Introduction

This book describes computational methods for studying subgroups and quotient groups of finitely presented groups. The procedures discussed belong to one of the oldest and most highly developed areas of computational group theory. In order to better understand the context for this material, it is useful to know something about computational group theory in general and its place within the area of symbolic computation.

The mathematical uses of computers can be divided roughly into numeric and nonnumeric applications. Numeric computation involves primarily calculations in which real numbers are approximated by elements from a fixed set of rational numbers, called floating-point numbers. Such computation is usually associated with the mathematical discipline numerical analysis. Examples of numerical techniques are Simpson's rule for approximating definite integrals and Newton's method for approximating zeros of functions.

One nonnumeric application of computers to mathematics is symbolic computation. Although it is impossible to give a precise definition, symbolic computation normally involves representing mathematical objects exactly and performing exact calculations with these representations. It includes efforts to automate many of the techniques taught to high school students and college undergraduates, such as the manipulation of polynomials and rational functions, differentiation and integration in closed form, and expansion in Taylor series.

The term "computer algebra" is frequently used as a synonym for "symbolic computation". The books [Akritas 1989], [Buchberger, Collins, & Loos 1983], [Davenport, Siret, & Tournier 1988], [Della Dora & Fitch 1989], and [Geddes, Czapor, & Labahn 1992] all have this phrase in their titles. Although the term "computer algebra" is well established, it conflicts somewhat with current usage within mathematics, where "algebra" usually is used in the narrower sense of "abstract algebra", the study of algebraic structures such as groups, rings, fields, and modules. The word "computer" in the phrase "computer algebra" is also not quite accurate. It is

true that much of what is done is motivated by the existence of computers. Nevertheless, the algebraic algorithms which have been developed represent substantial mathematical achievements, whose importance is not dependent entirely on their being incorporated into computer programs. Many of the algorithms, including a number presented in this book, can be useful in calculations carried out by hand.

Within symbolic computation there is a rapidly expanding area of computational (abstract) algebra, which is the study of procedures for manipulating objects from abstract algebra with particular concern for practicality. Computational group theory is the part of computational algebra which considers problems related to groups. Other flourishing subfields of computational algebra are computational number theory, which is described in such books as [Pohst & Zassenhaus 1989] and [Bressoud 1989], and computational algebraic geometry, where the software package Macaulay of David Bayer and Michael Stillman has had a major impact.

Symbolic computation is at the border between mathematics and computer science. The objects being manipulated are mathematical. However, the algorithmic ideas often have come from computer science, and individuals who identify themselves as computer scientists have made important contributions to the subject. This book emphasizes the close connection between techniques for investigating finitely presented groups and such topics as formal language theory and critical-pair/completion procedures, which are now considered to be important parts of computer science.

A point of continuing debate is the role of complexity theory in symbolic computation. The traditional complexity measure in theoretical computer science is asymptotic worst-case complexity. For users of symbolic software, worst-case analyses are often too pessimistic. Of much more relevance is average-case complexity. However, average-case analyses are lacking for many of the most important algebraic algorithms. Moreover, there are cases in which no agreement has been reached on what the average case is. Symbolic computations often require a great deal of computer memory. Frequently one has to tailor a program to fit a specific problem in order to get a solution on a particular computer. In this situation, it does not make sense to talk about asymptotic behavior.

In computation with finitely presented groups there is an even more fundamental difficulty with complexity. Many problems connected with such groups have been shown not to have general algorithmic solutions. There are procedures for trying to solve some of these problems, but the procedures terminate only if the solution has a particular form. In the remaining cases, the procedures continue indefinitely. There is no version of complexity analysis which adequately handles this situation. Ideally, one would want a program to give up quickly if it is not going to get an answer and to work hard when a solution can be found. However, because usually there

is no quick way to determine from the input data which outcome is likely, deciding when to throw in the towel is difficult.

Although we do not have an adequate theoretical framework on which to base comparisons of group-theoretic procedures and their computer implementations, we still need to make decisions about which techniques to use on a particular problem. Frequently, all that we can do is apply competing methods to a selection of test problems and compare the results. Experimental evidence is better than nothing, but one must be very careful about drawing conclusions from such evidence. It is hard to be sure that the sample problems are truly representative of the class of problems under consideration.

A computational problem in group theory typically begins "Given a group G, determine ...". Our ability to solve the problem depends heavily on the way G is given. There are three methods commonly used to specify a group G:

(a) One may describe G as the subgroup generated by an explicit finite list of elements in some other group H which is considered to be well-known.
(b) One may define G to be the group of automorphisms or symmetries of some combinatorial or algebraic object.
(c) One may give a finite presentation for G by generators and relations.

