such that if the machine can "learn" one of them, it can also learn the other concepts in the same class. In this spirit, our learning algorithms work for all groups.
- The computational procedure by which the machines learn, i.e., the learning algorithm, learns the desired concept such that the number of steps are polynomial in the inverse error probabilities as well as in the size of the sample. Our analysis accounts for worst-case possibility.

Observe that we are not allowed to explicitly program the recognition procedure itself.

In his paper, Valiant [18] laid the foundation for the theory of the learnable. He formalized the notion of *learnable* and proposed a framework for the development of the theory of the learnable. To *learn* a concept, we have to devise a polynomial time algorithm to construct a representation of the concept, such that we can answer the membership question in polynomial time. Valiant definition of learnable states the following.

DEFINITION 1.2.4. LEARNABLE. *A class C of concepts is learnable, with respect to a given learning protocol, if and only if there exists a deduction procedure P invoking the protocol with the following properties.*

1. *For all programs $f \in C$, and all distributions D over vectors v on which f outputs 1, the procedure will deduce with probability at least $(1 - h^{-1})$ a program $g \in C$ that never outputs yes when it should not, but outputs yes 'almost always' when it should. More specifically,*
   (a) *for all vectors v, $g(v) =$ yes implies $f(v) =$ yes;*
   (b) *the sum of $D(v)$ over all vectors v such that $f(v) =$ yes but $g(v) \neq$ yes is at most h^{-1}.*
2. *The running time of the algorithm is polynomial in the adjustable parameter h, the size of the underlying object set of G, and other measures of size of the concept.*

Valiant's paper [18] triggered a considerable interest in the theory of the *learnable*.

1.3. Overview of Results and Organization

In Section 1, we introduce and motivate the topic of probabilistic group theoretic algorithms, and also describe our results.

Section 2 covers these preliminary concepts. We discuss the three fundamental group theoretic problems we will be concerned with in this paper.

1. *Group Membership*: given an arbitrary input $x \in S_n$ where $G \leq S_n$, determine whether $x \in G$.
2. *Group Inclusion*: given two groups G and H, determine whether $G \leq H$.
3. *Group Equality*: given two groups G and H, determine whether $G = H$.

Section 3 describes methods for generating examples, and to infer group representation from random examples. We present algorithms for constructing strong sequence of generators for a finite abstract group, and algorithms for ascertaining membership, inclusion, and equality.

In Section 4, we specialize the discussion to permutation groups and present several learning algorithms for permutation groups. We introduce methods to compute G-orbits, and an algorithm to construct the Sims' table of a permutation group in $O(n^3 \log n)$ sequential time. We then proceed to show how the Sims' table can be used to ascertain group membership in $O(n^2)$ sequential time, and group inclusion as well as group equality in $O(n^2 k)$ time (where k is the number of generators required to generate the groups at hand).

Subsequently, in Section 5, we shift our attention to parallel algorithms for general permutation groups. We can use the parallel algorithm of Shiloach and Vishkin [19] to compute connected components of a graph in logarithmic time with a linear number of processors. (Alternatively, we may use the randomized parallel algorithm of Gazit [20], which uses logarithmic time with linear work; that is, the product of processors and time is linear.) This leads to several parallel algorithms for the problems mentioned above. Realizing that membership, inclusion, and equality problems appear inherently sequential for general permutation group, we introduce some limited parallelism for these problems. We parallelize the algorithm for constructing Sims' table to run in $O(n)$ time using $O(n^2 \log n)$ processors. We then exhibit how the Sims' table can be used to ascertain group membership in $O(n)$ time using n processors, and group inclusion as well as group equality in $O(n)$ time using nk processors (where k is the number of generators required to generate the groups at hand). Our modest processor bounds make our algorithm particularly feasible.

Section 6 presents some specialize polylog algorithms for two-groups. We show that learning two-groups is in class NC by reducing the membership, inclusion, and inequality problems to solving linear systems over $GF(2)$. Using parallel algorithms for the solution and basis vectors of linear systems, we present an $O(\log^3 n)$ time learning algorithm using n^ω processors for learning

concepts of two-groups from examples (where $n \times n$ matrix product takes logarithmic time using n^ω processors).

2. GROUPS, SUBGROUPS, AND TOWER OF SUBGROUPS

2.1. Fundamentals of Group Theory

There are several algebraic structures which exhibit similar properties. The unifying theme of abstract algebra is to construct adequate abstraction of algebraic structures to prove general results. In turn, these general results which can be specialized to the particular structure to which we decide to apply them. A *group* is one of the most elementary algebraic structures.

DEFINITION 2.1.1. GROUP. *A group $(G, \cdot)$ is a set G together with a binary operation mapping: $G \times G \mapsto G$, written $(a, b) \mapsto a \cdot b$ such that:*

Associativity: *the operation $\cdot$ is associative;*
Identity: *there is an element $e \in G$ such that $e \cdot g = g = g \cdot e$ for all $g \in G$;*
Inverse: *for this element e, there is to each element $g \in G$ an element $g^{-1} \in G$ with $g \cdot g^{-1} = e = g^{-1} \cdot g$.*

A one-to-one mapping from a finite set (S) onto itself is called a *permutation*. A *permutation group* (G) on the set S is a collection of permutations acting on S that form a group under the function composition (the group operation). S is called the *object* set of the group G. In this paper, we will use the Cauchy's cycle notation to represent permutations. For example, (134)(25) denotes the following mapping: $1 \mapsto 3$, $3 \mapsto 4$, $4 \mapsto 1$, $2 \mapsto 5$, $5 \mapsto 2$, and the rest of the elements map to themselves. In general, a permutation π of n elements can be represented as

$$\pi = (a_{1,1}a_{1,2}\ldots a_{1,j_1})(a_{2,1}a_{2,2}\ldots a_{2,j_2})\ldots(a_{m,1}a_{m,2}\ldots a_{m,j_m}),$$

where $a_{k,l} \in \{1, \ldots, n\}$, and π represents $a_{i,1} \mapsto a_{i,2}, a_{i,2} \mapsto a_{i,3}, \ldots, a_{i,j_{(i-1)}} \mapsto a_{i,j_i}$ and $a_{i,j_i} \mapsto a_{i,1}$ for all positive $i \leq m$.

The *order* of a group G is the cardinality of the group ($|G|$), and the *degree* is the cardinality of the object set ($|S|$). If a subset, H, of a group, G, is itself a group under the group operation, then H is called a subgroup of G (symbolically, $H \leq G$). For a subgroup H of G, and some $a \in G$, the set $aH = \{ah \mid h \in H\}$ is called the *left coset of H in G containing a*. G/H represents the set of all distinct cosets of G with respect to $H \leq G$. Some well-known properties of cosets are listed in the following lemma.

LEMMA 2.1.1. *Let H be a subgroup of a group G, and let $a, b \in G$, then*

1. *$a \in aH$;*
2. *$aH = H$ if and only if $a \in H$;*
3. *$aH = bH$ if and only if $a^{-1}b \in H$;*
4. *$|G| = |G/H||H|$ (Lagrange).*

For detailed exposition of the fundamental concepts of group theory, Wielandt's book [2] is recommended.

2.2. The Tower of Subgroups

Our goal is to efficiently compute a group representation (from random examples), which will be capable of accurately and efficiently answering the membership query and other related questions, with high probability.

Consider a finite group, G, which is generated by a finite set of generators $\langle g_1, g_2, g_3, \ldots, g_k \rangle$. Let $I = \{e\}$ denote the identity group, where e is the identity element. A computationally feasible representation of a group is in terms of its Strong Sequence of Generators (SSG). The

fundamental underlying idea is to represent any finite group G using the notion of factor groups as follows.

We start by setting $G_0 \leftarrow G$. If $G_0 \neq \{e\}$, then we "factor" G_0 by representing it as $(G_0/G_1)G_1$ for some subgroup G_1 of G_0. If $G_1 \neq \{e\}$, then we continue the process and "factor" G_1 by representing it as $(G_1/G_2)G_2$, for some subgroup G_2 of G_1.

We continue to repeatedly "factor" G_i by representing it as $(G_i/G_{i+1})G_{i+1}$, for some subgroup G_{i+1} of G_i, until $G_i = \{e\}$ (for some $i = h$). Since the cardinality of G_i is monotonically decreasing as i increases, and G_0 is finite, we are guaranteed to find some i $(= h)$ such that $G_i = \{e\}$. Thus, we can write

$$G = G_0 = \left(\frac{G_0}{G_1}\right)\left(\frac{G_1}{G_2}\right)\left(\frac{G_2}{G_3}\right)\left(\frac{G_3}{G_4}\right)\cdots\left(\frac{G_{h-1}}{G_h}\right)G_h.$$

EXAMPLE 2.2.1. For example, consider the group of integers under addition (mod 30) (denoted Z_{30}). We can represent Z_{30} by $(Z_{30}/G_1)(G_1/G_2)(G_2/G_3)G_3$, where

G_1 is the group $\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28\}$ under addition (mod 30);
G_2 is the group $\{0, 6, 12, 18, 24\}$ under addition (mod 30);
G_3 is the group $\{0\}$ under addition (mod 30);

and consequently,

$$\frac{Z_{30}}{G_1} = \{0 + G_1; 1 + G_1\} = \{(0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28); (1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29)\},$$

$$\frac{G_1}{G_2} = \{0 + G_2; 2 + G_2; 4 + G_2\} = \{(0, 6, 12, 18, 24); (2, 8, 14, 20, 26); (4, 10, 16, 22, 28)\},$$

$$\frac{G_2}{G_3} = \{0 \cdot G_3; 6 \cdot G_3; 12 \cdot G_3; 18 \cdot G_3; 24 \cdot G_3\} = \{(0); (6); (12); (18); (24)\}.$$

We are now in a position to formally define **Tower of Subgroups**.

DEFINITION 2.2.1. TOWER OF SUBGROUPS. *A permutation group G can be represented by a finite tower of subgroups $G_0, G_1, G_2, G_3, \ldots, G_h$, such that*

$$G = G_0 > G_1 > G_2 > G_3 > \cdots > G_h = I.$$

The tower has height h. We can write G as follows:

$$G = G_0 = \left(\frac{G_0}{G_1}\right)\left(\frac{G_1}{G_2}\right)\left(\frac{G_2}{G_3}\right)\left(\frac{G_3}{G_4}\right)\cdots\left(\frac{G_{h-1}}{G_h}\right)G_h,$$

where $G_h = \{e\}$.

EXAMPLE 2.2.2. The subgroup tower for Z_{30} defined above is

$$Z_{30} = G_0 > G_1 > G_2 > G_3 = I,$$

and can be represented as follows:

$G_3 = \{0\}$,
$G_2 = \{0, 6, 12, 18, 24\}$,
$G_1 = \{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28\}$,
$G_0 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29\}$.

Now, using the concept of subgroup tower, we introduce a representation scheme which is computationally more powerful than representing the concept of a group by the generators of the group. This scheme represents groups by their **Strong Sequence of Generators** which are defined as follows.

DEFINITION 2.2.2. STRONG SEQUENCE OF GENERATORS (SSG). *For each G_i in the subgroup tower, let R_i be a set containing exactly one element from each coset of G_{i-1}/G_i. By construction, R_i is a set which contains one representative from each coset. Each R_i is called a Complete Set of Coset Representatives for G_{i-1}/G_i. The sequence of sets $R_1, R_2, \ldots, R_h$ is called a strong sequence of generators.*

EXAMPLE 2.2.3. In the running example, from the cosets associated with the subgroup tower of Z_{30} we can select the SSG of the group Z_{30}:

$$R_1 = \{0, 1\},$$
$$R_2 = \{6, 8, 4\},$$
$$R_3 = \{0, 6, 8, 12, 18, 24\}.$$

REMARK 2.2.1. Now, every element $\pi \in G$ has a unique representation of the form $\pi = \pi_1\pi_2\pi_3 \ldots \pi_h$ such that for all $i \in \{1, \ldots, h\}$, $\pi_i \in R_i$.

