UNIVERSITY OF CALIFORNIA, SAN DIEGO

Parallel Algorithms for Group Word Problems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Mathematics

by

David Robinson

Committee in charge:

Professor Samuel Buss, Chair
Professor John Evans
Professor Christos Papadmitriou
Professor Ramamohan Paturi
Professor Jeffrey Remmel

1993

The dissertation of David Robinson is approved, and it is acceptable in quality and form for publication on microfilm:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

1993

TABLE OF CONTENTS

VITA

July 1, 1961	Born, San Diego, California
1983	B.A. University of California, Berkeley
1984–1985	Computer Programmer, Sperry Corp., San Diego, California
1985–1987	Software Engineer, Merdan Group, San Diego, California
1988–1989	Teaching Assistant, Department of Mathematics, University of California, San Diego
1990	M.A., University of California, San Diego
1990–1992	Teaching Assistant, Department of Mathematics, University of California, San Diego
1991	Candidate in Philosophy University of California, San Diego
1993	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

"Parallel algorithms for nilpotent groups, polycyclic groups and extensions", in preparation.

ABSTRACT OF THE DISSERTATION

Circuit Complexity of Group
Word Problems

by

David Robinson
Doctor of Philosophy in Mathematics
University of California, San Diego, 1993
Professor Samuel R. Buss, Chair

Given a set of generators, the word problem for a group is to determine whether a string of generators represents the identity element. The subject of this dissertation is boolean circuit algorithms for word problems. Of particular interest is the very low level complexity class TC^0, which consists of languages recognizable by families of $O(1)$-depth polynomial-size threshold circuits. TC^0 is a subclass of NC^1. The word problems of polycyclic groups are shown to be in TC^0. The M-group word problems are shown to be in NC^1. Uniform circuit families are explored as well. A method of describing uniform families of threshold circuits is developed. The word problems of solvable groups whose growth function is polynomially bounded is shown to be in uniform-TC^0. (The growth function $f(n)$ of a group with respect to a given set of generators is the number of elements definable by a string of length $\leq n$.) This includes the class of finitely generated nilpotent groups. The word problems of nonsolvable groups of polynomial growth are shown to be complete for NC^1. The word problems of nonabelian free groups are shown to be hard for NC^1, (these are also known as the 2-sided Dyck languages). Polynomial-size almost *log*-depth circuits are given for the word problems for free groups. Context-free groups and groups constructed by free products and extensions are discussed.

Chapter I

Introduction

The word problem for a group G is one of the three fundamental decision problems in combinatorial group theory formulated by Max Dehn in 1911. If G is generated by $\{a, b, c, \ldots\}$ then a word is defined to be a sequence of symbols from $\{a, b, c, \ldots\} \cup \{a^{-1}, b^{-1}, c^{-1}, \ldots\}$. If we view the string as a product then every word represents a group element. The word problem for G is to determine for arbitrary word w whether w represents the identity element of G. The subject of this thesis is the computational complexity of the word problems of some well known classes of groups with respect to certain models of parallel computation.

In 1986, Barrington [3] showed a class of finite group word problems to be complete (in the sense of a many-one reduction) for the parallel complexity class NC^1. NC^1 is the set of predicates that are recognizable by polynomial size, $O(\log n)$ depth bounded fanin boolean circuits. The class of groups that were shown to be complete were the finite nonsolvable groups, (the name "nonsolvable" does not in any way relate to algorithmic unsolvability). Further, Barrington showed that the word problems of solvable finite group are in a subclass (conjectured to be proper but not known) of NC^1 which is called ACC^0. In this way the property of nonsolvability can be said to characterize the class NC^1.

A separate issue in the theory of circuit complexity is that of circuit uniformity. Since an individual circuit only handles inputs of a fixed length, it is necessary

to have a family of circuits, (one for each possible input length), to recognize general languages. Loosely speaking, a family of circuits is "uniform" if a description of the i^{th} circuit of the family is easily computed. In Ruzzo [30] various versions of uniformity are compared and one very natural definition has the consequence that uniform NC^1= *Alogtime*, (time $O(\log n)$ on an alternating Turing machine). In Barrington, Immerman and Straubing [5] these definitions are extended to some subclasses of NC^1 defined in terms of unbounded fanin circuit models, including ACC^0 and a class that lies between NC^1 and ACC^0 called TC^0. TC^0 consists of predicates recognized by families of polynomial size, $O(1)$ depth unbounded fanin threshold circuits (circuits that in addition to the usual AND, OR, and NOT have $MAJORITY$ gates which output 1 if at least half their inputs are 1). In chapter 4 the issue of uniformity is taken up in depth. There, a different approach to uniformity based on an alternating Turing machine with oracles is presented which is equivalent to the previous definitions but perhaps easier to use.

The main goal of our research has been to extend the results of [3] into the domain of infinite groups. Since there exist solvable and nonsolvable infinite groups with undecidable word problem [7] [18] one cannot find an immediate analogue. Our main results are the classification of two large and important classes, the nilpotent groups and the polycyclic groups, both of which are defined in a manner that is similar in form to the solvable. The canonical examples of the nilpotent groups are the groups of upper triangular matrices with 1's along the diagonal. The polycyclic groups are exactly those solvable groups that are isomorphic to integer matrix groups.

Chapter 2 consists mainly of definitions and background on group theory and circuit algorithms.

Chapter 3 discusses the previous work on finite groups.

In chapter 4 a version of oracle Turing machine is defined and is used to define a uniform version of constant depth reducibility, denoted $\leq_{ucd}$. From this, uniform ACC^0 and TC^0 are defined. These are denoted by $uACC^0$ and uTC^0. It is proved that this definition is equivalent to having the direct connection language recognizable

in *Dlogtime*.

The emphasis of our research has been on how the process of building groups from other groups (extensions, free products, direct products, etc.) affects computational complexity. Chapter 5 looks at the problem of reducing the word problem for a group to that of a normal subgroup.

Chapter 6 examines free groups and free products. There it is shown that the word problem of a nonabelian free group is hard for *Alogtime*. Then polynomial size, almost log depth circuits are presented for those groups. Lastly, the context-free groups and free products are discussed.

The subject of chapter 7 is the nilpotent groups and their finite extensions. It is shown that the word problems of finitely generated nilpotent groups are in uTC^0. The results of chapter 5 then allow this result to be extended to classify two classes of groups characterized by their growth function $h(n)$, which is defined to be the number of elements of the group definable by a word of length $\leq n$. For groups whose growth function is bounded by a polynomial, it is shown that the word problems of the solvable are in uTC^0, and those of the nonsolvable are in *Alogtime*. The nonsolvable polynomial growth groups are then shown to be complete for *Alogtime*.

In chapter 8 the word problems of polycyclic groups are shown to be in (not necessarily uniform) TC^0. Using this and the results of chapter 5 it is then shown that the word problems of M-groups are in NC^1.

Chapter 9 discusses miscellaneous results including showing a group in none of the above mentioned classes to be in TC^0, and a recent result of Cai [9] concerning hyperbolic groups. Finally, chapter 10 discusses open problems in the area.

Chapter II

Preliminaries

II.A Languages, Machines, Circuits

A number of terms need to be defined.

Languages. If Σ is some set, then Σ^* is the set of finite length strings of elements of Σ. The empty string is denoted by 1. If $L \subseteq \Sigma^*$ then L is a **language** with alphabet Σ. The set of Σ-strings of length exactly p is denoted by Σ^p, and the set of Σ-strings whose length is a multiple of p is denoted by $(\Sigma^p)^*$. The length of the string w is denoted by $|w|$. The concatenation of strings w and x will usually be denoted by wx, but sometimes for the sake of clarity by $w \circ x$. Two classes of languages that will be of interest are the **regular languages** and the **context-free languages** as defined in Hopcroft and Ullman [22].

Turing Machines. A basic knowledge of deterministic Turing machines (DTM), nondeterministic Turing machines (NTM) and oracle Turing machines is assumed (see [22]). A language that is accepted by some Turing machine is said to be decidable.

Another computation model is the **alternating Turing machine** (ATM) [10], which generalizes the NTM. In an ATM each state is designated as either existential or universal. As in an NTM, a configuration may have more than one successor. A configuration is accepting if 1) it is existential and at least one successor configuration is accepting or 2) it is universal and all successor configurations are accepting. An

ATM is defined to accept a string if its initial configuration is accepting. For the applications considered here it will be necessary to allow the ATM (and in one case the DTM as well) to access its input via an index tape. When the ATM enters a special input state with a binary number i on its index tape, it accesses the contents of the i^{th} input symbol.