In (a), the group H could be the symmetric group on a finite set, the group of invertible n-by-n matrices over a commutative ring, or a free group of finite rank. Examples of the types of objects which might arise in (b) are graphs, block designs, and, in the case of Galois groups, finite extensions of one field by another. Problems related to groups given as in (c) are the main subject of this book, although finitely generated subgroups of free groups are also discussed at some length. The greatest successes in computational group theory so far have come in connection with permutation groups on finite sets, finite solvable groups, and finitely presented groups.

The study of groups given by presentations is called combinatorial group theory. The primary references on the subject are the books [Magnus, Karrass, & Solitar 1976] and [Lyndon & Schupp 1977]. The history of the subject given in [Chandler & Magnus 1982] also provides useful insights. Many examples of finite presentations of groups can be found in [Coxeter & Moser 1980].

As noted earlier, the computer science literature is also relevant. The three volumes by Donald Knuth contain a great deal of material related to our topic. Volume 2 is a particularly valuable reference. Other books describing the fundamental algorithms of computer science with applications to computational group theory are [Aho, Hopcroft, & Ullman 1974] and [Sedgewick 1990].

Because this is one of the first texts to cover a major area within computational group theory, some remarks on the history of the field have been included. Most chapters conclude with a brief section of historical notes. More information, particularly about the history of general group theory, combinatorial group theory, and abstract algebra, can be found in [Dieudonné 1978], [Chandler & Magnus 1982], [Waerden 1985], and [Wussing 1984]. Computational problems related to groups have been studied for at least 150 years. Algorithmic questions have been a part of combinatorial group theory from its beginning. In fact, the algorithms discussed in Chapter 8 are based directly on techniques developed in the middle of the nineteenth century, before presentations of groups were defined and before the concept of an abstract group was clarified.

Perhaps the first references to the potential of computers for group-theoretic calculations came from Alan Turing and Maxwell Newman. At the end of World War II, Turing began work on the design of an "automatic computing engine". As quoted in [Hodges 1983], Turing suggested in a document circulated at the end of 1945 that one of the tasks which might be assigned to this engine was the enumeration of the groups of order 720. In connection with the inauguration in 1951 of a computer at Manchester University, Newman described a probabilistic method by which computers might be able to obtain a crude estimate of the number of groups of order 2^8 (M. H. A. Newman 1951).

During the 1950s, group-theoretic calculations using machines were carried out by several researchers. However, it is reasonable to say that the field of computational group theory came into existence with the work of Joachim Neubüser, who was the first individual to make the application of computers to group theory his primary professional activity. Neubüser's first paper appeared in 1960.

By the late 1960s, the use of computers in abstract algebra was sufficiently common for a conference on the subject to be organized in Oxford, August 29 to September 2, 1967. The proceedings [Leech 1970] of that conference contained 35 papers, at least 20 of which dealt with groups. In his survey (Neubüser 1970) in those proceedings, Neubüser listed roughly 130 publications related to the use of computers in group theory. Not all the contributions to computational group theory which grew out of the Oxford conference were reflected in the proceedings; see the notes at the end of Chapters 5 and 11.

During the 1970s, computational group theorists participated frequently in conferences devoted primarily to traditional computer algebra. Thus the proceedings of the Second Symposium on Symbolic and Algebraic Manipulation sponsored by the Association for Computing Machinery (ACM) in 1971, the 1976 ACM Symposium on Symbolic and Algebraic Computation, and EUROSAM '79, an international symposium held in France, all contain important papers on computational group theory. More recently,

computational group theory has been treated for the purpose of organizing conferences as a field of its own or as a subfield within group theory. The first conference devoted exclusively to computational group theory was organized in Durham in 1982 by the London Mathematical Society. The proceedings [Atkinson 1984] of the Durham meeting give a good indication of the breadth of the field.

Let us conclude this introduction with a brief survey of the software currently available for symbolic computation. This is an area of rapid change, so the information given here can be expected to become outdated fairly quickly. The classical symbolic systems provide facilities for working with individual polynomials, rational functions, matrices, Taylor series, and similar objects, primarily with coefficients taken from the real numbers. Among the classical systems currently available are REDUCE and MACSYMA, which have been around for some time, and Maple and Mathematica, which are considerably newer. Axiom (formerly SCRATCHPAD), developed at IBM's Yorktown Heights laboratory, is a system which allows the user some flexibility to work with elements of more general algebraic systems. A new version, written in C, of George Collins's system SAC2 has recently been introduced. Henri Cohen and collaborators have created a package called Pari, which is aimed primarily at number theory but contains an implementation of the LLL algorithm discussed in Chapter 8. The system Macaulay referred to earlier supports computation in algebraic geometry and commutative algebra.