The unique representation is a direct consequence of the fact that π belongs to a fixed coset at each level i coupled with the construction of the sequence $\{R_i\}_{i=1,\ldots,h}$, such that each coset has exactly one representative. As we sift down (up) the tower, we filter out the appropriate components. For example, consider the representation of Z_{30} as $R_1 = \{0, 1\}$, $R_2 = \{6, 8, 4\}$, $R_3 = \{0, 6, 8, 12, 18, 24\}$. In this SSG, 7 has has a unique representation as $1 + 6 + 0$. 24 has a unique representation as $0 + 6 + 18$. Also, 3 has the unique representation $1 + 8 + 24$ (recall $+$ is addition mod 30.

For permutation groups, we are able to use a restricted SSG notion known as Sims' table (see [2,3]). The main idea is to specialize the definition of the subgroup tower by requiring that the i^{th} subgroup contain only elements which do not affect the elements $1, \ldots, i$. Consequently, in the tower each subgroup fixes one additional element of the object set.

DEFINITION 2.2.3. POINT STABILIZING TOWER. *Consider a permutation group $G \leq S_n$ over the points $\{1, \ldots, n\}$. For $i = 1, \ldots, n$, let G_i be the subgroup of G fixing the points $1, \ldots, i$. The resulting tower $G = G_0 > G_1 > \cdots > G_n = I$ of height n is called the point stabilizing tower of G.*

EXAMPLE 2.2.5. The point stabilizing tower of

$$S_3 = \{(1), (12), (13), (23), (123), (132)\}$$

is as follows:

$$G_3 = \{(1)\},$$
$$G_2 = \{(1)\},$$
$$G_1 = \{(1), (23)\},$$
$$G_0 = \{(1), (12), (13), (23), (123), (132)\}.$$

DEFINITION 2.2.4. SIMS' TABLE. *A strong sequence of generators $R_1, \ldots, R_n$ for a point stabilizing tower is called the Sims' table.*

EXAMPLE 2.2.5. The coset of S_3 with respect to its point stabilizing tower are

$$\frac{G_0}{G_1} = \{[(1), (23)]; [(12), (132)]; [(23), (123)]\},$$
$$\frac{G_1}{G_2} = \{[(1)]; [(23)]\},$$
$$\frac{G_2}{G_3} = \{[(1)]\},$$

The Sims' table for S_3 is

$$R_1 = \{(1), (12), (23)\}; \qquad R_2 = \{(1), (23)\}; \qquad R_3 = \{(1)\}.$$

REMARK 2.2.2. Now, every permutation $\pi \in G$ has a unique representation of the form $\pi = \pi_1\pi_2\pi_3 \ldots \pi_n$ such that for all $i \in \{1, \ldots, n\}$, $\pi_i \in R_i$, where π_i does not effect the elements $1, \ldots, i$.

For example, $(12) \in S_3$ has a unique representation $(12) \cdot (1) \cdot (1)$. Similarly, (132) has the unique representation $(12) \cdot (23) \cdot (1)$.

LEMMA 2.2.1. *For any group $G \leq S_n$ there is a point stabilizing tower of height n.*

PROOF. The lemma follows from definitions (Definition 2.2.3). ∎

Representation of group elements is more succinct in terms of SSG. Moreover, computations involving group elements can be carried out by sifting through the coset representatives. These preliminary concepts introduced above will prove to be useful in the development of our learning algorithms later.

2.3. An Illustrative Example

Consider the dihedral group of order 8 (D_4) associated with the group of symmetries of a rectangle. This group has two generators, namely, $\pi/2$ radians rotation ($\alpha = (1234)$), and reflection across the horizontal plane ($\beta = (14)(23)$). See Figure 2.

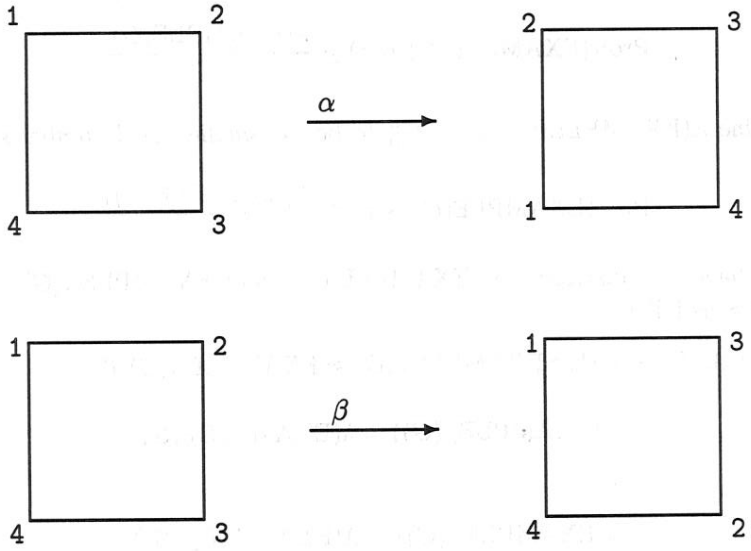


Figure 2. The α and β generators.

D_4 consists of the following elements:

$$\epsilon = (1), \qquad \alpha = (1234), \qquad \alpha^2 = (13)(24), \qquad \alpha^3 = (1432),$$
$$\beta = (14)(23), \qquad \beta\alpha = (24), \qquad \beta\alpha^2 = (12)(34), \qquad \beta\alpha^3 = (13).$$

We start with $G_0 = D_4$, and select $G_1 = \{\epsilon, \alpha, \alpha^2, \alpha^3\}$. So $\epsilon G_1 = \{\epsilon, \alpha, \alpha^2, \alpha^3\}$ and $\beta G_1 = \{\beta, \beta\alpha, \beta\alpha^2, \beta\alpha^3\}$. Consequently, we can express $G_0/G_1 = \{\epsilon G_1, \beta G_1\}$. We can choose R_1 to consist of ϵ and β. Now, since $G_1 = \{\epsilon, \alpha, \alpha^2, \alpha^3\}$, we can select $G_2 = \{\epsilon, \alpha^2\}$. This leads to $G_1/G_2 = \{\epsilon G_2, \alpha G_2\}$ and $R_2 = \{\epsilon, \alpha\}$. Finally, $G_2 = \{\epsilon, \alpha^2\}$ forces $G_3 = I = \{\epsilon\}$ and $G_2/G_3 = \{\epsilon G_3, \alpha^2 G_3\}$. As a result $R_3 = \{\epsilon, \alpha^2\}$.

Thus, a subgroup tower for D_4 is $G_0 = (G_0/G_1)(G_1/G_2)(G_2/G_3)G_3$, with $G_0 = D_4$, $G_1 = \{\epsilon, \alpha, \alpha^2, \alpha^3\}$, $G_2 = \{\epsilon, \alpha^2\}$, and $G_3 = I = \{\epsilon\}$. The associated strong sequence of generators will be $R_1 = \{\epsilon, \beta\}$, $R_2 = \{\epsilon, \alpha\}$, and $R_3 = \{\epsilon, \alpha^2\}$.

3. RANDOMIZED ALGORITHMS FOR FINITE GROUPS

3.1. Generating Examples

We choose to employ the most commonly used learning protocol for our algorithms: a function EXAMPLE(G) which returns an example of the concept group G. The distribution of EXAMPLE(G) conforms to some arbitrary prespecified probability distribution function. In the strict definition of learning, we are not allowed to make any assumptions about the distribution of EXAMPLE(G).

However, we shall first derive some results using uniform distribution.

DEFINITION 3.1.1. UNIFEX(G). *The function* UNIFEX(G) *generates examples of elements of a group G according to uniform distribution.*

1. *For all* $x \in G$, Prob(UNIFEX(G) *is* x) $= 1/|G|$.
2. *For all* $x \notin G$, Prob(UNIFEX(G) *is* x) $= 0$.

Given some function EXAMPLE(G), we also need to define λ(EXAMPLE(G)) and Λ(EXAMPLE(G)) to be lower and upper bound (respectively) on the probabilistic distribution corresponding to EXAMPLE(G).

DEFINITION 3.1.2. λ(EXAMPLE(G)) *and* Λ(EXAMPLE(G). *We define* $\lambda(\text{EXAMPLE}(G)) \in [0, |G|]$ *to be the greatest real number such that for all* $x \in G$*:*

$$\text{Prob}\,(\text{EXAMPLE}(G) \text{ is } x) \geq \frac{\lambda(\text{EXAMPLE}(G))}{|G|}.$$

And, we define $\Lambda(\text{EXAMPLE}(G)) \in [0, |G|]$ *to be the smallest real number such that for all* $x \in G$*:*

$$\text{Prob}(\text{EXAMPLE}(G) \text{ is } x) \leq \frac{\Lambda(\text{EXAMPLE}(G))}{|G|}.$$

Suppose we have two distributions $\text{EXAMPLE}_A(G)$ and $\text{EXAMPLE}_B(G)$. We define the relation $\equiv$ and $\approx$ as follows.

DEFINITION 3.1.3. *We say that* $\text{EXAMPLE}_A(G) \approx \text{EXAMPLE}_B(G)$ *if*

$$\lambda(\text{EXAMPLE}_A(G)) = \lambda(\text{EXAMPLE}_B(G))$$

and

$$\Lambda(\text{EXAMPLE}_A(G)) = \Lambda(\text{EXAMPLE}_B(G)).$$

We say that $\text{EXAMPLE}_A(G) \equiv \text{EXAMPLE}_B(G)$ *if both* $\text{EXAMPLE}_A(G)$ *and* $\text{EXAMPLE}_B(G)$ *have the exactly the same probability distribution function, that is,*

$$\forall g \in G, \qquad \text{Prob}(\text{EXAMPLE}_A(G) \text{ is } g) = \text{Prob}(\text{EXAMPLE}_B(G) \text{ is } g).$$

Observe that $\text{EXAMPLE}_A(G) \approx \text{EXAMPLE}_B(G)$ does not imply that every element occurs with exactly the equal probability in both $\text{EXAMPLE}_A(G)$ and $\text{EXAMPLE}_B(G)$. It merely indicates that the lower and upper bounds of the two corresponding probabilistic distribution are the same. Observe that $\text{EXAMPLE}_A(G) \equiv \text{EXAMPLE}_B(G)$ is a much stronger condition.

LEMMA 3.1.1. *If G is a group and $x \in G$, then* $\text{UNIFEX}(G) \equiv x \cdot \text{UNIFEX}(G)$.

PROOF. We can define a function $\chi_x : G \mapsto G$ such that $\chi_x(y) = x \cdot y$. Now, the function χ_x is one-to-one and onto. ($xy_1 = xy_2$ implies $y_1 = y_2$, and for any $y \in G$, $\chi_x(x^{-1}y) = y$.) The lemma follows. ∎

LEMMA 3.1.2. *Let $H < G$, and let R be a complete set of coset representatives for G/H. Then* $\text{UNIFEX}(G) \equiv \text{UNIFEX}(R) \cdot \text{UNIFEX}(H)$.

PROOF. Suppose $\text{UNIFEX}(R) = r$ for some arbitrary $r \in R$, and $\text{UNIFEX}(H) = h$ for some arbitrary $h \in H$. We are interested in the element selected $x = rh$. Intuitively, choice of r determines the coset $rH \in G/H$ which contains x, then choice of h determines the particular element x within the coset rH. Our task is to prove that x is uniformly distributed in G.

We define a function $f : R \times H \mapsto G$ such that $f(r, h) = rh$. This function $f : R \times H \mapsto G$ is one-to-one and onto because of the following.

1. For $f : R \times H \mapsto G$ defined above, let $g_1 = f(r_1, h_1)$ and $g_2 = f(r_2, h_2)$. If $g_1 = g_2$, then r_1h_1 and r_2h_2 are in the same coset. This implies $r_1 = r_2$ because R contains one and only one element from each coset. Multiplying both sides by r_1^{-1} we get $h_1 = h_2$. Thus, f is one-to-one.
2. For all $x \in G$, there exists a coset $r_1 \cdot H \in G/H$ such that $x \in r_1 \cdot H$. It follows that $\exists\, h \in H$ such that $x = r_1h$. Therefore, f is onto.

Consequently, for $x \in G$,

$$\begin{aligned}&\text{Prob}[(\text{UNIFEX}(R) \cdot \text{UNIFEX}(H)) \text{ is } x] \\ &\qquad = \text{Prob}[\text{UNIFEX}(R) \text{ is } r] \cdot \text{Prob}[\text{UNIFEX}(H) \text{ is } x] = \frac{1}{|G/H|}\frac{1}{|H|} = \frac{1}{|G|}.\end{aligned}$$

By Lagrange's Theorem $|G/H||H| = |G|$.