Time and space bounded versions of all three types of machines are defined in the usual way. From these, complexity classes of languages are defined. The class of languages recognized by deterministic (nondeterministic) machines whose space is bounded by order of the log of the length of the input will be designated *Dlogspace* (*Nlogspace*). The class P (resp. NP) is the class recognized by deterministic (resp. nondeterministic) machines running in polynomial time. The class of languages recognized by deterministic machines using polynomial space is called *Pspace*. The class of languages recognized by alternating Turing machines operating in time order of the log of the length of the input will be called *Alogtime*.

Circuits. This dissertation will be more concerned with computation models based on boolean circuits. In general, a **circuit** is a directed acyclic graph. Nodes with no edges directed into them are input nodes. Nodes with no edges directed out of them are output nodes. The inputs to the circuit, the outputs of the circuit, and the inputs to each node are all considered to be ordered. Each noninput node is labeled with a gate-type, which specifies a function $f : \{0,1\}^m \mapsto \{0,1\}$, where m is the number of edges directed into the node. A circuit C with n input nodes and p output nodes computes a function $f : \{0,1\}^m \mapsto \{0,1\}^p$ in the obvious way. The **size** of a circuit is the number of nodes and the **depth** is the length of the longest path from an input to an output. A **circuit family** C is a set of circuits $\{C_i\}_{i \in N}$ such that C_i has i input nodes. Such a family of circuits computes a function $f : \{0,1\}^* \mapsto \{0,1\}^*$ in the obvious way. If the range of f is $\{0,1\}$ then C is said to recognize the language $f^{-1}(1)$. We will also speak of languages over finite alphabets other than $\{0,1\}$ as being in these circuit classes. It will be understood that the alphabets of these languages are encoded as $\{0,1\}$ strings.

Various circuit complexity classes can be defined based on constraints on circuit size (the number of edges), circuit depth (the length of the longest path from an input to an output), the fanin (the number of inputs a node may have), and the types of functions that an individual node may compute. NC^k, AC^k, and TC^k are all language classes in which a language is in the class if it is recognized by a family of circuits $\{C_i\}_{i \in N}$ such that the size of C_i is bounded by a polynomial in i, and the depth of C_i is $O(\log^k i)$. (These names of language classes will also be used to denote the corresponding classes of circuit families.) The gates of the NC^k circuits are restricted to be of fanin 2, and they are allowed to compute only the boolean functions AND, OR, and NOT. The gates of AC^k circuits are restricted to these same types but they are allowed to take arbitrarily large fanin. The gates of TC^k circuits may be as in AC^k, with the addition of (unbounded fanin) $MAJORITY$ gates. Such a gate outputs 1 if and only if greater than or equal to half of its inputs are 1. ACC^k is defined to be the same as AC^k with the addition of *mod* g gates for one integer g per circuit family. (A *mod* g gate outputs 1 if and only if the number of its inputs that equal 1 is a multiple of g.) Finally, the class NC is defined as $\bigcup_{k \in N} NC^k$.

Some trivial inclusions are observed:

$$NC^0 \subsetneq AC^0 \subsetneq ACC^0 \subseteq TC^0 \subseteq NC^1 \subseteq \ldots \subseteq NC^k \subseteq AC^k \subseteq ACC^k \subseteq TC^k \subseteq \ldots \subseteq NC$$

Only the first two inclusions are known to be strict. (The class NC^0 is too restrictive to be of interest and will not be mentioned further.)

As is often observed, all of these circuit classes contain nonrecursive sets. By considering circuit families that are uniform, (meaning generally that a description of C_i is an easily computable function of i) we can relate circuit complexity to Turing machine complexity. In Ruzzo [30] various versions of uniformity are compared and one very natural definition has the consequence that uniform NC^1 is equal to *Alogtime*. In Barrington, Immerman and Straubing [5] these definitions are extended

to some classes of unbounded fanin circuits that lie below NC^1. In the uniform setting we have in addition to all the above inclusions:

$$AC^0 \subsetneq ACC^0 \subseteq TC^0 \subseteq NC^1 = Alogtime \subseteq Dlogspace \subseteq AC^1 \subseteq NC \subseteq P$$

Only the leftmost inclusion is known to be proper. In chapter 4 the issue of circuit uniformity is taken up at length.

An important concept in complexity theory is that of reducibility. If L_1 and L_2 are languages and X is a complexity class, then $L_1 \leq_X L_2$ means that there exists an X-class machine/circuit family that maps elements of L_1 to elements of L_2 and elements of $\overline{L_1}$ to elements of $\overline{L_2}$. In this case we say that L_1 is X-reducible to L_2, (in the many-one sense). If $L_1 \leq_X L_2$ and $L_2 \leq_X L_1$ then we write $L_1 \equiv_X L_2$ and say that L_1 and L_2 are X-equivalent. If for all languages L_1 in some complexity class Y $L_1 \leq_X L_2$, then L_2 is said to be **hard** for Y with respect to X-reductions. If additionally $L_2 \in Y$, then L_2 is said to be **complete** for Y with respect to X-reductions.

A particular many-one reduction we will use is the *Dlogtime* reduction, [8]. Such reductions use a DTM supplied with an index tape like that of the ATM. Suppose f maps instances of problem Q_1 to problem Q_2. Then f is a *Dlogtime* reduction if

i) for all x $|f(x)| \leq p(|x|)$ for some polynomial p.

ii) the function $\hat{f}(x,i) =$ the i^{th} character of $f(x)$ is computable by a DTM in time $O(\log |x|)$ when $i \leq p(|x|)$.

Another particularly simple many-one reduction will be called a **homomorphic reduction**. Suppose $L_1 \subseteq (\Sigma_1^p)^*, L_2 \subseteq (\Sigma_2^q)^*$. We will write $L_1 \leq_{hom} L_2$ when L_1 is reducible to L_2 in the many-one sense via a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ with the property that if $w = w_1 w_2 \ldots w_n$ with $w_i \in \Sigma_1^p$ then $f(w) = f(w_1) \circ f(w_2) \circ \ldots \circ f(w_n)$, with $|f(w_i)| = q$. In other words, f maps blocks of size p to blocks of size q.

A different form of reduction is the **constant-depth reduction**. We write $L_1 \leq_{cd} L_2$ when L_1 can be computed by a polynomial size, constant depth family of unbounded fanin circuits using NOT, AND and OR gates and L_2-gates. (An L_2-gate

takes as input a sequence $\vec{x}$ and outputs 1 if and only if $\vec{x} \in L_2$.) If there is such a circuit with the further restriction that at most one L_2-gate appears in any path, then L_1 is **constant depth truth table reducible** to L_2. This is denoted $L_1 \leq_{cd-tt} L_2$.

Lemma 2.2:

(a) There is a *Dlogtime* algorithm to determine the length of the input string in binary.

(b) if $L_1 \leq_{hom} L_2$ then $L_1 \leq_{Dlogtime} L_2$.

(c) if $L_1 \leq_{Dlogtime} L_2$ then $L_1 \leq_{AC^0} L_2$.

(d) For all $k \geq 0$ the classes ACC^k, AC^k, TC^k and NC^{k+1} are closed under constant depth reducibility.

Proof: (a) The machine M writes 1, 10, 100, ... successively on the index tape querying the input along the way until a blank symbol is found, indicating that the end of the input string has been surpassed. The last 0 is then erased and the index tape head is returned to the left end. The index tape head then begins to move to the right, changing each 0 to a 1, querying the input, and changing the 1 back to a 0 if the accessed symbol is a blank. When the right end of the index string is reached the length of the input will be written in binary on the index tape. The string can then be copied onto a worktape. Clearly the number of steps performed is $O(\log n)$.

(b) Suppose $L_1 \leq_{hom} L_2$ via the function f with p and q as in the definition. Two things need to be shown: that f is of polynomial growth rate and that $\hat{f}(x,i)$ is computable in time $O(\log |x|)$ on a DTM.

As for the growth rate: $|f(x)| = \frac{q}{p}|x|$ and thus it is polynomial.

To compute $\hat{f}(x,i)$:

1) verify that $i \leq \frac{q}{p} \log |x|$

2) compute $c = \lfloor i/q \rfloor$

3) compute $d = (i \bmod q) + 1$

4) put $c+1$ on the index tape and read input characters $x_{c+1}, x_{c+2}, \ldots, x_{c+p}$ recording the results in the finite control of the DTM

5) output the d^{th} character of $f(x_{c+1}x_{c+2}\ldots x_{c+p})$

Step 1 can be done as in part (a). Steps 2, 3 and 4 are straightforwardly computed in time linear in the length of i, thus linear in $\log |x|$.