Although quite a bit of group-theoretic software is available, no single package contains implementations of all the procedures discussed in this book. The most mature system for group-theoretic computation is Cayley, developed at Sydney University by John Cannon. A recent addition is the system GAP designed under the direction of Joachim Neubüser at the Technische Hochschule in Aachen. Various programs of a more specialized nature have been written at the Australian National University in Canberra under the leadership of Michael Newman. George Havas of the University of Queensland in Brisbane has produced free-standing implementations of several of the procedures presented here. David Epstein and Derek Holt at the University of Warwick have produced implementations of the Knuth-Bendix procedure for strings described in Chapter 2, as well as a number of other programs for studying "automatic groups" and for exploring the possibility that two given finitely presented groups are isomorphic.

1

Basic concepts

It was not possible to make the exposition in this book self-contained and keep the book to a reasonable length. In a number of places, results from various parts of mathematics and theoretical computer science are stated without proof. This chapter reviews several of the most fundamental concepts with which the reader is assumed to be somewhat familiar. More material on sets, monoids, and groups can be found in algebra texts like [Sims 1984] and books on group theory such as [Hall 1959]. Two standard sources for topics in computer science are [Knuth 1973] and [Aho, Hopcroft, & Ullman 1974]. After a brief discussion of some concepts in set theory, we define monoids, summarize some important facts about groups, consider presentations of monoids and groups by generators and relations, contemplate some sobering facts about our ability to compute using presentations, agree upon a method for describing computational procedures, look at some basic algorithms for computing with integers, and study an important search technique. Readers to whom most of these topics are familiar may wish to proceed directly to Chapter 2 and refer back to this chapter as needed.

1.1 Set-theoretic preliminaries

The concept of a set is central to the formal exposition of mathematics. Despite the importance of set theory, most mathematics texts do not make explicit the axioms for set theory being used. Instead, they adopt a "naive" set theory based largely on intuition.

This book will be no exception. We shall assume that the reader has a basic understanding of *sets* and *set membership*, as well as the operations of *union*, *intersection*, and *difference* of sets. The *empty set* will be denoted $\emptyset$. If A is a *subset* of B, then we shall write $A \subseteq B$ or $B \supseteq A$. To indicate that A is a *proper subset* of B, we shall write $A \subset B$ or $B \supset A$. The *cartesian product* of sets X and Y is the set $X \times Y$ of all ordered pairs (x, y) with

x in X and y in Y. The *diagonal* of $X \times X$ is $D = \{(x,x) \mid x \in X\}$. For a finite set A, the *cardinality* of A will be denoted $|A|$.

A *relation* from X to Y is a subset R of $X \times Y$. If $X = Y$, then we say that R is a relation on X. If (x,y) is in R, then we sometimes write xRy. Suppose A is a subset of X. Then AR is defined to be

$$\{y \in Y \mid aRy \text{ for some } a \text{ in } A\}.$$

The *inverse* of R is the relation $R^{-1} = \{(y,x) \mid (x,y) \in R\}$.

Let R be a relation from X to Y and let S be a relation from Y to Z. The *composition* of R and S is the relation T from X to Z which consists of the pairs (x,z) such that there is an element y in Y with the property that xRy and ySz. We write $T = R \circ S$.

An *equivalence relation* on a set X is a relation E on X such that the following conditions hold:

(i) xEx for all x in X.
(ii) If xEy, then yEx.
(iii) If xEy and yEz, then xEz.

These conditions are called *reflexivity*, *symmetry*, and *transitivity*, respectively. Symmetry can be stated as $E^{-1} = E$, and transitivity means that $E \circ E$ is contained in E. Suppose E is an equivalence relation on X. Then for x and y in X, the sets $\{x\}E$ and $\{y\}E$ are either equal or disjoint. Thus $\Pi = \{\{x\}E \mid x \in X\}$ is a *partition* of X. The elements of Π are called the *equivalence classes* of E.