Thus, for any fixed x, there is a probability of $\text{UNIFEX}(R) \cdot \text{UNIFEX}(H)$ is x is $1/(|G/H||H|) = 1/|G|$. The lemma follows.

A more concise but less instructive proof of this lemma can be formulated by the use of Bayes' theorem. ∎

EXAMPLE 3.1.1. For the SSG of Z_{30} described in Example 2.2.3, observe that the probability of generating any element of G_0 by $\text{UNIFEX}(R_1) \cdot \text{UNIFEX}(G_1)$ is exactly $(1/2)(1/15) = (1/30)$. This is exactly the probability of generating any element of G using $\text{UNIFEX}(G_0)$.

Given a strong sequence of generators $R_1, \ldots, R_n$ for group G, with respect to its subgroup tower $G = G_0 > G_1 > G_2 > \cdots > G_n = I$, we will present a simple algorithm for random element generation based on the following lemma.

LEMMA 3.1.3. $\text{UNIFEX}(G) \equiv \text{UNIFEX}(R_1) \cdots \text{UNIFEX}(R_h)$.

PROOF. The proof is by induction on the height of the subgroup tower in conjunction with Lemma 3.1.2. By Lemma 3.1.2, $\text{UNIFEX}(G_0) \equiv \text{UNIFEX}(R_1) \cdot \text{UNIFEX}(G_1)$, and for all i such that $1 \leq i \leq n-1$, $\text{UNIFEX}(G_i) \equiv \text{UNIFEX}(R_{i+1}) \cdot \text{UNIFEX}(G_{i+1})$. Finally, $\text{UNIFEX}(R_h) \equiv e$. Consequently, $\text{UNIFEX}(G) \equiv \text{UNIFEX}(R_1) \cdots \text{UNIFEX}(R_h)$. ∎

Intuitively, this lemma relies on the fact that each element of any group G can be represented uniquely by selecting elements from its SSG (see also Remark 2.2.2). Now, we can compute $\text{UNIFEX}(G)$ from strong sequence of generators by a simple algorithm.

ALGORITHM 3.1.1. EXAMPLE GENERATION.

INPUT: strong Sequence of Generators $R = \{R_1, R_2, \ldots, R_h\}$ of a group G.
OUTPUT: an example element of group G.
BEGIN
  for $i = 1$ to h do
    $x_i = \text{UNIFEX}(R_i)$;
  end for
  return $x_1 \cdot \ldots \cdot x_h$
END

THEOREM 3.1.1. *Algorithm 3.1.1 can produce any element of the group G with equal probability in $O(h)$ sequential time.*

PROOF. The proof of correctness follows from Lemma 3.1.3. The complexity is clearly $O(h)$. ∎

REMARK 3.1.1. It is interesting to observe that a random element of G can be generated by this method in parallel by a binary product tree of depth $O(\log n)$ using $O(n)$ processors. There are n input nodes of the tree, and the i^{th} input node is fed with UNIFEX(R_i). The rest of the processors perform the group operation on the two inputs they receive.

EXAMPLE 3.1.2. It follows from the discussion in Section 2.3 that UNIFEX(D_4) $\equiv$UNIFEX($\{\epsilon, \beta\}$) $\cdot$ UNIFEX($\{\epsilon, \alpha\}$) $\cdot$ UNIFEX($\{\epsilon, \alpha^2\}$).

3.2. Finding Generators from Random Examples

In this section, we will show an important result concerned with group inference from random examples. Let G be a fixed finite group of permutations. Let $L = \{x_1, x_2, x_3, \dots\}$ be a possibly infinite set of random elements chosen independently from G using UNIFEX(G). Our objective is to find an upper bound on the number of examples required to generate G (with very high success probability).

THEOREM 3.2.1. *For $0 < \epsilon < 1$, Prob$(G = \langle x_1, x_2, \dots, x_m\rangle) \geq 1-\epsilon$ if $m \geq 1+(\log|G|)\log(1/\epsilon_1)$, where $\epsilon_1 = 1-(1-\epsilon)^{1/\log|G|}$.*

PROOF. Let $J = \{j \in Z^+ \mid x_j \notin < x_1, x_2, \dots, x_{j-1} >\}$. Intuitively, the set J indexes the generators of the group by adding an element of the group (at each step) if it cannot be generated with the set of generators found so far. Even though, the list J can be constructed by repeatedly drawing a random element from the group, and appending it to the list of generators if it cannot be generated by the previous set of generators. Since the main concern of the theorem is finding an upper bound, and not actual computation, we will only keep track of the indices of the elements we decide to keep. The idea is to repeat, m times, the process of finding new elements of the group G, which cannot be produced by generator set found thus far. For example, if the list $L = \{15, 0, 5, 3, 4, 23\}$ for Z_{30} would result in $J = \{1, 4\}$, since only 15 and 3 cannot be generated by previous generators.

List all the elements of J in ascending order, $J = \{j_{[1]}, j_{[2]}, \dots, j_{[|J|]}\}$. This will give us a tower of subgroups $G = G_0 > G_1 > \cdots > G_{|J|} = I$, where $G_s = \langle x_{j_{[1]}}, \dots, x_{j_{[|J|-s]}}\rangle$. We can see that $G_0 = \langle x_{j_{[1]}}, \dots, x_{j_{[|J|]}}\rangle = G$, and $G_{|J|} = \{e\}$ by default. In particular, $G = G_0 = \langle x_{j_1}, \dots x_{j_{|J|}}\rangle$, and an upper bound on $j_{[|J|]}$ will also be the upper bound on the number of examples necessary.

By Lagrange's theorem $|G_s| \leq |G_{s-1}|/2$. By induction $|G_s| \leq |G|/2^s$, so for $s = |J|$, $|G_{|J|}| \leq |G|/2^{|J|}$. Since $|G_{|J|}| = 1$, it follows from $|G_{|J|}| \leq |G|/2^{|J|}$ that $|J| \leq \log|G|$. Since $|G_s| \leq |G|/2^s$, if we choose a random example using UNIFEX(G), then the probability that this element is in G_s is equal to $1/2^s$. In other words, Prob $(x_{(j_{[|J|-s]}+1)} \in \langle x_{(j_{[1]})}, \dots, x_{(j_{[|J|-s]})}\rangle) \leq 1/2^s$. This is equivalent to saying that Prob $(j_{[|J|-s+1]} > j_{[(|J|-s)]} + 1) \leq 1/2^s$ because if $x_{(j_{[|J|-s]}+1)} \in \langle x_{(j_{[1]})}, \dots, x_{(j_{[|J|-s]})}\rangle$, then $j_{[|J|-s]} + 1$ is not added to the set J. Hence, the next element of J (i.e., $j_{[|J|-s+1]}$) is strictly greater than $j_{[|J|-s]} + 1$.

Now the probability of drawing k consecutive elements which are already in G_s is less than $(1/2^s)^k$. This implies that for any positive integer k, Prob $[j_{[|J|-s+1]} > (j_{[|J|-s]} + k)] \leq 1/2^{ks}$. Consequently, Prob $[(j_{[|J|-s+1]} - j_{[|J|-s]}) \leq k] \geq 1 - 1/2^{ks}$. Substituting $\epsilon_1 = 1/2^{ks}$, we can see that Prob $[(j_{[|J|-s+1]} - j_{[|J|-s]}) \leq (1/s)\log(1/\epsilon_1)] \geq 1 - \epsilon_1$.

Now, if $m \geq j_{[|J|]}$, then we can find the generators of the group. Note that $j_{[|J|]} = j_{[1]} + \sum_{s=1}^{|J|-1}(j_{[|J|-s+1]} - j_{[|J|-s]})$. Since the first element picked cannot be generated by a vacuous generator set, by default $j_{[1]} = 1$. Consequently, $j_{[|J|]} = 1 + \sum_{s=1}^{|J|-1}(j_{[|J|-s+1]} - j_{[|J|-s]})$. Now, we know that Prob $[(j_{[|J|-s+1]} - j_{[|J|-s]}) \leq (1/s)\log(1/\epsilon_1)] > 1 - \epsilon_1$. Observe that $(1-\epsilon_1)^{|J|-1} =$

$((1-\epsilon)^{1/\log|G|})^{|J|-1} > 1-\epsilon$. Hence, with probability greater than $(1-\epsilon_1)^{|J|-1} > 1-\epsilon$:

$$\Rightarrow j_{[|J|]} \leq 1 + \sum_{s=1}^{|J|-1} \left(\frac{1}{s}\log\left(\frac{1}{\epsilon_1}\right)\right).$$

Replacing $1/\epsilon_1$ by 2^{ks}:

$$\Rightarrow j_{[|J|]} \leq 1 + \sum_{s=1}^{|J|-1} \left(\frac{1}{s}\log\left(2^{ks}\right)\right).$$

Simplifying

$$\Rightarrow j_{[|J|]} \leq 1 + \sum_{s=1}^{|J|-1} k < |J|k.$$

Replacing $k = (\log(1/\epsilon_1))/s$:

$$\Rightarrow j_{[|J|]} \leq 1 + \sum_{s=1}^{|J|-1} k < \frac{|J|\log(1/\epsilon_1)}{s} \leq |J|\log\left(\frac{1}{\epsilon_1}\right).$$

Since $|J| \leq \log(|G|)$,

$$j_{[|J|]} \leq 1 + (\log(|G|))\log\left(\frac{1}{\epsilon_1}\right).$$

Consequently, $m \geq 1 + (\log(|G|))\log(1/\epsilon_1)$ suffices. The proof of the theorem is now complete. ∎

REMARK 3.2.1. For small ϵ, $\epsilon_1 \approx \epsilon/\log(|G|)$, so $m \approx 1 + (\log(|G|) \cdot \log((\log|G|)/\epsilon)$. In practice, this bound can be deceiving because for small ϵ the $\log((\log|G|)/\epsilon)$ factor can be fairly large.

For $|G| = 2^{1024}$ and $\epsilon = 2^{-10}$, $m \approx 20480$.

Retracing our steps back through the proof of Theorem 3.2.1 above, we can also find upper bounds on the total number of generators, and the height of the subgroup tower.

THEOREM 3.2.2. *For any finite group G, there is a subgroup tower of height at most $\log_2(|G|)$. Furthermore, G can be generated by a subset K (of G) consisting of no more than $\log_2(|G|)$ elements of G.*

PROOF. The proof is by induction. For any group G, we can construct the subgroup tower $G = G_0 > G_1 > G_2 > G_3 > \cdots > G_h = I$. Inductively assume that $G_i =< K_i >$ where K_i is a subset of at most $h-i$ elements of G_i. Now, there must be a permutation π_{i-1} of G_{i-1} which is not a member of the group G_i. Let $K_{i-1} = K_i \cup \{\pi_{i-1}\}$ and $G_{i-1} = \langle K_{i-1}\rangle$, and K_{i-1} contains no more than $h-(i-1)$ elements. This proves that there is subset K_0 (of G_0) which contains no more than h elements.

Observe that $G_{i-1} > G_i$, so by Lagrange's theorem $|G_i| \leq |G_{i-1}|/2$. Hence, $|G_h| \leq |G_0|/(2^h)$. Therefore, $h \leq \log_2(|G|)$. Furthermore, since $|K_0| \leq h$, G can be generated by a subset consisting of no more than $\log_2(|G|)$ elements. ∎

3.3. Constructing Strong Generators

Suppose, we wish to find a complete set of coset representatives of G/H. Let L denote a random list of m $(= |L|)$ independently selected random examples of elements of G. With G, L, and m defined above, the following two lemmas are due to [2].

LEMMA 3.3.1. *For some $H \leq G$, let E denote the event that L contains at least one representative of each coset of G/H. If G/H contains c cosets ($|G/H| = c$), then* Prob $(E) \geq 1 - c(1 - 1/c)^m$.