(c) In computing the i^{th} output bit of the reduction, at most $O(\log |x|)$ input bits can be accessed by a *Dlogtime* algorithm. Thus the i^{th} output bit can be a function of only that many input bits and it can be computed by a depth 3 DNF circuit of size polynomial in $|x|$. And since there are only polynomially many output bits, it follows that the entire reduction circuit is polynomial size and depth 3.

(d) Suppose $L_1 \leq_{cd} L_2$. If $L_2 \in ACC^k$, AC^k or TC^k then a circuit of the appropriate type that recognizes L_1 can be constructed easily from L_2-circuits and the constant depth reduction circuit by simply substituting an L_2-circuit in place of every L_2-gate. If $L_2 \in NC^k$ (for $k \geq 1$) then first transform the reduction circuit by replacing all unbounded fanin ANDs and ORs with balanced trees of fanin 2 ANDs and ORs. This may increase the depth from constant to $O(\log n)$, where n is the length of the input. Now as in the previous case replace the L_2-gates with L_2-circuits and we are left with a depth $O(\log n) + O(\log^k n) = O(\log^k n)$ circuit. In all cases it is straightforward to see that the circuit size remains polynomial. □

Previous work putting common functions of arithmetic into the circuit complexity hierarchy is summarized in the next set of theorems. The functions *Addition*, *Multiplication* and *Division* take two binary inputs and return one binary output. The functions *Iterated-Addition* and *Iterated-Multiplication* take n n–bit binary inputs and return respectively the $(n + \log n)$–bit sum and n^2–bit product. The function x mod m takes two binary inputs and returns the remainder of $x \div m$ in binary.

Theorem 2.3 (see Chandra, Stockmeyer, Vishkin 1984 [11]):

(a) $Multiplication \equiv_{cd-tt} Majority$

(b) $Iterated\text{-}Addition \equiv_{cd-tt} Majority$

(c) *Addition* $\in AC^0$

Corollary 2.4:

(a) *Multiplication* $\in TC^0$

(b) *Iterated-Addition* $\in TC^0$

Proof: By lemma 2.2(d).□

Theorem 2.5 (Beame, Cook, Hoover 1986 [6]):

(a) Computing x mod m is in NC^1.

(b) *Iterated-Product* $\in NC^1$

(c) *Division* $\in NC^1$

In theorem 2.5 the given circuits are P-uniform. It is not known whether these functions are in *Alogtime*.

II.B Basic Group Theory

A **group** G is a set with an associative operation $\circ$ such that

a) there exists an element 1 (called the **identity element**) such that $\forall g \in G\ 1 \circ g = g \circ 1 = g$

b) every element has an inverse: $\forall g \exists h\ g \circ h = h \circ g = 1$

(Usually the circle will be omitted.) The inverse of g is designated by g^{-1}. A **subgroup** of G is a subset that is also a group under the same operation. H is a **normal subgroup** of G, designated $H \lhd G$, if for all $g \in G$ and all $h \in H$, $g^{-1}hg \in H$. The kernel of a group homomorphism is always a normal subgroup. Isomorphism between groups G_1 and G_2 is designated $G_1 \cong G_2$.

A subset $S = \{g_1, g_2, \ldots\}$ **generates** G if every element of G can be expressed as a product of elements of S and their inverses. We write $G = \langle S \rangle$. If G is generated by a single element then G is called **cyclic**. If for all $g, h \in G, gh = hg$ then G is called

abelian. The **cartesian product** of G and H, designated $G \times H$, is a group whose elements are $\{(g,h) : g \in G, h \in H\}$ with the operation $(g_1,h_1)\circ(g_2,h_2) = (g_1 \circ g_2, h_1 \circ h_2)$. This can be generalized to an arbitrary number of groups in the obvious way. We can now state a standard theorem, the proof of which can be found in [29]

The Fundamental Theorem of Finitely Generated Abelian Groups: Every finitely generated abelian group G is the Cartesian product of cyclic groups.

For $g \in G$, $gH = \{gh : h \in H\}$ is the **left coset** of H determined by g. The set $\{gH : g \in G\}$ partitions G and if $H \lhd G$ it forms a group under the operation $(g_1H) \circ (g_2H) = (g_1 \circ g_2)H$ called the **quotient group** of G mod H. It is designated G/H. Also, if $K \cong G/H$ we say that G is an **extension** of H by K. The element g is said to be a **coset representative** of gH, and a subset of G that has exactly one representative for each coset is called a set of coset representatives. The cardinality of G/H is called the **index** of H in G.

The rest of the lemmas in this section are standard.

Lemma 2.6: Let G be a group with H a subgroup of finite index. Then H has a subgroup N such that $N \lhd G$ and N is of finite index in G.

Proof: Let $A = \{k_1H, k_2H, \ldots, k_nH\}$ be a complete set of left coset representatives for H. Then each $g \in G$ induces a permutation on A by left multiplication: for each i $g(k_iH) = k_jH$ for some j. These permutations form a group σ and the map $f : G \to \sigma$ is a homomorphism of G onto a finite group. Thus if the kernel of f is K then $K \lhd G$ and K is of finite index. Lastly, to see that $K \subseteq H$, suppose $k \notin H$. Then $k(1H) \neq H$ and so the permutation induced by k is not the identity and k is not in the kernel. □

For $g,h \in G$, the **commutator** of g and h is $[g,h] = g^{-1}h^{-1}gh$. For H,K subgroups of G, $[H,K]$ is the subgroup generated by $\{[h,k] : h \in H, k \in K\}$. It

is basic that $[G,G]$ is necessarily a normal subgroup of G, (see lemma 2.7(a)); it is called the **commutator subgroup** of G.

Lemma 2.7: (a) If $H \lhd G$ then $[G,H] \lhd G$.

(b) $G/[G,G]$ is abelian.

(c) If $H \lhd G$ and R is a set of representatives for G/H then every $g \in G$ has a unique representation as rh for some $r \in R$ and $h \in H$.

(d) If $H \lhd G$ then for any $g \in G$ and $h_1 \in H$ there is an $h_2 \in H$ such that $gh_1 = h_2 g$.

Proof: (a) Define an arbitrary $k \in [G,H]$ by $k = d_1 d_2 \dots d_n$ with $d_i = [g_i, h_i]$. It needs to be shown that for an arbitrary element $p \in G$ we have $p^{-1}kp = (p^{-1}d_1p)(p^{-1}d_2p)\dots(p^{-1}d_np)$ it suffices to show that $p^{-1}d_ip \in [G,H]$.

$$\begin{aligned} p^{-1}d_ip &= p^{-1}g_i^{-1}h_i^{-1}g_ih_ip \\ &= (p^{-1}g_i^{-1}p)(p^{-1}h_i^{-1}p)(p^{-1}g_ip)(p^{-1}h_ip) \end{aligned}$$

resulting in a commutator of the correct form.

(b)

$$\begin{aligned} a[G,G]\cdot b[G,G] &= ab[G,G] \\ &= ab(b^{-1}a^{-1}ba)[G,G] \\ &= ba[G,G] \\ &= b[G,G]\cdot a[G,G] \end{aligned}$$

(c) Since the set $\{rH : r \in R\}$ partitions G it is immediate that $g = rh$ for some $r \in R,\ h \in H$. To show uniqueness, suppose $r_1h_1 = r_2h_2$. Then

$$\begin{aligned} r_1h_1H &= r_2h_2H \\ r_1H &= r_2H \\ r_1 &= r_2 \quad \text{by definition of set of representatives} \\ \text{and} \quad h_1 &= h_2 \quad \text{by cancellation} \end{aligned}$$

(d) Since H is normal $gh_1g^{-1} = h_2$ for some $h_2 \in H$, and then $gh_1 = h_2g$. □

Two important chains of subgroups are defined: the derived series and the lower central series.

The n^{th} **derived group** $G^{(n)}$ is defined recursively by

$$G^{(0)} = G$$

$$G^{(n+1)} = [G^{(n)}, G^{(n)}].$$

If for some n, $G^{(n)} = \{1\}$ then G is a **solvable group**. If G is not solvable then it is called **nonsolvable**. Examples of solvable groups include the symmetric groups S_n for $n \leq 4$. For $n > 4$, S_n is nonsolvable.