A *function* from X to Y is a relation f from X to Y such that for all x in X the set $\{x\}f$ has exactly one element. We write $f\colon X \to Y$ and refer to X as the *domain* of f. If y is the unique element of $\{x\}f$, then we write $y = xf$, $y = f(x)$, $y = x^f$, or $x \overset{f}{\longmapsto} y$, depending on the context. The composition of two functions is a function. The term "map" is a synonym for "function". Let E be an equivalence relation on X and let Π be the set of equivalence classes of E. The *natural map* from X to Π is the function π such that $\pi(x) = \{x\}E$ for all x in X.

Let f be a function from X to Y. We say that f is *surjective* or that f maps X *onto* Y if $Xf = Y$. We say that f is *injective* if for all x_1 and x_2 in X the equality $f(x_1) = f(x_2)$ implies that $x_1 = x_2$. A function which is both injective and surjective is said to be *bijective*, or to be a *one-to-one correspondence*. Functions which are surjective, injective, or bijective are called *surjections*, *injections*, or *bijections*, respectively. A *permutation* of a set X is a bijection from X to itself. The *identity function* on a set X is the function e such that $e(x) = x$ for all x in X. Clearly e is a permutation of X.

A convenient way to describe a function on a small set is by a matrix with two rows. The first row lists the domain and the second row gives the images. For example,

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 2 & 4 & 0 & 1 & 3 \end{pmatrix}$$

defines the permutation of $\{0, 1, 2, 3, 4\}$ which takes 0 to 2, 1 to 4, and so on.

Exercises

1.1. Show that the intersection of any nonempty collection of equivalence relations on a set is again an equivalence relation.
1.2. Suppose that $E_1, E_2, \ldots$ is an infinite sequence of equivalence relations on X and E_i is contained in E_{i+1} for $i \geq 1$. Prove that the union of the E_i is an equivalence relation on X.
1.3. Show that for any set X there is a bijection from the set of equivalence relations on X to the set of partitions of X.
1.4. Suppose $f: X \to Y$, $g: Y \to Z$, and $h: Y \to Z$ are functions. Assume that f is surjective and $f \circ g = f \circ h$. Show that $g = h$.

1.2 Monoids

A *semigroup* is a pair $(S, \bullet)$ consisting of a set S and an *associative binary operation* $\bullet$ *on* S. That is, $\bullet$ is a function from $S \times S$ to S, and if we write the image under $\bullet$ of a pair (s, t) as $s \bullet t$, then

$$s \bullet (t \bullet u) = (s \bullet t) \bullet u$$

for all s, t, and u in S. If $s \bullet t = t \bullet s$ for all s and t in S, then $\bullet$ is said to be *commutative*. In this book, multiplicative notation will normally be used for binary operations. Thus $s \bullet t$ will be written st and referred to as the *product* of s and t. When the binary operation is clear from the context, we refer to S as the semigroup. If S is finite, then the *order* of S is $|S|$.

An *identity element* in a semigroup S is an element e of S such that $es = se = s$ for all s in S. There is at most one identity element in S, and if S has an identity element, then S is called a *monoid*. Note that a semigroup may be empty, but a monoid always contains at least one element, its identity element. When multiplicative notation is used, the identity element is normally denoted by 1.

Let M be a monoid with identity element 1. An element u of M is called a *unit* if there is an element v of M such that $uv = vu = 1$. The element v is unique. We call v the *inverse* of u and write $v = u^{-1}$. If u is a unit with inverse v, then v is a unit with inverse u. The inverse of 1 is 1.

Suppose that x and y are units in M. Then

$$(xy)(y^{-1}x^{-1}) = x(yy^{-1})x^{-1} = x1x^{-1} = xx^{-1} = 1.$$

Similarly $(y^{-1}x^{-1})(xy) = 1$, so xy is a unit with inverse $y^{-1}x^{-1}$. Therefore the set of units is closed under multiplication.

If x is an element of a semigroup S and n is a positive integer, then x^n is defined to be $xx\ldots x$, where the product has n factors. If S is a monoid with identity element 1, then we set $x^0 = 1$. If x is a unit and $n < 0$, then x^n is defined to be $(x^{-1})^{-n}$. The usual laws of exponents hold.

A *group* is a monoid in which every element is a unit. If M is a monoid, then the set of units of M is a group. Commutative groups are said to be *abelian*.

Here are some examples of monoids and groups.