PROOF. Let C denote a fixed coset in G/H. Since UNIFEX(G) has an equal probability of being a member of any coset in G/H, Prob $(L \cap C = \phi) = (1 - 1/c)^m$. Since there are c cosets in G/H, Prob $(\neg E) \leq c(1 - 1/c)^m$. Thus, Prob $(E) = 1$ - Prob $(\neg E) \geq 1 - c(1 - 1/c)^m$. ∎

Now, if $L \cap C \neq \phi$, choose a random $r_C \in L \cap C$ for *each* coset $C \in G/H$. We define $R = \{r_C \in L \cap C \mid C \in G/H\}$. Since L contains at least one element from every coset with a very high probability ($\geq 1 - c(1 - 1/c)^m$), R contains exactly *one* element from *each* coset in G/H with the same high probability $\geq 1 - c(1 - 1/c)^m$. For $x \in C$, we define a function $f_C : C \Rightarrow H$ as follows: $f_C(x) = r_C^{-1}x$ where r_C is the previously selected element of $L \cap C$. Note that, by property number 3 of Lemma 2.1.1, we know that $r_C^{-1}x \in H$.

LEMMA 3.3.2. *For C, G, H, and the function $f_C(x) = r_C^{-1}x$ defined above,*

$$f_C(\text{UNIFEX}(C)) \equiv \text{UNIFEX}(H).$$

PROOF. For all $x \in G$, there exists a coset $C \in G/H$ such that $x \in C$. Consequently, $f_C(x) = r_C^{-1}x \in H$ because $x \in C$ implies $r_C^{-1}x \in H$. By property number 3 of Lemma 2.1.1, we know that $r_C^{-1}x \in H$. By the definition of the function, the distribution of $f_C(\text{UNIFEX}(C))$ is identical to that of $r_C^{-1}(\text{UNIFEX}(C))$, for some randomly chosen $C \in G/H$. Now using the fact that $\forall C \in G/H$, $r_C^{-1}C = H$, we have

$$\begin{aligned} f_C(\text{UNIFEX}(C)) &\equiv r_C^{-1}(\text{UNIFEX}(C)) \\ &\equiv \text{UNIFEX}(r_C^{-1}C) \equiv \text{UNIFEX}(H). \end{aligned}$$ ∎

Given a subgroup tower $G = G_0 > G_1 > G_2 > G_3 > \cdots > G_{h-1} > G_h$ corresponding to $G = G_0 = (G_0/G_1)(G_1/G_2)(G_2/G_3)(G_3/G_4)\ldots(G_{h-1}/G_h)G_h$, where $G_h = I$ of height h, and width $w \equiv \max_i \{|G_{i-1}/G_i|\}$. We present an algorithm to construct a list of strong generators for G with respect to this subgroup tower.

ALGORITHM 3.3.1. STRONG SEQUENCE OF GENERATOR.

INPUT: the input consists of the following.

(1) L_0 be a list of m elements drawn independently from UNIFEX(G).
(2) A subgroup tower $G = G_0 > G_1 > G_2 > G_3 > \cdots > G_{h-1} > G_h$.

OUTPUT: strong sequence of generators.

BEGIN

1. Let L_0 be a list of m elements drawn independently from UNIFEX(G);
2. for $i = 1$ to h do
   begin
3. Initialize $R_i \Leftarrow \phi$;
4. for each $C \in G_{i-1}/G_i$ do
   begin
5. Initialize $L_{i,C} \Leftarrow \phi$;
6. Let $L_{i,C}$ be list of elements of L_{i-1} in C;
7. if $L_{i,C} \neq \phi$, then
   begin
8. Choose and delete a random element r_C from $L_{i,C}$
9. Add r_C to R_i
10. for each remaining element $x \in L_{i,C}$ do
11. $L_i \Leftarrow L_i \cup \{r_C^{-1}x\}$
   end

end
end
12. return strong sequence of generators $R_1, \ldots R_h$.
END

EXAMPLE 3.3.1. Let us work through Algorithm 3.3.1 for the group S_4. Let $\alpha = (1234)$, $\beta = (234)$, and $\gamma = (34)$. Then

$$S_4 = \left\{\alpha^i\beta^j\gamma^k \mid 0 \le i \le 3, 0 \le j \le 2, 0 \le k \le 1\right\}.$$

Given INPUT:

(1) $L_0 = \left\{\epsilon, \beta^2, \beta\gamma, \alpha, \alpha\beta\gamma, \alpha\beta^2\gamma, \alpha^2\beta, \alpha^2\gamma, \alpha^3, \alpha^3\beta^2, \alpha^3\gamma, \alpha^3\beta^2\gamma\right\}$,

(2) $G_0 = S_4$; $G_1 = \left\{\beta^j\gamma^k \mid 0 \le j \le 2, 0 \le k \le 1\right\}$; $G_2 = \left\{\gamma^k \mid 0 \le k \le 1\right\}$; $G_3 = \{\epsilon\}$,

The algorithm proceeds as follows.
First Pass.

1. $R_1 \leftarrow \phi$.
2. $G_1 = \{\beta^j\gamma^k \mid 0 \le j \le 2, 0 \le k \le 1\}$.
3. $G_0/G_1 = \{G_1, \alpha G_1, \alpha^2 G_1, \alpha^3 G_1\}$.
4. • $L_{1,G_1} \leftarrow \{\epsilon, \beta^2, \beta\gamma\}$;
   • $L_{1,\alpha G_1} \leftarrow \{\alpha, \alpha\beta\gamma, \alpha\beta^2\gamma\}$;
   • $L_{1,\alpha^2 G_1} \leftarrow \{\alpha^2\beta, \alpha^2\gamma\}$;
   • $L_{1,\alpha^3 G_1} \leftarrow \{\alpha^3, \alpha^3\beta^2, \alpha^3\gamma, \alpha^3\beta^2\gamma\}$.
5. $R_1 \leftarrow \{\epsilon, \alpha, \alpha^2\beta, \alpha^3\}$.
6. $L_1 \leftarrow \{\epsilon, \beta^2, \beta\gamma, \beta^2\gamma, \beta, \gamma\}$.

Second Pass.

1. $G_2 = \{\gamma^k \mid 0 \le k \le 1\}$.
2. $G_1/G_2 = \{G_2, \beta G_2, \beta^2 G_2\}$.
3. • $L_{2,G_2} \leftarrow \{\epsilon, \gamma\}$;
   • $L_{2,\beta G_2} \leftarrow \{\beta\gamma, \beta\}$;
   • $L_{2,\beta^2 G_2} \leftarrow \{\beta^2, \beta^2\gamma\}$.
4. $R_2 \leftarrow \{\epsilon, \beta\gamma, \beta^2\}$.
5. $L_2 \leftarrow \{\epsilon, \gamma\}$.

Third Pass.

1. $G_3 = \{\epsilon\}$.
2. $G_2/G_3 = \{G_3, \gamma G_3\}$.
3. • $L_{3,G_3} \leftarrow \{\epsilon\}$;
   • $L_{3,\gamma G_3} \leftarrow \{\gamma\}$.
4. $R_3 \leftarrow \{\epsilon, \gamma\}$.
5. $L_3 \leftarrow \{\epsilon\}$.

Thus,

- $R_1 \leftarrow \{\epsilon, \alpha, \alpha^2\beta, \alpha^3\}$;
- $R_2 \leftarrow \{\epsilon, \beta\gamma, \beta^2\}$;
- $R_3 \leftarrow \{\epsilon, \gamma\}$.

THEOREM 3.3.1. *For a subgroup tower of height h and width w, if $m \ge hw$, then Algorithm 3.3.1 outputs a strong sequence of generators of G with very low error probability ($\le hw(1 - 1/w)^{m-hw}$).*

PROOF. By Lemma 3.3.1, the probability of failure at any stage i with c_i cosets is no greater than $c_i(1 - 1/c_i)^{m-hc_i} \le w(1 - 1/w)^{m-hw}$. Hence, the failure probability at one or more stages

is at most $\sum_{i=1}^{h} c_i(1-1/c_i)^{m-hc_i} \leq hw(1-1/w)^{m-hw}$. Thus, with very high probability $(1 - hw(1-1/w)^{m-hw})$, there is no failure. Hence, each R_i is a complete set of coset representatives for G_{i-1}/G_i. ∎

THEOREM 3.3.2. *Algorithm 3.3.1 computes the strong sequence of generators of input group G (using $m \geq hw$) calls of* UNIFEX(G)) *in $O(m\log(|G|)))$ time, with a success probability $\geq 1 - hw(1-1/w)^{m-hw}$.*

PROOF. The most expensive steps are 6 and 10,11. They can be executed in $O(m)$ time (upper bound). Since they are executed $O(h)$ time, i.e., by Lemma 2.2.1 $O(\log(|G|))$ times, the total time is $O(m\log(|G|))$. ∎

3.4. Membership, Inclusion, Equality

Suppose $G = G_0 > G_1 > G_2 > \cdots > G_h = I$ is a subgroup tower, and $R_1, \ldots, R_h$ is the corresponding strong sequence of generators of G as discussed in the previous subsection. Assume that for all $x \in G_{i-1}$, we can effectively find the coset representative $y \in R_i$ such that $x \equiv_i y$ (i.e., $y^{-1}x \in G_i$).

We will now describe Sims' algorithm for group membership, in the general context of finite groups. Given an input x, and a strong sequence of generators the Sims' group membership is as follows.

ALGORITHM 3.4.1. MEMBERSHIP TESTING.

INPUT: a strong sequence of generators for a group $G \leq U$, and an element $x \in U$.

OUTPUT: whether $x \in G$ or $x \notin G$.

BEGIN

```
for i = 1 to h do
    begin
      if (∃ y ∈ R_i such that x ≡_i y)
          then x ⇐ y^{-1}x
          else return ("x ∉ G")
      end if
    end
return ("x ∈ G")
```

END

THEOREM 3.4.1. *Sims' group membership test has (worst case) sequential running time of $O(|G|)$.*

REMARK 3.4.1. Sims' group membership algorithm is inherently sequential in nature, and the parallel time for its execution is at least $\Omega(h)$, where h is the height of subgroup tower.

The following tree sketches the path followed by the algorithm for testing membership of $6 \in Z_{12}$, where Z_{12} is represented by $R_1 = \{3,4\}$; $R_2 = \{0,6\}$; $R_3 = \{0,4,8\}$.

Consider another example in S_4.

EXAMPLE 3.4.1. Let us see how Algorithm 3.4.1 would ascertain $\alpha\beta\gamma \in S_4$.

Phase One.

Observe $\alpha \in R_1$ and $\alpha \equiv_1 \alpha\beta\gamma$. So we set $x \leftarrow \beta\gamma$.

Phase Two.

Observe $\beta \in R_2$ and $\beta \equiv_2 \beta\gamma$. So we set $x \leftarrow \gamma$.

Phase Three.

Observe $\gamma \in R_3$ and $\gamma \equiv_3 \gamma$. So we set $x \leftarrow \epsilon$.

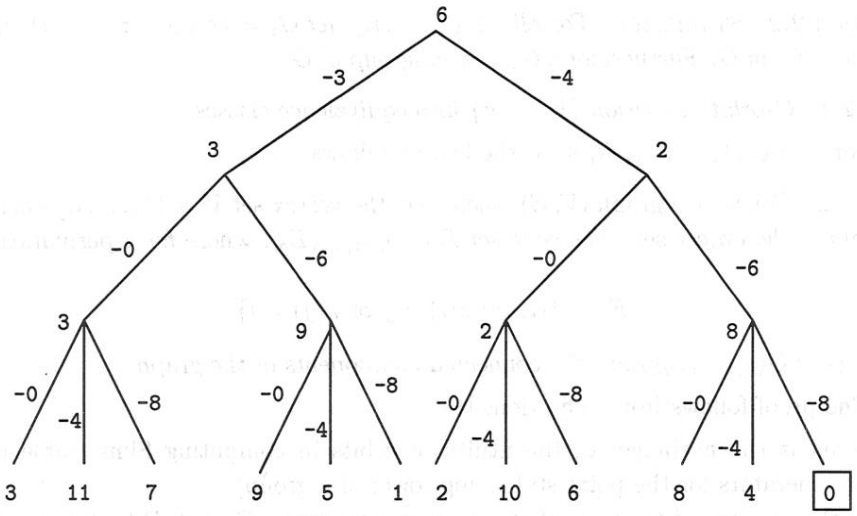


Figure 3. Testing $6 \in Z_{12}$ from SSG.

Consequently, we return "$x \in G$".