The **lower central series** is defined recursively by

$$G_1 = G$$

$$G_{n+1} = [G_n, G]$$

If for some n, $G_n = \{1\}$ then G is called **nilpotent**. Examples include all groups of upper triangular matrices with 1's along the diagonal. Since $G^{(n)} \subseteq G_{n+1}$ for all n, it is straightforward that if G is nilpotent then it is also solvable. (Note the somewhat awkward circumstance that G is the zeroth term of the derived series but the first term of the lower central series.)

A sequence of subgroups $G = H_1 \rhd H_2 \rhd \ldots \rhd H_n = \{1\}$ is called a **subnormal series**. By lemma 2.7(a) both of the above are examples. (Note: the H_i need not be normal in G itself.) If G has a series such that for all i H_i/H_{i+1} is cyclic then G is called **polycyclic**. The polycyclic groups lie intermediate between the nilpotent and the solvable groups. In summary, we have the following relationships among classes of finitely generated groups. (See the chapters devoted to the particular classes for proofs.)

$$\text{Abelian} \subsetneq \text{Nilpotent} \subsetneq \text{Polycyclic} \subsetneq \text{Solvable}$$

One way to specify a group is by a presentation in terms of generators and relators. A **relator** is a string of generators that is equal to 1 in the group. The **presentation** $G = \langle g_1, g_2, \ldots ; R_1, R_2, \ldots \rangle$ defines a group whose elements are equivalence classes of strings in the alphabet $\{g_1, g_2, \ldots\} \cup \{g_1^{-1}, g_2^{-1}, \ldots\}$. Two such strings are equivalent if and only if one can be transformed into another by repeated insertion and

deletion of the relators $R_1, R_2, \ldots$ and the **trivial relators** $g_1 g_1^{-1}$, $g_1^{-1} g_1$, $g_2 g_2^{-1}, \ldots$. If G is defined with a finite number of generators (relators) then it is called **finitely generated (related).** If it is both finitely generated and finitely related then G is **finitely presented.**

If G has a presentation with no relators then G is a **free group.** The number of generators in the presentation does not depend on the choice of (free) generators and is called the rank of the group, (see Rotman [29], p. 240). Some properties of free groups are discussed in the following lemma. The symbol $\mathbb{Z}$ indicates the group of integers under the operation of addition.

Lemma 2.8: (a) If G is free of rank 1 then $G \cong \mathbb{Z}$.

(b) If G is free of rank ≥ 2 then G is nonsolvable.

Proof: (a) $\langle a; \rangle = \{a^k : k \in \mathbb{Z}\}$

(b) It will be shown that for all i $G^{(i)}$ contains elements with reduced forms av_1b, $a^{-1}v_2b$, av_3b^{-1} and $a^{-1}v_4b^{-1}$. For $i = 0$ the strings ab, $a^{-1}b$, ab^{-1} and $a^{-1}b^{-1}$ suffice. Now suppose that strings of the four forms above are in $G^{(i)}$. Then

$$1 \neq \left\{ \begin{array}{l} (av_3b^{-1})(b^{-1}v_2^{-1}a)(bv_3^{-1}a^{-1})(a^{-1}v_2b) \\ (a^{-1}v_4b^{-1})(b^{-1}v_1^{-1}a^{-1})(bv_4^{-1}a)(av_1b) \\ (av_1b)(bv_2^{-1}a)(b^{-1}v_1^{-1}a^{-1})(a^{-1}v_4b^{-1}) \\ (a^{-1}v_2b)(bv_3^{-1}a^{-1})(b^{-1}v_2^{-1}a)(av_3b^{-1}) \end{array} \right\} \in G^{(i+1)}$$

□

Earlier it was shown that groups can be constructed from other groups via the cartesian product. Another source of new groups is the **free product.** If $G = \langle x_1, x_2, \ldots; R_1, R_2, \ldots \rangle$, $H = \langle y_1, y_2, \ldots; S_1, S_2, \ldots \rangle$ and the two presentations have no generators in common then the free product $G \otimes H$ has presentation

$$\langle x_1, x_2, \ldots, y_1, y_2, \ldots; R_1, R_2, \ldots, S_1, S_2, \ldots \rangle.$$

An example is the free group $\langle x, y; \rangle = \langle x; \rangle \otimes \langle y; \rangle$.

If G is generated by a finite set S then we define the **word problem** for G (with respect to S) as the subset of strings of elements of S and their inverses that are equal to 1 in G. With regard to group presentations, it was shown in the 1950's that there exist finitely presented groups with undecidable word problems [7]. More recently it has been shown that there are finitely presented solvable groups with undecidable word problem [18].

II.C Circuits for Word Problems

The subject of this thesis is the computational complexity of the word problems of finitely generated infinite groups, particularly with regard to circuit models of computation. The point of departure is Barrington's 1986 paper on finite group word problems [3], which is discussed in the next chapter.

One way that circuit models differ from machine models of computation is that the former technically can only recognize languages with alphabet $\{0,1\}$, whereas the latter can work with arbitrary finite alphabets. Thus, when using circuits to recognize word problems of groups with more than one generator it is necessary to encode each generator as a string of bits. It will be assumed that in a given encoding each generator's code will be of the same length. Another fact to consider is that different sets of group elements can be chosen to be the generators. We will always assume that the identity element is represented by one of the generators. When the choices of generators or encodings are otherwise varied, the actual $\{0,1\}$-language of the word problem changes, making references to "the word problem of group G" ambiguous. The following lemma addresses this issue.

Lemma 2.9: Let G be a finitely generated group.

(a) if L_1 and L_2 are two $\{0,1\}$-languages representing the word problem for G using the same set of generators but different encodings of those generators then $L_1 \equiv_{hom} L_2$.

(b) if L_1 and L_2 are two representations of the word problem for G using different sets of generators then $L_1 \equiv_{hom} L_2$.

Proof: Part (a) is immediate. Given the assumption that the identity element is a generator in both languages, part (b) is also straightforward. □

Thus, the choices of generators and encodings are irrelevant since all of the complexity classes we are interested in are closed under homomorphic reductions.

Some final notational conventions will be needed. If g is a letter and v and $w = w_1 w_2 \ldots w_{n-1} w_n$ are strings, then:

1) g^{-1} is to be regarded as a single letter (distinct from g),

2) g^{+1} is to be regarded as a single letter identical to g,

3) $(g^{-1})^{-1}$ is identical to g,

4) w^{-1} is the string $w_n^{-1} w_{n-1}^{-1} \ldots w_2^{-1} w_1^{-1}$,

5) $v = w$ means that v and w represent the same group element,

6) $v \equiv w$ means that v and w are identical strings.

7) The **net occurence** of g in w is defined to be the number of occurences of g in w minus the number of occurences of g^{-1} in w.

Chapter III

Finite Groups

This chapter addresses the computational complexity of finite group word problems. We begin with an unsurprising observation:

Theorem 3.1: If G is finite then $G \in$ *Alogtime*.

Proof: Assume the input alphabet consists of one symbol for each group element. The following ATM algorithm recognizes the word problem for G.

(A) *START UP*

A1 Put n in binary on a worktape as in lemma 2.2.

A2 Put $0^{p-1}1$ on the index tape where $p =$ the length of n. Put the index tape head on the first blank to the left of the string.

A3 Set result r to 1.

(B) *VERIFY* $x_i x_{i+1} \ldots x_q = r$ *where*

$i =$ the number written on the index tape

$q =$ the number that would be on the index tape if all 0's to the right of the head were changed to 1's

B1 Move index tape head one square to the right

B2 If not yet reached a blank on the index tape

then

B2.1 Existentially guess a group element h

B2.2 Universally

B2.2.1 leave the index square 0, set $r := h$, go to (B)

B2.2.2 change the index square to 1, set $r := h^{-1}r$, go to (B)

else

B2.3 Use the contents of the index tape to access input $= x_i$

Case 1: if $x_i = r$ then accept.

Case 2: if $x_i =$ blank and $r = 1$ then accept.

Case 3: otherwise, reject.

The algorithm is a straightforward divide and conquer strategy. A result for the product of a string is guessed and then the string is split in half. A guess is made for the product of each half such that the two guesses together equal the guess for the full string. The two halves are then treated recursively. Clearly each stage takes time $O(1)$ and there are $\log n$ stages, giving a total time of $O(\log n)$.□

The next theorem shows the previous to be as good as can be hoped for.

Theorem 3.2: (Barrington 1986): Let G be a finite nonsolvable group. Then:

1) The word problem for G is complete for NC^1 with respect to AC^0 reductions.