Example 2.1. Let Ω be a set. The set $\mathrm{Rel}(\Omega)$ of all relations on Ω is a monoid with composition as the binary operation. The identity element of $\mathrm{Rel}(\Omega)$ is the identity function on Ω. The set $\mathrm{Fun}(\Omega)$ of functions on Ω is also a monoid. The group of units of $\mathrm{Rel}(\Omega)$ is the set $\mathrm{Sym}(\Omega)$ of permutations of Ω, which is called the *symmetric group* on Ω. If Ω is finite and $|\Omega| = n$, then

$$|\mathrm{Rel}(\Omega)| = 2^{n^2}, |\mathrm{Fun}(\Omega)| = n^n, |\mathrm{Sym}(\Omega)| = n!.$$

If $\Omega = \{1, 2, \ldots, n\}$, then $\mathrm{Sym}(\Omega)$ is also denoted $\mathrm{Sym}(n)$.

Example 2.2. Let R be a commutative ring with identity and let n be a positive integer. The set $M_n(R)$ of n-by-n matrices over R is a monoid under matrix multiplication. The identity element of $M_n(R)$ is the n-by-n identity matrix. The group of units of $M_n(R)$ is the *general linear group* $\mathrm{GL}(n, R)$ of matrices whose determinants are units in R.

Example 2.3. Let M be a monoid. For subsets A and B of M let $AB = \{ab \mid a \in A, b \in B\}$. With this operation the set of all subsets of M becomes a monoid.

Example 2.4. The set $\mathbb{Z}$ of integers with addition as the binary operation is a group. The identity element is 0 and the inverse of n is $-n$. (Here we use *additive*, not multiplicative, notation.)

Example 2.5. The set $\mathbb{Z}$ with multiplication as the binary operation is a monoid with identity element 1. Only 1 and -1 are units.

Example 2.6. The set $\mathbb{N}$ of nonnegative integers with addition as the binary operation is a monoid. Only 0 is a unit.

Example 2.7. Let X be a set. A *word* over X is a finite sequence $U = u_1, u_2, \ldots, u_m$ of elements of X. The empty sequence, with $m = 0$, will be denoted ε. The set of all words over X is denoted X^*. If $V = v_1, \ldots, v_n$ is also a word over X, then UV is defined to be the word $u_1, \ldots, u_m, v_1, \ldots, v_n$. With this multiplication, X^* is a monoid with identity element ε, which is the only unit. We identify an element x of X with the corresponding word of length 1, and we write U as $u_1 \ldots u_m$, without commas. The length m of U is denoted $|U|$. Words over some finite set X will be the objects most commonly manipulated by the algorithms discussed in this book.

If A, B, and C are in X^* and $U = ABC$, then A is a *prefix* of U, C is a *suffix* of U, and B is a *subword* of U. If $U = u_1 \ldots u_m$, then any of the words $u_i u_{i+1} \ldots u_m u_1 \ldots u_{i-1}$, $1 \le i \le m$, is called a *cyclic permutation* of U, and $U^\dagger = u_m u_{m-1} \ldots u_1$ is called the *reversal* of U. The term "string" is sometimes used as a synonym for "word". In particular, if $X = \{0, 1\}$, then elements of X^* are frequently referred to as *bit strings*.

Let M be a monoid with identity element 1. A *submonoid* of M is a subset N of M such that the following conditions hold:

(i) 1 is in N.
(ii) If x and y are in N, then xy is in N.

If in addition all elements of N are units and N contains the inverse of each of its elements, then N is a *subgroup* of M. A submonoid N is a monoid under the restriction to N of the binary operation on M. Similarly, subgroups are groups in their own right.

Example 2.8. Let Ω be a set. The set $\mathrm{Fun}(\Omega)$ is a submonoid of $\mathrm{Rel}(\Omega)$, and $\mathrm{Sym}(\Omega)$ is a subgroup of $\mathrm{Rel}(\Omega)$. Let α be an element of Ω. The set of functions f on Ω such that $f(\alpha) = \alpha$ is a submonoid of $\mathrm{Fun}(\Omega)$, and the set of permutations of Ω which fix α is a subgroup of $\mathrm{Sym}(\Omega)$.

Example 2.9. The set $\mathbb{N}$ is a submonoid of the monoids $(\mathbb{Z}, +)$ and $(\mathbb{Z}, \times)$. For any integer n, the set $n\mathbb{Z}$ of multiples of n is a subgroup of $(\mathbb{Z}, +)$.

Proposition 2.1. *Let M be a monoid. The intersection of any nonempty collection of submonoids of M is a submonoid of M. The intersection of any nonempty collection of subgroups is a subgroup.*

Proof. Exercise. □

Let Y be a subset of a monoid M. The set $\mathcal{N}$ of submonoids of M which contain Y is nonempty, since M is in $\mathcal{N}$. By Proposition 2.1, the intersection K of the elements of $\mathcal{N}$ is a submonoid. Clearly K contains Y