REMARK 3.4.2. Given another finitely generated group $G' = \langle h_1, \ldots, h_{k'} \rangle$, and a strong generator sequence for G, we can test group inclusion $G' \leq G$ by simply ascertaining $\forall i = 1, \ldots k'$, that $h_i \in G$.

REMARK 3.4.3. Moreover, given strong generator sequences for finitely generated groups $G = \langle g_1, \ldots, g_k \rangle$ and $G' = \langle h_1, \ldots, h_{k'} \rangle$, we can test group equality ($G = G'$) by ascertaining $\forall i = 1, \ldots, k$ that $g_i \in G'$, and $\forall j = 1, \ldots, k'$, that $h_j \in G$.

THEOREM 3.4.2. *Let A be class of representations of an arbitrary group G in terms of strong sequence of generators. Then class A is learnable.*

PROOF. A can be *learned* from examples in polynomial time using Algorithm 3.3.1. Thus, by definition concept A is learnable. ∎

4. RANDOMIZED ALGORITHMS FOR PERMUTATIONS GROUPS

4.1. Permutation Groups

Group theoretic algorithms may be classified into two categories: algorithms for abstract groups (represented by generators and relations) and algorithms for permutation groups (specified by permutations of an object set). Computer scientists have shown deeper interest in permutation groups for several reasons. At the outset, permutation groups have a natural concrete representation on the computer, and to complement this there is a natural definition of group operations. Furthermore, permutation groups have direct applications to the graph isomorphism problem as well as solid state physics and molecular symmetry. In this section, we will specifically concentrate on problems which arise in permutation groups. We also take this opportunity to recall Cayley's theorem which assures us that every group has an isomorphic permutation group.

REMARK 4.1.1. Every group is isomorphic to a group of permutations.

4.2. Computation of Orbits

Let $G \leq S_n$ be a permutation group on the set $A = \{1, \ldots, n\}$. Since, our algorithms rely heavily on computation of orbits. We recall some elementary definitions before proceeding further.

DEFINITION 4.2.1. ORBITS. *For all $i \in \{1, \ldots, n\}$, define $i^G \equiv \{\pi(i) \mid \pi \in G\}$. The set $i^G \subseteq \{1, \ldots, n\}$ is called the orbit of i under G. A G-orbit is the set of orbits, where each orbit contains a collection of elements which have the same orbit in G.*

DEFINITION 4.2.2. STABILIZER. *For all $i \in \{1,\ldots,n\}$, let $G_i = \{\pi \in G \mid \pi(i) = i\}$. G_i is called the stabilizer of i in G. Furthermore, G_i is a subgroup of G.*

LEMMA 4.2.1. *G-orbits partition $\{1,\ldots,n\}$ into equivalence classes.*

PROOF. For all $i \in \{1,\ldots,n\}$, $G_i \leq G$, the lemma follows. ∎

LEMMA 4.2.2. *Consider a graph (V,E), such that the vertex set $V = \{1,\ldots,n\}$ corresponds to the elements of the object set. The edge set $E = \cup_{i=1,\ldots,k} E_{g_i}$, where for a permutation $\pi \in S_n$, we define*

$$E_\pi = \{(i,j) \mid \pi(i) = j \text{ or } \pi(j) = i\}.$$

The G-orbits of $G\langle g_1,\ldots,g_k\rangle$ are the connected components of the graph.

PROOF. The proof follows from definitions [13]. ∎

This lemma is the harbinger of the utility of orbits in computing Sims' table (the strong sequence of generators for the point stabilizing tower of a group).

Let k be the number of G-orbits of the permutation group G. Let B be be any partition of $\{1,\ldots,n\}$ into $|B|$ blocks such that no block of B contains more than a single G-orbit. Now if we are given any permutation $\pi \in G$, we construct B_π from B as follows.

Repeat until no further changes can be made.

- If there exist two different blocks $b_1,\ b_2 \in B$, and $\exists\, i \in \{1,\ldots,n\}$, such that $i \in b_1$ and $\pi(i) \in b_2$, then merge b_1 and b_2.

Thus, the general scenario is the following.

- B is a (not necessarily proper) refinement of B_π and the G-orbits.
- B_π is a (not necessarily proper) refinement of the G-orbits.

We define a function $\gamma : B \mapsto Z$ to provide us a measure of how many blocks in B are still unresolved.

DEFINITION 4.2.3. γ FUNCTION. *$\gamma : B \mapsto Z$ is defined as follows:*

$$\gamma(B) = |B| - k.$$

Actually, $\gamma(B)$ is equal to the number of blocks which still need to be merged before we have the G-orbits of the group. (When $|B| = k$, $\gamma(B) = 0$.) The two lemmas that follow will be used in the proof of the next theorem.

LEMMA 4.2.3. *If $|B| > k$ and $\pi = \text{UNIFEX}(G)$, then* $\text{Prob}[\gamma(B_\pi) \leq \gamma(B)/2] \geq 1/2$.

PROOF. In anticipation of a contradiction, suppose that $\text{Prob}\,[\gamma(B_\pi) > \gamma(B)/2] > 1/2$. This implies that $\gamma(B)/2 > \gamma(B_\pi)$ for more than $|G|/2$ permutations $\pi \in G$. To collect these permutations, for every distinct block A of B we define the set $H_A \equiv \{\pi \in G \mid \pi(A) = A\}$. The set H_A contains all the permutation which leave the member of the block A unaffected. Hence, by the pigeon hole principal, there exists a a block $A \in B$ such that $|H_A| > |G|/2$. (Suppose, then, not all the blocks in B remain unaffected for $\leq |G|/2$. This counters our supposition that $\gamma(B)/2 > \gamma(B_\pi)$ for more than $|G|/2$ permutations $\pi \in G$.) Now, we observe that H_A forms a subgroup of G, because H_A is closed under group operation and inverses.

- *Group operation*: $\forall \pi_1, \pi_2 \in H_A$, if $\pi_1(A) = A$ and $\pi_2(A) = (A)$, then obviously $\pi_1 \cdot \pi_2(A) = \pi_1(A) = A$.
- *Inverses*: $\pi(A) = A \Longrightarrow A = \pi^{-1}(A)$.

Our supposition implies that $|H_A| > |G|/2$, but this is a contradiction to Lagrange's theorem. The lemma follows. ∎

Having developed the necessary machinery, we are ready to sketch our algorithm for computing orbits: start with the partition $B^0 \equiv \{[1],[2],\ldots,[n]\}$, and (independently) choose $m = \alpha c \log(n)$

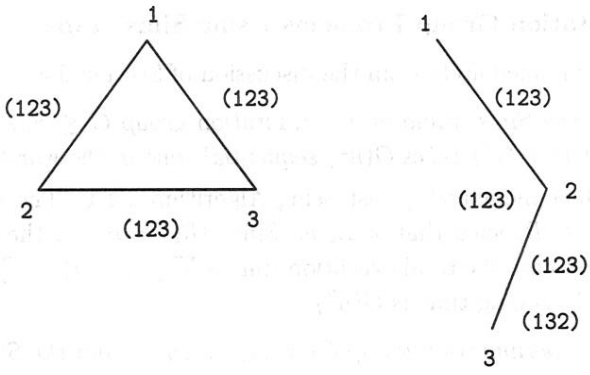


Figure 4. First iteration of the Sims' table algorithm.

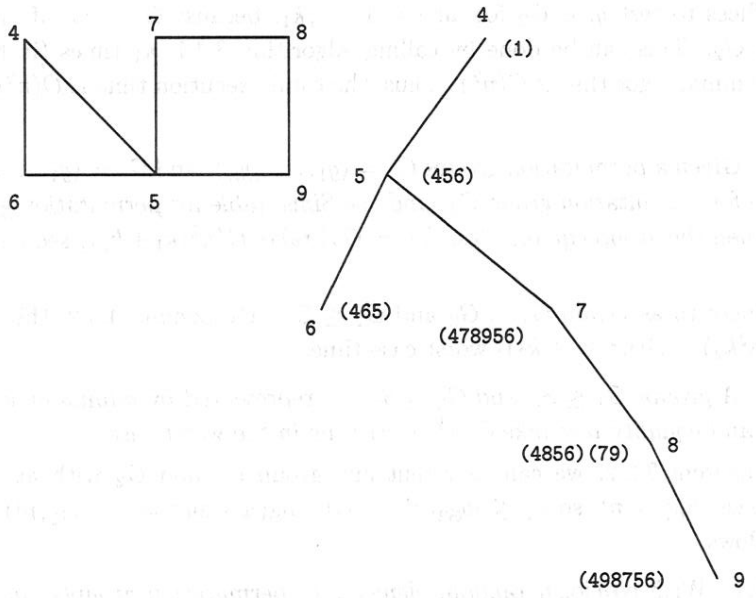


Figure 5. Computation of R_4 from spanning tree.

THEOREM 4.3.1. *Given $(\alpha+1)cn\log n$ examples of a fixed permutation group $G \leq S_n$, we can use the Algorithm 4.3.1 to construct a Sims' table for G in expected sequential time of $O(n^3 \log n)$ with very high probability ($\geq 1 - 1/n^\alpha$ for sufficiently large constant c given any large constant $\alpha > 1$).*

PROOF. In Algorithm 4.3.1, we shall fix $m = (\alpha+1)cn\log(n)$, where $c > 1$ is a sufficiently large constant, and $\alpha \geq c$.

Let us first prove the correctness of this algorithm. Assume inductively that for $i > 1, L_{i-1}$ is a list of $(n-i)(\alpha+1)c\log(n)$ elements of UNIFEX(G_{i-1}). By Lemma 4.2.2, with probability $\geq 1 - n^{-\alpha}$, V_i is the G_i-orbit (of G_i) containing i. We can compute a spanning tree T_i and its preorder traversal in sequential time $O(n^2)$ using depth-first search since there are at most n vertices. Observe that for all $j \in V_i$, the permutation $r_{i,j}$ is in G_{i-1} such that $r_{i,j}(i) = j$. Hence, $R_i = \{r_{i,j} \mid j \in V_j\}$ is a complete set of coset representatives for G_{i-1}/G_i, as required. Furthermore, $L_i = \{r_{i,\pi(i)}^{-1} \cdot \pi \mid \pi \in F_{i-1}\}$ is a list of $(n-i-1)(\alpha+1)c\log(n)$ elements of UNIFEX(G_{i-1}). Thus, $R_1, \ldots, R_h$ is the Sims' table with very high probability $\geq 1 - 1/n^\alpha$.

The complexity of the algorithm can be calculated by observing that the most costly step of each iteration is Step 10. Step 10 takes $O(n^2\log(n))$ time because there are $O(n\log(n))$ elements in F_{i-1}, and each it takes $O(n)$ operations to compute product of two permutations of n elements. Since there are n iterations of the loop, the total time complexity is $O(n^3\log(n))$. ∎

4.4. Solving Permutation Group Problems Using Sims' Table

Three lemmas follow immediately from the discussion of Section 3.4.