2) The word problem for G is complete for *Alogtime* with respect to *Dlogtime* reductions.

Proof: Barrington's proof of this involves the use of bounded-width branching programs. Rather than develop the necessary background to do that, we will recreate the part of the proof that pertains only to circuits and word problems. Only the nonuniform version (part 1) will be addressed here; it is straightforward to modify the proof for the uniform setting.

We can assume without loss of generality that G is its own commutator subgroup. We can do this because we know that for any nonsolvable finite group G, there

is an n such that $G^{(n)} = G^{(n+1)} \neq \{1\}$. It easily follows that if $G^{(n)}$ is hard for NC^1 then so is G. Let G be nonsolvable and $|G| = m$. Most of the proof is contained in the following lemma.

Lemma: Let L be a language and L_n be the members of L of length exactly n. Suppose that L_n is recognized by a bounded fanin circuit C of depth d consisting only of OR and NOT gates. Let σ be an arbitrary nonidentity element of G. Then there exists a depth 1 unbounded fanin circuit that takes n inputs and maps $\vec{x}$ to $w(\vec{x})$ such that $w(\vec{x})$ is a string of generators of G and the group element represented by $w(\vec{x})$ is σ if $\vec{x} \in L_n$ and is the identity (denoted e here) if $\vec{x} \notin L_n$, and such that $|w(\vec{x})| \leq (4g)^d$.

Proof of lemma: by induction on d.

Base case: if $d = 0$ then L_n is either all of the length n strings, none of them, all of those with $x_i = 1$ for some i, or all of those with $x_i = 0$ for some i. In all these cases w can easily be computed using a depth 1 circuit with $|w(\vec{x})| = 1$.

Induction step: Suppose the above holds for all languages whose restriction to length n is recognized by depth $d-1$ circuits.

Case 1: the output gate of C is an OR. Then let C_1 and C_2 denote the two input circuits to C. They are then circuits with n inputs of depth $d-1$, so the induction hypothesis holds.

Since G is its own commutator subgroup, σ is equal to a product of commutators $d_1 d_2 \ldots d_k$, with $d_i = a_i^{-1} b_i^{-1} a_i b_i$. By the induction hypothesis (and since σ is arbitrary) there exist depth 1 circuits $A^1_{a_i^{+1}}$, $A^1_{a_i^{-1}}$, $A^2_{b_i^{+1}}$, $A^2_{b_i^{-1}}$ such that on input $\vec{x}$, A^j_γ outputs a string equivalent to e if C_j accepts $\vec{x}$ and γ if C_j rejects $\vec{x}$. Furthermore, the strings these reductions create are bounded in length by $(4g)^{d-1}$.

We create the reduction circuit for L^n by laying the above circuits side by side in the following order in a sequence of commutators mirroring $d_1 d_2 \ldots d_k$:

$$(A^1_{a_1^{-1}} A^2_{b_1^{-1}} A^1_{a_1^{+1}} A^2_{b_1^{+1}})(A^1_{a_2^{-1}} A^2_{b_2^{-1}} A^1_{a_2^{+1}} A^2_{b_2^{+1}}) \ldots (A^1_{a_k^{-1}} A^2_{b_k^{-1}} A^1_{a_k^{+1}} A^2_{b_k^{+1}})$$

Case 1a: $\vec{x} \in L_n$ (i.e., at least one of C_1 and C_2 returns 1). Then each of the commutators will be of the form $eb^{-1}eb$, $a^{-1}eae$, or $eeee$. All of these equal the

identity element and thus the resulting output string will equal the identity, reflecting acceptance of $\vec{x}$ as desired.

Case 1b: $\vec{x} \notin L_n$. Then the resulting string will equal $d_1 d_2 \ldots d_k = \sigma$ as desired.

As to the size of the resulting circuit, each subcircuit A_γ^j is bounded by $(4g)^{d-1}$. It is easy to see that k is bounded by g, so we have $\leq 4g$ subcircuits and thus total size $\leq (4g)^d$.

Case 2: The output gate of C is a NOT. We must then have a circuit of depth $d-1$ recognizing $\overline{L_n}$. By induction hypothesis we have an appropriately sized reduction circuit that outputs e if $\vec{x} \in \overline{L_n}$ and σ^{-1} if $\vec{x} \in L_n$. If we simply take that circuit and append the constant output σ to the end we have a reduction circuit that outputs $e \cdot \sigma = \sigma$ if $\vec{x} \in L_n$ and $\sigma^{-1} \cdot \sigma = e$ if $\vec{x} \notin L_n$. Since the size of the circuit only increases by 1 it is clear that the size bound is maintained. *q.e.d. lemma*

Proof of theorem: If L is in NC^1 then it is easy to see from DeMorgan's laws that L is recognized by a family of bounded fanin AND-NOT circuits of depth $O(\log n)$. Choose an arbitrary nonidentity $\sigma \in G$. The reduction in the lemma produces a circuit of depth 1 and size bounded by $(4g)^{depth} = (4g)^{O(\log n)} = (4gn)^{O(1)} =$ polynomial in n. Thus we have an AC^0 circuit that maps inputs $\vec{x}$ to the identity of G if $\vec{x} \in L$ and to something else (σ) if $\vec{x} \notin L$.

That the word problem is actually in NC^1 follows from the finiteness of G as observed in the previous theorem. □

In addressing the finite solvable groups, the following well known lemma will be needed.

Lemma 3.3: If G is finite and solvable then G is polycyclic.

Proof: Use induction on the size of G. Let $H = G^{(1)}$ and let $R = \{r_1, \ldots, r_k\}$ be a set of coset representatives for G/H. Since G is solvable, H is a proper subset of G, (or else for all n $G^{(n)} = G$). Since G/H is abelian (see lemma 2.7(b)) it is by

the fundamental theorem a cartesian product of cyclic groups. Let $\{s_1H, \ldots, s_mH\}$ be a minimal set of generators of G/H.

Claim: the subgroup K generated by $\{s_2, \ldots, s_m\} \cup H$ is normal in G. To see this let $g = s_1^{p_1}s_2^{p_2}\ldots s_m^{p_m}h_g$ and $k = s_2^{q_2}\ldots s_m^{q_m}h_k$ be arbitrary elements of G and K respectively, with $h_g, h_k \in H$. Then $g^{-1} = s_1^{-p_1}s_2^{-p_2}\ldots s_m^{-p_m}h_{g^{-1}}$ for some $h_{g^{-1}} \in H$. It must be shown that $gkg^{-1} \in K$. Since G/H is abelian

$$\begin{aligned} gkg^{-1} &= s_1^{p_1-p_1}x_2^{p_2+q_2-p_2}\ldots s_m^{p_m+q_m-p_m}h \quad \text{for some } h \in H \\ &= s_2^{q_2}\ldots s_m^{q_m}h \in K \end{aligned}$$

Now since clearly s_1K generates G/K we have $K \lhd G$ with G/K cyclic. Since K is a proper subgroup of G it follows by induction that G is polycyclic. □

Theorem 3.4 (Barrington 1986): If G is a finite solvable group then the word problem for G is in ACC^0. In particular, if $|G| = m$ then the word problem for G is decidable by constant depth circuits with gates of type AND, OR, NOT, and MOD-m.

Proof: Following the lemma, let $G = H_0 \rhd H_1 \rhd \ldots \rhd H_k = \{1\}$, with H_i/H_{i+1} cyclic. The proof will be by induction on k. If $k = 0$ then G is the trivial group and the theorem obviously holds. Now suppose $k > 0$, $|G| = m$, aH generates G/H_1, and $\{g_1, \ldots, g_p\}$ is the set of generators given for G. For each g_i there is an $h_i \in H_1$ and an integer z_i such that $g_i = a^{z_i}h_i$. (We have $0 \leq z < m$ since $a^m = 1$.)

If the input is $x_1x_2\ldots x_n = g_{i_1}g_{i_2}\ldots g_{i_n}$ then we can convert this to

$$(a^{z_1}h_{i_1}a^{-z_1})(a^{z_1+z_2}h_{i_2}a^{-(z_1+z_2)})\ldots(a^{z_1+z_2+\ldots+z_n}h_{i_n}a^{-(z_1+z_2+\ldots+z_n)})a^{z_1+z_2+\ldots+z_n}$$

and since $a^m = 1$ we can replace all of the exponent sums by their values mod m. The final block after the last conjugate is then equal to a^p for some $0 \leq p < m$. This number p can be tested to see if it is a multiple of the order of a in constant depth. If it is not, then the input is rejected; if it is, then we continue:

Each of the products in parentheses is a conjugate of an element of H and thus is in H. So after replacing the exponents by their values mod m we have just a finite

number of possible things in the parentheses, each of which can be translated into a string in the generators of H_1 in constant depth. Thus we have reduced the word problem for G to the word problem for H_1 using constant depth MOD-m circuits. The final circuit is obtained by iterating this process k times. □

This procedure of creating conjugates in the proof of theorem 2 is commonplace and will appear again. This process of moving coset representatives step by step to one end will be called "conjugate collection." This sort of algorithm is used by P. Hall [16] in demonstrating that the word problems of finitely generated nilpotent groups are decidable.

Lastly, the following theorem relates finite group word problems to classical language theory. The proof here differs from the original, and from that appearing in Muller and Schupp [28].

Theorem 3.5: (Anisimov [1]) The word problem for G is a regular language if and only if G is finite.

Proof: Clearly if G is finite one can create a DFA to recognize its word problem. For each group element g there will be one state s_g. The start state and only accepting state will be s_1. The transition function will be such that input symbol h will change that state for s_g to s_{goh}. The resulting machine will be in state s_g if and only if the portion of the input string read up to that point is equivalent to g in the group G.

Now suppose G is an infinite group and we will see that the word problem is not regular. Suppose for the sake of contradiction that there is a DFA recognizing the word problem. Since G is infinite there must be two strings w_1 and w_2 representing different group elements that lead to the same state s. Since $w_1 w_1^{-1} = 1$ it follows that the string w_1^{-1} must send the machine form s to an accepting state. This implies that $w_2 w_1^{-1}$ will also be accepted. However, since $w_2 \neq w_1$ that string should not be accepted, giving a contradiction. □

Chapter IV

Uniform Circuits

This chapter addresses some technical issues regarding circuit uniformity that will be needed in upcoming chapters. The goal is to develop a way of describing uniform circuit algorithms that is easy to use and compatible with the Barrington, Immerman and Straubing [5] approach.

IV.A Oracle Turing Machines

Alternating Turing machines are used to create uniform versions of AC^0 (in the form of the Logtime hierarchy LH) and NC^1 (*Alogtime*). In order to extend this to other circuit classes such as TC^0 and ACC^0 we introduce a type of oracle ATM. The idea is for it to be a uniform version of constant depth reducibility.

Definition: A k-oracle ATM consists of a finite control and two types of tapes and their associated finite alphabets. The two types of tapes are:

1) 1 input tape

2) j work tapes

All tapes are two–way infinite. Each work tape has an associated tape head which scans one tape square. Each work tape will be associated with a variable, which will have the worktape's contents as its value. To make possible sublinear runtimes the

input tape is accessed via indexing. Any work tape can be used as an index tape. If work tape w has on it the binary number z then we consider the z^{th} tape square of the input tape as being scanned. Thus the finite control is considered to be scanning up to j input squares and j worktape squares at any given time. Of course, work tapes may be used for other purposes than indexing. Each tape has an alphabet which may or may not intersect with other tapes' alphabets. Each alphabet includes the blank symbol.

The finite control stores the state. There are a finite number of states. Each state is one of $k + 5$ different types:

1) existential

2) universal

3) negation

4) deterministic

5) k oracle query state types labeled $L_1, L_2, \ldots, L_k$.

6) halting

Oracle query state type L_p is intended to allow access to oracle language L_p. One state is designated as the initial state.

If the current state type is existential, universal or negation then in one move the machine may depending on the current state and all the scanned worktape symbols:

1) change state

2) print a symbol on each tape square scanned except the input

3) move each tape head right or left

If the current state type is deterministic then the machine can do all of the above and additionally may base its move on the scanned input symbols as well. A configuration reached in one move is called a successor to the current configuration. The machine may indicate several successors to one configuration.

When an oracle state is entered the machine must designate one of the work

tapes as the *first auxiliary*. If the current state type is L_p with first auxiliary w and the number z is written in binary on w then the current configuration has z successors. Each successor is identical to the previous except that they all may have the same new state and on some prearranged tape (called the *second auxiliary*) each has a different binary number in the range 1 to z written on it. The shorthand "Invoke $L_p(i,j)$" will mean move into an L_p-type state with i and j as the first and second auxiliary tapes respectively.

For the sake of convenience, existential and universal oracles are always assumed to be present. (E.g., the AND of k items can be computed by writing the number k on a worktape and invoking the *Universal* oracle.)

There are exactly two halting states called "Accept" and "Reject" (also denoted by 1 and 0 resp.) When one of these states is entered the computation halts.

The language accepted is a set of strings in the input alphabet. An input string is accepted if the initial configuration (the state being the initial state, the input string on the input tape and all other tapes blank) is accepting. In general, whether a configuration is accepting or rejecting is defined recursively. A configuration is accepting if:

1) the state is Accept;

2) the state type is existential and at least one of its successors is accepting;

3) the state type is universal and all of its successors are accepting;

4) the state type is negation and its only successor is rejecting;

5) the state type is deterministic and its only successor is accepting;

6) the state type is L_p and the string $s_1 s_2 \ldots s_m \in L_p$ where s_q is the acceptance status (0 or 1) of the q^{th} of m successor states.

A configuration is rejecting if this recursive procedure does not show it to be accepting. (This necessarily includes the case when the state is Reject.)

That concludes the definition of the oracle ATM to be used here. The first goal of defining the machine like this is that the uniform classes arrived at be equivalent

to those defined in [5]. If we designate the uniform version of class X by uX then this will give us then:

$$LH \subsetneq uACC^0 \subseteq uTC^0 \subseteq Alogtime$$

And of course the uniform classes will have the desired relationship to their corresponding nonuniform class (i.e. containment).

The second goal is that it be easy to "program." This is the reason for various unusual features of the model. For example, the reason more than one oracle language is allowed is to let several equivalent languages serve as oracles.

Another peculiarity is the way the oracle steps work. Once an oracle state is entered an unbounded number of successor configurations appear at once. This is done to avoid having to program–in a step by step creation of each of the successors needed each with their number. Also, this orders the inputs to the oracle, which allows the use of nonsymmetric languages. Another feature that is different from some models is that the input square corresponding to the number on an index tape is automatically assumed to be "visible,"— no special query state need be entered, (although the state must be of deterministic type). Also there is no preassigned "query tape"; this eliminates the need of copying strings computed on worktapes onto index tapes.

Now to define the uniform classes. In the following definition, with regard to counting time used, one oracle invocation counts as one time step. Also, we count as an alternation any time the state type changes or any time an oracle invocation is made (even if the previous state was an oralcle invocation of the same type).

Definition: $L_0 \leq_{ucd} \{L_1, L_2, \ldots, L_k\}$ if L_0 is recognizable by an oracle ATM supplied with $L_1, L_2, \ldots, L_k$ oracle states that uses time $O(\log n)$ and does $O(1)$ alternations. (The label *ucd* stands for "uniform constant depth.")

Definition: a) $ucd(L_1, L_2, \ldots, L_k) = \{L : L \leq_{ucd} \{L_1, L_2, \ldots, L_k\}\}$;

b) $uACC^0 = \{L : L \in ucd(Mod\text{-}p) \text{ for some } p\}$;

c) $uTC^0 = ucd(Majority)$.

First we prove two general properties of this kind of reducibility.

Lemma 4.1: If $L_0 \leq_{ucd} \{L_1, L_2, \ldots, L_k\}$ and $L_k \leq_{ucd} L_1$ then $L_0 \leq_{ucd} \{L_1, L_2, \ldots, L_{k-1}\}$.

Proof: Let us assume that we have *ucd* programs P_0 computing L_0 from $\{L_1, L_2, \ldots, L_k\}$, and P_1 computing L_k from L_1. We need to create a program P_2 that computes L_0 from $\{L_1, L_2, \ldots, L_{k-1}\}$. The basic idea is to simulate P_0 by substituting P_1 for each invocation of the L_k oracle in P_0. Some difficulty arises when P_1 makes input queries since these must now receive what were inputs to the L_k oracle rather than the actual inputs to the program. The most straightforward way of getting an input — existentially guessing it and universally verifying correctness of the guess and subsequent acceptance assuming the correctness of the guess — is unacceptable because it introduces too many alternations, (since there may be $O(\log n)$ many input queries on a path). So instead we simulate each deterministic segment as a whole (recall that the inputs can only be queried in deterministic states) by existentially guessing the entire sequence of bits received from input queries in the segment and then universally verifying that 1) the guess was correct, and 2) given that the guess was correct the original machine (P_0) goes on to accept.