LEMMA 4.4.1. *Given the Sims' table of a permutation group $G \leq S_n$, and an input $x \in S_n$, Sims' membership test ($x \in G$?) takes $O(n^2)$ sequential time in the worst case.*

PROOF. We can conduct membership test using Algorithm 3.4.1. The critical step in this algorithm is to find a $y \in R_i$ such that $x \equiv_i y$. Since $|R_i| = n - i$, the cost of the i^{th} step is $n-i$. Thus, in the worst case the total execution time is $\sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n-1}(i) = (n(n-1))/2$. Consequently, the total running time is $O(n^2)$. ∎

LEMMA 4.4.2. *Given a permutation group $G_1 = \langle g_1, \ldots, g_{k_1} \rangle$, and the Sims' table for a permutation group G_2, where $G_1, G_2 \leq S_n$, then the group inclusion test ($G_1 \leq G_2$) takes $O(n^2 k_1)$ sequential time in the worst case.*

PROOF. It suffices to test $g_i \in G_2$ for all $i = 1, \ldots, k_1$, because $G_1 \leq G_2$ if and only if $\forall i = 1, \ldots, k_1$, $g_i \in G_2$. This can be done by calling Algorithm 3.4.1, k_1 times (in the worst case). By the above lemma, algorithm is $O(n^2)$. Thus, the total execution time is $O(n^2 k_1)$ in the worst case. ∎

LEMMA 4.4.3. *Given a permutation groups $G_1 = \langle g_1, \ldots, g_{k_1} \rangle$, and $G_2 = \langle g_1, \ldots, g_{k_2} \rangle$, as well as the Sims' table for permutation group G_1, and the Sims' table for permutation group G_2, where $G_1, G_2 \leq S_n$, then the group equality test ($G_1 = G_2$) takes $O(n^2(k_1 + k_2))$ sequential time in the worst case.*

PROOF. It suffices to ascertain $G_1 \leq G_2$ and $G_2 \leq G_1$. By Lemma 4.4.2, this can be done in $O(n^2 k_1) + O(n^2 k_2) = O(n^2(k_1 + k_2))$ worst case time. ∎

LEMMA 4.4.4. *A groups $G_1 \leq S_n$ and $G_2 \leq S_n$ are represented by minimum set of generators, then inclusion and equality test take $O(n^3 \log n)$ time in the worst case.*

PROOF. By Theorem 3.2.2, we can represent any group G_1 and G_2 with at most $\log_2(|S_n|)$ generators. Since $|S_n| = n!$, so $k_1 \leq \log_2(n!) = O(n \log(n))$ and $k_2 \leq \log_2(n!) = O(n \log(n))$. The lemma follows. ∎

THEOREM 4.4.1. *With $c n \log n$ random elements of permutation group(s) in S_n, we can ascertain permutation group membership, group inclusion, and group inequality in $O(n^3 \log(n))$ expected time. These bounds hold with very high probability $\geq 1 - n^{-\alpha}$ for any sufficiently large constant $\alpha > 1$.*

PROOF. Using Algorithm 4.3.1, we can construct Sims' table in expected sequential time of $O(n^3 \log n)$ with very high probability $\geq 1 - n^{-\alpha}$ for sufficiently large constant $\alpha > 1$. Using Sims' table, we can perform group membership, inclusion, and equality test in $O(n^3 \log(n))$ worst case time. The corollary follows. ∎

THEOREM 4.4.2. *Let A be class of representations of an arbitrary group G in terms of Sims' table. Then class A is learnable.*

PROOF. Algorithm 4.3.1 is a polynomial time algorithm, which computes a (correct) representation from examples of elements of G, with very high success probability. This representation can be used to ascertain group membership, group inclusion, and group equality in polynomial time. ∎

5. PARALLEL ALGORITHMS FOR PERMUTATION GROUPS

5.1. An Important Result from Graph Algorithms

We will assume CRCW PRAM (Concurrent Read, Concurrent Write Parallel Random Access Machine) model described in [19] (see also [16,17]). Shiloach and Vishkin [19] show the following lemma.

random permutations from G. For every $i = 1, 2, \ldots, m$ in succession, compute B^i from $B^{i-1}_{\pi_i}$ by using π_i to merge blocks in B^i by the process outlined earlier. Repeat this procedure until $|B^i| =$ the number of distinct orbits in G (which is equal to k). We label the i^{th} stage as successful if $|B^i| = k$ (i.e., $\gamma(B^i) = 0$) or $\gamma(B^i) \leq \gamma(B^{(i-1)})/2$. Observe that after at most $\log n$ successes (at any stage before m) $|B^m| = k$, because $k \geq 1$. Furthermore, this B^m is exactly the set of G-orbits.

LEMMA 4.2.4. *Let X is a binomial random variable with $m = \alpha c \log n$ trials, each with independent success probability of $1/2$. For sufficiently large c,* Prob $[X \geq \log n] \geq 1 - 1/n^{\alpha}$.

PROOF. This lemma is a result of known probabilistic bounds of [21,22]. ∎

The above discussion leads us to an algorithm for computing G-orbits.

ALGORITHM 4.2.1. COMPUTING G-ORBITS.

INPUT: m random examples from permutation group G.
OUTPUT: G-blocks of group G.
BEGIN

```
Initialize B = {[1], [2], ..., [n]};
for i = 1 to m do
    begin
        If there exist two different blocks b1, b2 ∈ B,
        and ∃ i ∈ {1, ..., n}, such that i ∈ b1 and π(i) ∈ b2,
            then merge b1 and b2;
    end;
return (B);
```

END.

For merging blocks, we can use an efficient set union algorithm.[2]

EXAMPLE 4.2.1. Consider $S_6 = \{\alpha^i\beta^j\gamma^k\phi^l\rho^m \mid 0 \leq i \leq 5, 0 \leq j \leq 4, 0 \leq k \leq 3, 0 \leq l \leq 2, 0 \leq m \leq 1\}$. It is easy to verify that almost any couple of examples from S_6 would be sufficient. For instance, let $\pi_1 = \beta\gamma = (246)(135)$ and $\pi_2 = \alpha^5\phi^2\rho = (16432)$. Thus,

$$\begin{aligned} B^0 &= \{[1], [2], [3], [4], [5], [6]\}, \\ \Longrightarrow B^1 &= \{[1, 3, 5], [2, 4, 6]\}, \\ \Longrightarrow B^2 &= \{[1, 2, 3, 4, 5, 6], \}. \end{aligned}$$

One of the worst cases would be $\pi_1 = (56)$, $\pi_2 = (456)$, $\pi_3 = (3456)$, $\pi_4 = (23456)$, $\pi_5 = (123456)$. In this case, five examples are necessary to compute G-orbits. Such cases are quite unlikely.

THEOREM 4.2.1. *Algorithm 4.2.1 successfully computes the G-orbits of the input group G with very high success likelihood (at least $1 - 1/n^{\alpha}$ for any constant α) by at most $c\alpha \log n$ calls to a set union algorithm (for a sufficiently large constant c). Algorithm 4.2.1 requires at most $c\alpha \log n$ examples in order to accomplish its objective.*

PROOF. The proof of this theorem follows from Lemmas 4.2.3 and 4.2.4. ∎

Considering the fact that there is deep underlying interaction between groups and graphs, it is not surprising that computing G-orbits is intimately related to determining connected components of graphs (Lemma 4.2.2). This leads us to the following theorem.

THEOREM 4.2.2. *There exists a constant $c > 0$ such that for all sufficiently large α, if $m = \alpha c \log(n)$, and if $\pi_1, \ldots, \pi_m$ are independently chosen random elements of G, then with probability*

[2]We shall not indulge in further discussion of set union algorithms since it is a tangential topic.

at least $1 - 1/n^{\alpha}$, *the G-orbits are the connected components of graph with vertex set* $V = \{1, \ldots, n\}$, *and edge set* $E = \cup_{i=1,\ldots,m} E_{\pi_i}$.

PROOF. The proof of this theorem employs Lemma 4.2.2. From Theorem 4.2.1, with a probability $1 - 1/n^{\alpha}$, B^m is the G-orbit of the group G. Furthermore, B^m is exactly the set of connected components of the graph $(V, E) \equiv (\{1, \ldots, n\}, E_{\pi_1} \cup \cdots \cup E_{\pi_m})$. ∎

COROLLARY 4.2.1. *We can compute the G-orbits using* $O(\log n)$ *elements from* UNIFEX(G) *in* $O(n \log n)$ *sequential time, with error probability* $\langle n^{-\alpha}$ *for any sufficiently large constant* $\alpha \rangle 1$.

PROOF. We use the depth first search algorithm developed by Hopcroft and Tarjan [23] to compute the connected components of a graph. There would be at most n vertices ($\{1, \ldots, n\}$) in the graph. Since $|V| = n$, and there are $O(n \log(n))$ edges ($E_{\pi_1} \cup \cdots \cup E_{\pi_m}$). The corollary follows from the $O(|V| + |E|)$ complexity of the depth-first search algorithm. ∎

4.3. The Construction of the Sims' Table

Unfortunately, it is very expensive to construct the Sims' table by known techniques, and often construction of the Sims' table becomes the bottleneck of the algorithms that use it. The first polynomial time algorithm ($O(n^6)$) was presented by Furst, Hopcroft and Luks [5]. Jerrum [6] (also see [8]) improved this time bound to the best known worst-case bound of $O(n^5)$. We present an algorithm with $O(n^3 \log n)$ time complexity.

ALGORITHM 4.3.1. SIMS' TABLE.

INPUT: $m = (\alpha + 1)cn \log(n)$ elements of $G \leq S_n$.

OUTPUT: The Sims' table for G.

BEGIN

- Let L_0 be a list of independently drawn m random elements using UNIFEX(G);
- for $i = 1$ to n do begin
  1. Let F_{i-1} be the set consisting of the first $(n - i)(\alpha + 1)c \log(n)$ elements of L_{i-1}.
  2. Compute the connected components of the graph
  $$(V, E) \equiv (\{1, \ldots, n\}, \cup_{\pi \in F_{i-1}} E_{\pi}).$$
  3. Let V_i be the connected component containing i.
  4. Compute the spanning tree T_i rooted at i of component V_i.
  5. Label each edge $e \in T_i$ with a permutation $l(e)$, where $\pi \in F_{i-1}$, and $e \in E_{\pi}$. The edge connects the two vertices that are related by π.
  6. Let $r_{i,i}$ be the identity permutation.
  7. In a preorder traversal of tree T_i, iteratively compute for each $j \in V_i - i$ the permutation $r_{i,j} = r_{i,j'} \cdot l(j', j)$, where (j', j) is the edge connecting j to its parent j'.
  8. Assign $R_i \Leftarrow \{r_{i,j} \mid j \in V_i\}$.
  9. $L_i \Leftarrow \phi$.
  10. for each $\pi \in F_{i-1}$ do Add $r_{i,\pi(i)}^{-1} \cdot \pi$ to L_i.
- end
- return the Sims' table $R_1, \ldots, R_n$.

END

Consider $L_0 = \{(123), (456), (5789)\}$. The connected component and the associated spanning tree is given below in Figure 4.

$R_1 = \{(1), (123), (132)\}$ is computed, leaving with $L_1 = \{(1), (456), (5789)\}$, as shown in Figure 5.

Now from Figure 5, $R_4 = \{(456), (465), (478956), (4856)(79), (498756)\}$, leaving $L_2 = \{(1), (5789)\}$.

Finally, we can wind up the algorithm by computing R_5 to be $\{(1), (5789), (58)(79), (5987)\}$.

LEMMA 5.1.1. *Given an undirected graph of $|V|$ vertices and $|E|$ edges, the connected components, a spanning forest, and a preorder of each tree in the forest can all be computed in $O(\log |V|)$ time using $|V| + |E|$ processors.*

Alternatively, we may use the randomized parallel algorithm of [20], which uses logarithmic time with linear work; that is, the product of processors and time is linear.

5.2. Parallel Computation of Orbits and Blocks of Permutations Groups

Suppose $G \leq S_n$ be a permutation group over $\{1, 2, \ldots, n\}$.

THEOREM 5.2.1. *In the worst case, we can compute the orbits of the group $G = \langle g_1, \ldots, g_k \rangle$ in time $O(\log(n))$ using $O(\min\{nk, n^2\})$ processors.*

PROOF. The proof follows from Lemma 4.2.2 and Lemma 5.1.1. From Lemma 4.2.2, we know that G-orbits are the connected components of the graph with vertex set $V = \{1, \ldots, n\}$, and the edge set $E = \cup_{i=1,\ldots,k} E_{g_i}$. This graph has n vertices, and $\min\{nk, n^2\}$ edges. By Lemma 5.1.1, we can compute the connected components in $O(\log n)$ time using $O(\min\{nk, n^2\})$ processors. The theorem follows. ∎

THEOREM 5.2.2. *If G is given by a random representation, we can compute the G-orbits in $O(\log n)$ time using $O(n^2)$ processors, with very high probability $\geq 1 - n^{-\alpha}$ for sufficiently large constant $\alpha > 1$.*

PROOF. The proof follows from Theorem 4.2.2 and Lemma 5.1.1. Suppose we are given m independently chosen random elements of G: $\pi_1, \ldots, \pi_m$. We know from Theorem 4.2.2 that there exists a constant $c > 0$ such that for all sufficiently large α, there is probability at least $1 - 1/n^{\alpha}$, such that if $m = \alpha cn \log(n)$, then the G-orbits are the connected components of graph with vertex set $V = \{1, \ldots, n\}$, and edge set $E = \cup_{i=1,\ldots,m} E_{\pi_i}$. There are n vertices and $\min\{\alpha cn^2 \log n, n^2\} = O(n^2)$ edges, so we should be able to compute the connected components in $O(\log n)$ time using $O(n^2)$ processors. ∎

Suppose we are given a group G represented by $\langle g_1, \ldots, g_k \rangle$. For two distinct elements $a, b \in \{1, \ldots, n\}$, let us construct the undirected graph with vertex set $\{1, \ldots, n\}$, and edge set $E_{a,b} \equiv \{(a, b)\} \cup \{(g_i(a), g_i(b)) \mid 1 \leq i \leq k\}$. Atkinson [4] shows (also see [3]) the following lemma.