Assume that the L_k oracle was invoked in P_0 with q as the value on the first auxiliary tape, that P_1 works in time $c_1 \log n + c_2$ for inputs of length n, and that P_1 has z worktapes. Then the deterministic sequence $\vec{D}$ in P_1 is replaced by:

1) Compute $r := 2^{zc_2} q^{zc_1}$; i.e., $r = 2^{z(c_1 \log q + c_2)}$ (the length of r is an upper bound on the number of input queries on a path)
2) $Existential(r, i)$; i then contains a sequence of $\leq z(c_1 \log q + c_2)$ bits
3) Universally

3a) Simulate $\vec{D}$ by taking the bits of i in place of the input query responses

3b) *Universal*(r, j) ; verify guesses are correct –(note that r is actually
; much bigger than necessary)

3b.1) Simulate $\vec{D}$ using bits from i for $j-1$ steps

3b.2) If the next (i.e., the j^{th}) query looks at the e^{th} input bit
then

Case 1: if the j^{th} bit from i is a 1
then simulate the e^{th} input to the L_k oracle in P_0

Case 2: if the j^{th} bit from i is a **0**
then negate and then simulate as in case 1

Line 1 computes a number large enough that some i guessed in line 2 will contain the sequence of input queries as a prefix. In line 3a the original computation of P_1 is continued, with the deterministic segment $\vec{D}$ simulated using the guessed bits in place of the input queries. In line 3b the correctness of each guessed bit is verified by checking that for all j if the first $j-1$ guesses were correct then the j^{th} guess is correct. It is a simple matter to verify that this program segment has $O(1)$ alternations and uses time $O(\log n)$.

The new program P_2 is still *ucd*. Since each oracle invocation counts as an alternation any path in P_0 has only $O(1)$ invocations of L_k. As the modified P_1 program inserted into P_0 has $O(1)$ alternations and uses time $O(\log n)$, so does the thus created program P_2. So P_2 is a *ucd* program. □

Lemma 4.2: If $L_1 \leq_{ucd} L_2$ and $L_2 \leq_{ucd} L_3$ then $L_1 \leq_{ucd} L_3$.

Proof: If $L_1 \leq_{ucd} L_2$ then trivially $L_1 \leq_{ucd} \{L_2, L_3\}$. Theorem 4.1 then tells us that $L_2 \leq_{ucd} L_3$ implies that $L_1 \leq_{ucd} L_3$. □

Lemma 4.3: For any language L, $ucd(L)$ is closed under *Dlogtime* reductions (and therefore under $\leq_{hom}$ as well).

Proof: Suppose $L_1 \leq_{Dlogtime} L_2$, and $L_2 \in ucd(L)$, which is to say $L_2 \leq_{ucd} L$. A

Dlogtime reduction translates straightforwardly into a $ucd(L_2)$ program: the polynomial p of the definition (see Chapter 2) is computed deterministically, an invocation $L_2(p,i)$ is made, and $\hat{f}(x,i)$ is computed giving the i^{th} input to the oracle. Thus $L_1 \leq_{ucd} L_2$. By transitivity of $\leq_{ucd}$ (lemma 4.2) $L_1 \leq_{ucd} L$, and thus $L_1 \in ucd(L)$. □

IV.B Functions in Uniform TC^0

This section contains some examples of *ucd* programs. To begin, let *Exactly-Half* be the set of binary strings consisting of an equal number of 1's and 0's.

Example 1: $Exactly\text{-}Half \in uTC^0$.

Proof: We describe an algorithm with *Majority* as the single oracle.

1) Get $n = |x|$ on a worktape using the *Dlogtime* algorithm of lemma 2.2.
2) Universally
   2a) Invoke $Majority(n,i)$
      If $x_i = 1$ then Accept else Reject
   2b) Invoke $Majority(n,i)$
      If $x_i = 0$ then Accept else Reject

The program checks that the majority of the inputs are 0's and that the majority of the inputs are 1's. It is clear that only a constant number of alternations occur and that the time used is $O(\log n)$. □

Example 2: a) $\mathbb{Z}_k \in uACC^0$

b) $\mathbb{Z} \in uTC^0$

Proof: a) Suppose that $\mathbb{Z}_k$ is generated by element g and that the input alpabet is $\{g, g^{-1}\}$. The following program which uses *Mod-k* as its oracle recognizes $\mathbb{Z}_k$. The plan is to replace each occurence of g with a string of one 1 and $k-1$ 0's and each

occurence of g^{-1} by a string of one 0 and $k-1$ 1's. (As a matter of fact, this idea shows that $\mathbb{Z}_k \leq_{hom} Mod\text{-}k$ and so by lemma 4.3 we need do no more.)

1) Compute n

2) Compute $m := kn$

3) Invoke $Mod\text{-}k(m, i)$

4) Compute $j := \lfloor i/k \rfloor$

5) Compute $r := i - j$

6) Case 1, $x_j = g$: then if $r = 0$ Accept else Reject

7) Case 2, $x_j = g^{-1}$: then if $r = 0$ Reject else Accept

b) Suppose that $\mathbb{Z}$ is generated by element g and that the input alphabet is as above. Then the input is equal to 1 if and only if exactly half of the input characters are g which we saw could be done in Example 1. □

We will also want to define something like subroutines or subcircuits. These would be program fragments that perform some sort of transformation on the work-tape contents. The transformation may be affected by the input but it cannot change the input. The idea is that you be able to say "do transformation T" at any stage in a computation. An example is the algorithm that puts the input length n onto a worktape. This is a *Dlogtime* algorithm which we have already accommadated. But if we want to compute some transformation that requires alternation then we have to do more.

The basic idea is to existentially guess the new tape contents and state that would result from the transformation and then for each guess universally verify that the guess is correct and that given that it is correct the computation continues on to accept. So we will write such transformations into our programs just as if they were deterministic procedures.

A subroutine we will need frequently is *Binary-Count*.

Example 3: There is a uTC^0 subroutine that puts the number of ones in the input onto a worktape.

Proof: Say we want the count to be put on worktape i. The algorithm uses the observation that if there are exactly i 1's in the string then if we append another equal length string with exactly i 0's then the new string will have exactly half of its bits being 1's.

1) Compute $n :=$ the length of the input
2) Invoke $Existential(n, i)$
(To use this as a subroutine in a program one would at this point universally verify that i is correct —by continuing the computation of this subroutine— and that given that it's correct, the rest of the computation of the calling program is accepting.)
3) Compute $m = 2n$.
4) Compute $p = n + i$.
5) Invoke $Exactly\text{-}Half(m, k)$.
6) If $k \leq n$ halt outputting x_k ; the first half of the oracle input is x
else if $n < k \leq p$ then halt outputting 0 ; the second half consists of i 0's
else halt outputting 1 ; and $n - i$ 1's

□

So given this we are able in a program to write simply:

Compute $i :=$ the number of 1's in the input in binary.

Example 4: A *Binary-Addition* subroutine is computable in *Dlogtime*.

Proof: Since the worktape strings are all $O(\log n)$ in length, one can add two such numbers deterministically in the usual way by starting with the lowest order bits and carrying to the next higher, etc. □

In general we can have a subroutine if we can recognize the graph of the transformation. We existentially guess the result of the transformation and universally verify that it is correct and that the computation goes on to accept with these new values. The point is that in describing subroutines we just need to describe graph recognition.

Example 5: There is a *Dlogtime* subroutine that returns the absolute value of the difference of two numbers.

Proof: Have to confirm that $|i - j| = k$

1) If $i \geq j$ then confirm that $i = j + k$.

2) If $i < j$ then confirm that $j = i + k$.

The previous example showed we can recognize the graph of "+." □

Example 6: There are subroutines in $ucd(\phi)$ for

a) *Multiplication*, $(b_1 \cdot b_2)$;

b) *Division*, $(\lfloor b_1/b_2 \rfloor)$; and

c) *Remainder*, $(b_1 \bmod b_2)$.

Proof: a) In Lipton [24] it is shown that multiplication of two n bit numbers is in the linear time hierarchy. It follows immediately that there is an $O(\log n)$ time constant alternation ATM that computes the product of two $O(\log n)$ bit numbers.

b) It is sufficient to show that the graph of division can be recognized. The following program accepts if and only if $d = \lfloor b_1/b_2 \rfloor$. It makes use of the multiplication algorithm of part a).