LEMMA 5.2.1. *The connected components of $(\{1, \ldots, n\}, E_{a,b})$ containing a is the smallest G-block containing $\{a, b\}$.*

We have essentially reduced the problem of finding G-blocks to the problem of computing undirected graph connectivity.

THEOREM 5.2.3. *The smallest G-blocks can be computed in $O(\log n)$ using $n^2(n + k + 1)$ processors.*

PROOF. There are n vertices and $k + 1$ edges in the graph for each pair (a, b). There are a total of n^2 ordered pairs (a, b). For each of these ordered pairs, we can compute the smallest G-blocks in $O(\log n)$ time using $n + k + 1$ processors (Lemma 5.1.1). If we compute smallest G-blocks in parallel (for all pairs), we can do it in the same amount of time ($O(\log n)$), using $n^2(n + k + 1)$ processors. ∎

THEOREM 5.2.4. *If G is represented by a list of random elements, the smallest G-blocks can be found in expected time $O(\log n)$ using $O(n^3 \log n)$ processors.*

PROOF. There are n vertices and $\alpha cn \log n + 1$ edges in the graph for each pair (a, b). There are a total of n^2 ordered pairs (a, b). For each of these ordered pairs, we can compute the smallest G-blocks in $O(\log n)$ time using $O(n \log n)$ processors (Lemma 5.1.1). If we compute smallest G-blocks in parallel (for all pairs), we can do it in the same amount of time ($O(\log n)$), using $O(n^3 \log n)$ processors. ∎

5.3. Limited Parallelism for General Permutation Group

Sims' group membership algorithm was improved by Furst, Hopcroft and Luks [5] to be a polynomial time algorithm. Nevertheless, it appears to be inherently sequential (cannot be speeded up to polylog time by parallelization). We face several obstacles in our effort to parallelize our randomized algorithms. Our algorithms do not have a polylog running time, but our lower processor bounds can prove to be useful from a practical point of view. We observe the following results.

LEMMA 5.3.1. *Given a Sims' table for a permutation group in S_n, we can execute a membership test ($x \in G$?) in $O(n)$ time using n processors.*

PROOF. If we analyze the membership testing algorithm (Algorithm 3.4.1), it is easy to see that, at each of the n levels, we can search in parallel for $y \in R_i$ (such that $x \equiv_i y$) using n processors in $O(1)$ time. As a result, we can ascertain membership in $O(n)$ time using n processors. (Note the height, $h = n$.) ∎

LEMMA 5.3.2. *Given a permutation group $G_1 = \langle g_1, \ldots, g_{k_1} \rangle$, and the Sims' table for a permutation group G_2, where $G_1, G_2 \leq S_n$, then the group inclusion test ($G_1 \leq G_2$) takes $O(n)$ time using $O(nk_1)$ processors.*

PROOF. It suffices to test $\forall i = 1, \ldots, k_1,\ g_i \in G_2$ because $G_1 \leq G_2$ if and only if $\forall i = 1, \ldots, k_1,\ g_i \in G_2$. This can be done by using parallel membership algorithm for all k_1 generators (simultaneously). We can do this $O(n)$ time using $O(nk_1)$ processors. ∎

LEMMA 5.3.3. *Given permutation groups $G_1 = \langle g_1, \ldots, g_{k_1} \rangle$ and $G_2 = \langle g_1, \ldots, g_{k_2} \rangle$, as well as the Sims' table for permutation group G_1, and the Sims' table for permutation group G_2, where $G_1, G_2 \leq S_n$, then the group equality test ($G_1 = G_2$) takes in $O(n)$ time using $n(k_1 + k_2)$ processors.*

PROOF. It suffices to ascertain $G_1 \leq G_2$ and $G_2 \leq G_1$. By Lemma 5.3.2, this can be done in $O(n)$ time using $n(k_1 + k_2)$) processors. ∎

LEMMA 5.3.4. *A Sims' table can be constructed from a random representation of a given permutation group $G \leq S_n$, in expected $O(n \log n)$ time using n^2 processors.*

PROOF. All the steps in the main loop in Algorithm 4.3.1 can be performed in $O(\log n)$ time using at most n^2 processors. Since there are n executions of the main loop, the total time of execution is $O(n \log n)$ using n^2 processors. ∎

COROLLARY 5.3.1. *Suppose we are given a random representation of a group $G \leq S_n$ (conforming to the requirements of Algorithm 4.3.1). We can ascertain permutation group membership, inclusion and equality can all be done in $O(n \log n)$ expected time using $O(\max\{n^2, n(k_1 + k_2)\}$ processors.*

PROOF. The proof follows from Lemmas 5.3.1–5.3.4. ∎

6. POLYLOG TIME ALGORITHMS FOR TWO-GROUPS

6.1. The Structure of Two-Groups

In the previous sections, we traded efficiency for generality. However, if we restrict our attention to two-groups, then we can construct polylog time learning algorithms for group theoretic problems.

DEFINITION 6.1.1. TWO-GROUPS. *A finite group G is a two-group if every element is of the order 2^l for some integer l. (l is not an invariant for a given group, i.e., all elements are not necessarily of the same order.)*

Very frequently, two-groups prove to be useful in a vast variety of applications. In this section, we will show that any two-group has a certain subgroup tower of height $h = \lfloor \log n \rfloor$. Using

this, we can test for membership from the generators in $O(\log n)^3$ time using $n^{O(1)}$ processors. Moreover, if G is given by random presentation, we can construct such a tower in $O((\log n)^3)$ time using $n^{O(1)}$. Both problems are in class NC.

6.2. Structure Forest Representation of Two-Groups

We can represent the group S_n where $n = 2^a$, for some integer a, by automorphisms of a binary tree with n leaves (one for each element in the object set). In the same spirit, we can construct a *structure forest* F_G of complete binary trees for a two-group G_{II}. Each structure tree is a complete binary tree, and can be identified by its leaves. The set of leaves belonging to each structure tree is an orbit in G_{II}. Hence, G_{II} is a subgroup of natural direct product of the automorphism groups of the *structure trees* in F_G. Any two-group G_{II} can be decomposed into a subgroup of the natural direct products of the iterated wreath products. It follows that if B_1 and B_2 are the set of leaves of two immediate subtrees of a (nonleaf) structure tree T, then $\{B_1, B_2\}$ are required to be G-blocks in the set of leaves of T.

LEMMA 6.2.1. *The structure forest F_G of two-group $G \leq S_n$ can be constructed from the generators of the group in $O(\log n)^2$ time using $O(n^2)$ processors.*

PROOF. Suppose we are given the set of generators $\{g_1, \ldots, g_k\}$ of a two-group $G_{II} \leq S_n$. By executing first the G-orbit Algorithm 4.2.1, we can compute the orbits of the group G_{II}. This can be done in $O(\log n)$ parallel time using $O(n \log n)$ processors (cf. Corollary 4.2.1). Now, we can use the method suggested in Section 5.2 to compute G-blocks. This can be done in $O(\log n)$ using $O(n^2)$ processors (see Theorem 5.2.2). Now the structure forest can be constructed in $O((\log n)^2)$ time by examination of each G-block. The structure forest becomes the cornerstone in the design of efficient algorithms. The proof follows. ∎

6.3. Root Flips are Linear for Two-Groups

In this section, we will prove that root flips are linear, to exhibit that membership testing can be reduced to solving linear system over $GF(2)$. Let G_{II} be a two-group generated by $\langle g_1, \ldots, g_k \rangle$.

Let $a_1, \ldots, a_t$ be the roots of the structure forest F_G. We say that $\pi \in G$ *flips* root a_i if π permutes the two children of a_i. For any $\pi \in S_n$, let $\vec{A}(\pi) = (a_1(\pi), \ldots, a_r(\pi))^{\mathsf{T}}$, where $a_i(\pi) = 1$ if π flips the root r_i, otherwise $a_i(\pi) = 0$. Thus, $\vec{A}(\pi)$ is a Boolean column vector which has 1 in the i^{th} position (row), if π flips the i^{th} root. The next lemma follows.

LEMMA 6.3.1. *For $\vec{A}$ defined above, the permutations are commutative with respect to root flips: $\forall \pi_1,\ \pi_2 \in S_n,\ \vec{A}(\pi_1 \cdot \pi_2) = \vec{A}(\pi_2 \cdot \pi_1)$.*

PROOF. Basically $a_i(\pi_1 \cdot \pi_2) = 1$, if and only if exactly one of the two permutations π_1 and π_2 flip the root a_i of the i^{th} structure tree. Thus, $\vec{A}(\pi_1 \cdot \pi_2) = \vec{A}(\pi_1) \oplus \vec{A}(\pi_2) = \vec{A}(\pi_1) + \vec{A}(\pi_2) (\mathrm{mod}\, 2)$. Using the commutativity property $\vec{A}(\pi_1) + \vec{A}(\pi_2) (\mathrm{mod}\, 2) = \vec{A}(\pi_2) + \vec{A}(\pi_1) (\mathrm{mod}\, 2) = \vec{A}(\pi_2) \oplus \vec{A}(\pi_1) = \vec{A}(\pi_2 \cdot \pi_1)$. ∎

Thus, permutations flip commutatively on the roots of the structure forest. Now, we define a $r \times k$ Boolean matrix M. We define i^{th} column to be exactly the column vector $\vec{A}(g_i)$. Thus, the element $m_{i,j}$ (in row major form) to be equal to 1 if g_i flip a_j. Let $x \in \{0 \mid 1\}^k$. Let $\vec{A}(G) \equiv \{\vec{A}(\pi) \mid \pi \in G\}$. The next lemma follows.

LEMMA 6.3.2. *For the matrix M defined above, $\vec{A}(G)$ is the linear space $\{Mx \mid x \in \{0,1\}^k\}$ over $GF(2)$.*

PROOF. Suppose $\pi \in G$, then $\pi = g_{j_1} \ldots g_{j_d}$ where $\forall i = 1, \ldots, d$, the element $g_{j_i} \in \{g_1, g_2, \ldots, g_k\}$. This is the canonical factorization of π. Now, for all $i = 1, \ldots, k$, let x_i be 0 if g_i occurs an even number of times in the canonical factorization of π, otherwise $x_i = 1$. So, x_i determines whether the g_i factors in π have any effect on the root flips. By the previous

lemma, $\vec{A}(\pi) = \vec{A}(g_1^{x_1} \cdots g_k^{x_k}) = \vec{A}(g_1^{x_1}) \cdots \vec{A}(g_k^{x_k})$. Now, the j^{th} element of $\vec{A}(\pi)$ is the sum $\sum_i^k x_i (\text{mod } 2)$. It follows from the definition that $\vec{A}(\pi) = Mx$. ∎

Let G_1 be the subgroup of G consisting of permutations that fix the roots $a_1, \ldots, a_r$. So $G^{(1)} = \{\pi \in G \mid \vec{A}(\pi) = (0, \ldots, 0)^\top\}$. The next lemma follows.

LEMMA 6.3.3. *$\pi \in G$ if and only if $\exists x \in \{0,1\}^k$ such that $Mx = \vec{A}(\pi)$ and $(g_1^{x_1} \cdots g_k^{x_k})^{-1}\pi \in G^{(1)}$.*

PROOF. If $\exists x \in \{0,1\}^k$ such that $Mx = \vec{A}(\pi)$ and $(g_1^{x_1} \cdots g_k^{x_k})^{-1}\pi \in G^{(1)}$, then $\pi \in G$ because $(g_1^{x_1} \cdots g_k^{x_k}) \in G$ (using closure property of groups).

On the other hand, if $\pi \in G$, then by Lemma 6.2.2, $\exists x \in \{0,1\}^k$ such that $Mx = \vec{A}(\pi)$, and so $\vec{A}((g_1^{x_1} \cdots g_k^{x_k})^{-1}) = \vec{A}(\pi)$. Hence, $\vec{A}((g_1^{x_1} \cdots g_k^{x_k})^{-1}\pi) = (0, \ldots, 0)^\top$, so $(g_1^{x_1} \cdots g_k^{x_k})^{-1}\pi \in G$. The lemma follows. ∎

6.4. Two-Group Membership Testing Using Block Structure Tower

Let the *block structure tower* of G_{II} be a sequence of subgroups $G_{II} = G_0 > G_1 > \cdots > G_h = I$. We require that G_i contains only the permutations that fix all nodes of depth less than i in the structure forest F_G. Since the depth of F_G is at most $\lfloor \log n \rfloor$, the tower has height, $h \leq \lfloor \log n \rfloor$. This leads to an important result.

ALGORITHM 6.4.1. TWO-GROUP MEMBERSHIP TEST.

INPUT: structure forest F_G and a permutation π to be tested for membership in G_{II}.

OUTPUT: $x \in G_{II}$ or $x \notin G_{II}$.

BEGIN

```
Initialize G_0 ← G_II;
for i = 0 to h
begin
   Define M and A for G_i;
   If there is a solution to Mx = A(π)
     then solve for x and let π = (g_1^{x_1} ··· g_k^{x_k})^{-1} π
     else return ("x ∉ G_II");
end
return ("x ∈ G_II")
```

END

Let ω be the smallest real number such that $n \times n$ matrix product can be done in logarithmic time using n^ω processors.

THEOREM 6.4.1. *Suppose we have generators for the block structure tower $G_{II} = G_0 > G_1 > \cdots > G_h = I$ of the two-group G_{II}. Then for the worst case input, we can test membership in G in $O(\log n)^3$ expected time using n^ω processors.*

PROOF. Given a permutation $\pi \in S_n$ $(> G_{II})$, we want to ascertain $\pi \in G$? In order to do this, we first determine if there exists a solution $x \in \{0,1\}^k$ to the linear equation $Mx = \vec{A}(\pi)$, for M and A defined previously. If a solution does not exist, then we can immediately reject π. Otherwise, we reduce to problem to testing $(g_1^{x_1} \cdots g_k^{x_k})^{-1}\pi \in G_1$. We continue this process iteratively until we determine whether or not $x \in G$ (as shown in the previous algorithm). Each stage requires a rank test and a solution of a linear system of size at most $n \times n$, over $GF(2)$. This can be accomplished by the parallel algorithm of [24] in $O(\log n)^2$ expected time using n^ω processors. This test is repeated at most $\log n$ times. So the total running time is $O(\log n)^3$ using n^ω processors. ∎

6.5. Construction of Block-Structure Tower from Random Examples

Let $L = \{\pi_1, \ldots, \pi_{m_0}\}$ be list of m_0 permutation independently chosen from UNIFEX(G_{II}). We fix $m = m_0/\log n$. Let Y be the linear space over $GF(2)$ generated by $\vec{A}(\pi_1), \ldots, \vec{A}(\pi_m)$. Since $\vec{A}(G)$ is a group of size $\leq 2^n$. As a consequence of Theorem 2, we have the following lemma.

LEMMA 6.5.1. *If $m \geq 1 + n' \log(1/\epsilon_1)$, then* Prob $(Y = \vec{A}(G)) \geq 1 - \epsilon$, *where n' are the number of nodes in the structure forest and $\epsilon_1 = 1 - (1-\epsilon)^{1/n'}$.*

PROOF. Substituting $|G| \leq 2^{n'}$ in Theorem 3.2.1, we get the desired bound. ∎

REMARK 6.5.1. Since the total number of nodes of the structure forest is $2n$, we can bound the probability of error to be at most ϵ using $m = 1 + 2n \log(1/(1-(1-\epsilon)^{1/2n}))$. For $\epsilon = n^{-\alpha}$, $m = o(n)$ elements suffice.

Let M' be an $r \times m$ matrix such that $\forall i = 1, \ldots, m$ its i^{th} column is $\vec{A}(\pi_i)$. We construct $\vec{Y} = \{M'x \mid x \in \{0,1\}^m\}$. Let $Y_1, \ldots, Y_l$ be the basis for Y. Now $\forall i = 1, \ldots, l$, we find x^i such that $M'x^i = Y_i$, and using x^i define the permutation $\sigma_i = \pi_1^{x_1^i} \cdots \pi_m^{x_m^i}$. Since for all valid $i : \vec{A}(\sigma_i) = y_i$, we have (by construction) $Y = \vec{A}(\langle \sigma_1, \ldots, \sigma_l \rangle)$.

LEMMA 6.5.2. *The set $R = \{\sigma_1^{z_1} \cdots \sigma_l^{z_l}\}$ contains a complete set of coset representatives for G/G_1.*

PROOF. Now, let M'' be the $r \times l$ Boolean matrix whose i^{th} column is $\vec{A}(\sigma_i)$ for $i = 1, \ldots, l$. Again, by construction, we have $Y = \{M''z \mid z \in \{0,1\}^l\}$. For the purposes of membership testing, it is sufficient to have the list $\sigma_1, \ldots, \sigma_l$ which generate the coset representatives of G/G_1. ∎

LEMMA 6.5.3. *For each $\pi \in S_n$, let $f_C(\pi) = (\sigma_1^{z_1} \cdots \sigma_l^{z_l})^{-1}\pi$, where $M^\pi z = \vec{A}(\pi)$. Then*

$$f_C(\text{UNIFEX}(G_{II}) \equiv \text{UNIFEX}(G_1).$$

PROOF. $\forall \pi \in G$, $f_C(\pi) \in G_1$. Hence, the lemma follows from Lemma 3.1.1. ∎

An application of the above lemma leads to the following important result.

LEMMA 6.5.4. *$L_1 = \{f_C(\pi_{m+1}), \ldots, f_C(\pi_{m_0})\}$ is a list of independently chosen examples from G_1.*

PROOF. Since $L = \{\pi_1, \ldots, \pi_{m_0}\}$ is a list of independently chosen elements from UNIFEX(G), and we have utilized only the first $m_0/\log(n)$ elements of L to construct a random representation of G_1, this lemma follows from Lemma 3.1.2 and Lemma 6.5.3. ∎

Lemma 6.5.4 implies that we can repeat the above procedure for construction of G_2 from the random elements of G_1. We redefine $m_0 = m_0(1 - 1/\log n)$, and then proceed in a similar fashion using first $m = m_0/\log n$ elements of the list. After $\log n$ stages of the procedure yields the entire block structure tower.

The linear algebraic computations (such as computing basis vectors), required in each stage of the above construction of G_i, can be done using the method of [24]. This would take $O(\log n)^2$ using n^ω processors. Since there are at most $\log n$ stages, the theorem follows.

THEOREM 6.5.1. *Given a random presentation of a two-group G_{II}, we can construct generators foe each subgroup of the block structure tower in $O((\log n)^3)$ time using n^ω processors.*

PROOF. Lemma 6.5.4 implies that we can repeat the above procedure for construction of G_2 from the random elements of G_1. We redefine $m_0 = m_0(1 - 1/\log n)$, and then proceed in a similar fashion using first $m = m_0/\log n$ elements of the list. Exactly $\log n$ stages of the procedure yield the entire block structure tower. The linear algebraic computations (such as computing basis vectors), required in each stage of the above construction of G_i, can be done using the algorithm of [24]. This would take $O(\log n)^2$ using n^ω processors. Since there are at most $\log n$ stages, the theorem follows. ∎

COROLLARY 6.5.1. *Given a random presentation of (worst case) two-group $G_{II} < S_n$, and some $x \in S_n$, we can test membership in G_{II} in expected time $O(\log n)^3$ using n^ω processors.*

PROOF. The proof follows from Theorem 6.4.1 and Theorem 6.5.1. ∎

COROLLARY 6.5.2. *Given random presentations of (worst case) two-group $G_{II}, G'_{II} \leq S_n$, we can ascertain $G_{II} \leq G'_{II}$, and $G_{II} = G'_{II}$ in $O(\log n)^3$ expected time using n^ω processors.*

PROOF. The proof follows from Theorem 6.4.1 and Theorem 6.5.1. ∎

REFERENCES

1. J.H. Reif, Probabilistic algorithms in group theory, In *Foundations of Computation Theory (FCT85)*, Cottbus, Democratic Republic of Germany, September 1985, Lecture Notes in Computer Science, Vol. 199, pp. 341–350, (1985).
2. H. Wielandt, *Finite Permutation Groups*, Academic Press, New York, (1964).
3. C.M. Hoffman, Group theoretic algorithms and graph isomorphism, In *Lecture Notes in Computer Science*, Springer-Verlag, New York, (1982).
4. M.D. Atkinson, An algorithm for finding the blocks of a permutation group, *Math. of Comp.* **29**, 911–913, (1975).
5. M. Furst, J. Hopcroft and E. Luks, Polynomial time algorithms for permutation groups, In *Proc. 21st IEEE Symp. on Foundations of Computer Science*, pp. 36–41, (1981).
6. M. Jerrum, A compact representation for permutation groups, In *Proc. 23rd Symp. on Foundations of Computer Science*, pp. 126–133, Chicago, IL, (November 1982).
7. M. Jerrum, A compact representation for permutation groups, *J. Algorithms* **7**, 60–78, (1986); 126–133, (November 1982).
8. C.A. Brown, L. Finkelstein and P.W. Purdom, An efficient implementation of Jerrum's algorithm, Tech. Rep. NU-CCS-87-19, Northeastern University, Boston, MA, (1987).
9. L. Babai, M. Luks and A. Seress, Fast management of permutation groups, In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pp. 272–282, (1988).
10. G.D. Cooperman, L.A. Finkelstein and P.W. Purdom, Fast group membership using strong generating set for permutation groups, In *Computer and Mathematics*, (Edited by E. Kaltofen and S.M. Watt), pp. 27–36, Springer-Verlag, Berlin, New York, (1989).
11. P. McKenzie, *Parallel Complexity of Permutation Groups*, TR713, Dept. of Computer Science, University of Toronto, (1984).
12. P. McKenzie and S. Cook, Parallel complexity of Abelian permutation group membership problem, In *Proc. 24th Symp. on Foundations of Computer Science*, pp. 154–161, (1983).
13. L. Babai, Monte Carlo algorithms in graph isomorphism testing, Technical Report, D.M.S. No. 79-10, Dept. of Math, Univ. of Montreal, Quebec, (1979).
14. E.M. Luks, Isomorphism of graphs with bounded valence can be tested in polynomial time, In *Proc. 21st Symp. on Foundations of Computer Science*, pp. 42–49, (1981).
15. Z. Galil, C.M. Hoffman, E.M. Luks, C.P. Schnorr and A. Weber, An $O(n^3 \log n)$ deterministic and an $O(n^3)$ probabilistic isomorphism test for trivalent graphs, In *23rd Annual IEEE Symp. on Foundations of Computer Science*, pp. 118–125, Chicago, IL, (November 1982).
16. J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, (1992).
17. J.H. Reif, Editor, *Synthesis of Parallel Algorithms*, Morgan Kaufmann, (1993).
18. L.G. Valiant, A theory of learnable, *Comm. ACM* **27** (11), 1134–1142, (1984).
19. Y. Shiloach and U. Vishkin, An $O(\log n)$ parallel connectivity algorithms, *J. Algorithms* **3**, 57–67, (1982).
20. H. Gazit, An optimal randomized parallel algorithm for finding connected components in a graph, *SIAM Journal on Computing* **20** (6), 1046–1067, (December 1991).
21. H. Chernoff, A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations, *Annals of Math. Stat.* **23**, (1952).
22. D. Angluin and L.G. Valiant, Fast probabilistic algorithms for Hamiltonian paths and matchings, *J. Comp. Syst. Sci.* **18**, 155–193, (1979).
23. J.E. Hopcroft and R.E. Tarjan, Efficient algorithms for graph manipulation, *Comm. ACM* **16** (6), 372–378, (1973).
24. A. Borodin, von zur Gothen and J. Hopcroft, Fast parallel matrix and GCD computations, *Information and Control* **52** (3), 241–256, (1982).