1) $Existential(b_2, i)$; guess remainder of b_1/b_2

2) If $d \cdot b_2 + i = b_1$ then Accept else Reject

c) The remainder r of b_1/b_2 is computed by $r = b_1 - \lfloor b_1/b_2 \rfloor$. □

Using the addition, subtraction and multiplication subroutines one can compute any polynomial in *Dlogtime*.

IV.C Equivalence to Previous Uniformity Definitions

The last goal of this chapter is to show that the preceding coincides with the many equivalent uniformity definitions in Barrington, Immerman and Straubing [5].

Definition: Given a circuit family $C = \{C_n\}_{n\geq 1}$, its direct connection language (DCL) is the set of tuples of the form $\langle n, g, 0, t\rangle$ or $\langle n, g, 1, h\rangle$ such that in C_n gate g is of type t and gate g has gate h as an input. The set of gate types includes *output* which distinguishes the output gate and the set of inputs $\{x_1, x_2, \ldots, x_n\}$, as well as the usual $\{AND, OR, NOT, \ldots\}$.

Definition: A circuit family is *Dlogtime*-uniform if its DCL is recognizable by a DTM using time $O(\log n)$.

This definition is equivalent (in spirit at least) to the version used in [5].

Lemma 4.4: Let L_2 be a symmetric language. If $L_1 \leq_{ucd} L_2$ then there exists a *Dlogtime*-uniform family of constant depth, polynomial size $\{AND, OR, NOT, L_2\}$-circuits that recognizes L_1.

Proof: Suppose the $ucd(L_2)$ program computing L_1 works in time bounded by $c_1 \log n + c_2$. First, the program is restructured to disallow moving into a nondeterministic state after the input has been queried (with the exception of determining n in the beginning). Let n, m, and i be variables not used in the original program.

1) The program begins by computing

$n :=$ the length of the input in binary, and

$m := 2^{c_2} n^{c_1}$; $= 2^{c_1 \log n + c_2}$

2) Every (maximal length) deterministic sequence $\vec{D}$ is replaced by:

Existential(m, i) ; i is a guess of the sequence of input bits accessed in $\vec{D}$

Universally

$U1$: deterministically confirm that the sequence of input bits accessed in $\vec{D}$ is a prefix of i, and

$U2$: simulate $\vec{D}$ using i for the results of the input queries, continuing according to the original program into the next nondeterministic state. The states of the simulation are arbitrarily chosen to be existential.

To create an L_2-circuit computing L_1, begin by associating a tree with the behavior of the modified program on inputs of length n. The root node of the tree is labeled by the configuration of the modified program that immediately follows the computation of m. The rest of the tree is created recursively by adding as a child a node for each successor configuration that has a node in the tree, with the exception of the deterministic portions described in the line marked $U1$ above. Each of these deterministic portions is represented by a single node. Note that if one follows a path from the root to a leaf in this tree only a constant number of alternations will be encountered.

Next a constant depth circuit is built as follows:

The root of the tree is the output gate and its type matches the state type t of the configuration. The rest of the circuit is constructed recursively: if N is a node of the tree represented by a gate g of type t in the circuit then every node of type other than t (except the remaining Deterministic) that is reachable from N via a path of configurations of type t is represented by a gate whose type is the state type of the configuration (or an AND gate in the case of Deterministic) and which inputs to g in the circuit. An exception to this is if t is an oracle invocation: then all of its successors have corresponding gates. It now just remains to deal with the inputs to

the AND gates that came from the deterministic portions.

After the program is modified as described above, the only deterministic configurations left are the ones that confirm that the sequence of guessed input query responses is in fact the sequence one would get in the deterministic computation. When one of these nodes is reached in the circuit building process we need the corresponding gate to output true in exactly the case that the guessed sequence i is a prefix to the sequence of input query responses one would get in the deterministic computation. Thus the guess is correct if some number $\leq |i|$ of the input bits have some specified $\{0,1\}$ value. So to the AND gate corresponding to a deterministic node we have as input x_j for each x_j accessed and specified to have value 1, and $\neg x_j$ for each x_j accessed and specified to have value 0. For each input there will be one NOT gate attached to it which will input to multiple other gates.

Claim: The circuit as just constructed computes L_1.

pf: This follows very straightforwardly by induction on the depth of the circuit. What needs to be proved is that for a given x each node of the circuit evaluates to 1 if and only if the configuration that it is labeled with is accepting.

As the base case we use the AND gates that correspond to the deterministic sequences. These deterministic sequences query a subset of the inputs and accept only if the results of the queries exactly match a predetermined sequence held on a worktape. The sequence is programmed to look at the same set of inputs regardless of the results. Such a deterministic sequence is accepting exactly when the set of inputs queried have the predetermined value and the corresponding AND gate in the circuit evaluates to 1 in just the same circumstance.

As for the induction step, it is clear that the circuit duplicates the computation.

Now we give names to the gates. The output gate is given the sequence $\langle 0 \rangle$ as its name. Input x_j is given the sequence $\langle j \rangle$ as its name. The NOT gate attached to x_j is given the sequence $\langle n + j \rangle$ as its name. To define the names of the rest of

the gates we number the descendants of each node in the computation tree. When the node represents an oracle invocation (including the *existential* and *universal* oracles) then each descendant has the value returned to the second auxiliary as its number. Otherwise the node has a constant number of descendants depending only on the state of the associated configuration. These can be numbered in an arbitrary fashion. The names of the rest of the gates (those that are neither the output, an input or the negation of an input) are given by the sequence $\langle d_1, d_2, \ldots, d_z \rangle$ of descendant numbers followed to reach the corresponding node in the computation tree starting from the root.

Claim: Given these gate names, the DCL for the circuit is recognizable in time $O(\log n)$.

proof: The DCL can be recognized in a very straightforward manner by simulating the Turing machine along the paths described by the gate names. This can easily be done in time linear in the length of the DCL tuples. Since each computation path has $O(1)$ many oracle invocations and takes time $O(\log n)$, it happens that the gate names are $O(\log n)$ in length. Thus the DCL can be recognized in time $O(\log n)$. □

Since the definition of DCL used by Barrington, Immerman and Straubing [5] does not allow nonsymmetric functions as gate types it is necessary to limit the statement of the converse.

Lemma 4.5: Let L_2 be *Majority* or *Mod-k*. If L_1 is recognized by a *Dlogtime*-uniform polynomial size constant depth family of $\{AND, OR, NOT, L_2\}$-circuits then $L_1 \leq_{ucd} L_2$.

Proof: It is a necessary consequence of the definition of *Dlogtime*-uniformity that the values of the gate names must be bounded by some polynomial function of n. Let $q(n)$ be a polynomial that for technical reasons is at least two times this bound.

Let d be the depth of the circuit family in question. The idea will be to create an oracle ATM program with L_2 as the oracle language that performs within the constraints described in the definition of $\leq_{ucd}$. The general idea is to have each gate simulated by an auxiliary type invocation with $q(n)$ successors, regardless of the actual fanin of the gate. The number returned on the second auxiliary is interpreted as a gate number and if that numbered gate is in fact an input to the simulated gate then we want it to return its usual value. However, a problem arises because we are allowing as inputs gates that were not inputs in the circuit. This presents no particular problem in the case when the gate type is *Existential* or *Mod-k*: the extra inputs can be made to return a value of 0 thereby leaving the result unchanged. Similarly if the gate type is universal then the extra inputs can be given the value 1 without changing the result. However, if $L_2 = Majority$ then neither of these solutions works. To fix this we have for each gate number two input bits; if the gate is an actual input then it returns two copies of its value, and if it is not an actual input it returns a 1 and a 0. Thus the gates that are not actual inputs to the *Majority* gate do not affect the outcome. We now give a *ucd* program that simulates a *Mod-k* or *Majority* circuit whose DCL is in *Dlogtime*.

Work tapes:

n: the length of the input.

g: the name of the gate currently being simulated.

t: the type of gate g.

q: the upper bound on the length of possible gate names $q(n)$.

i: to be used as a second auxiliary— the i^{th} input to gate g.

w: a particular prefix of i.

c: holds the depth of g in the circuit.

1) Set $c := 0$; initialize depth c

2) Compute $q := q(n)$; q is the largest possible *value* of the gate names

3) Invoke $Existential(q, g)$; existentially guess a gate name

4) Universally verify ; find the output gate

a) $\langle n, g, 0, output \rangle \in DCL$

b) continue

5) Determine gate type t of g.

6) If $t = x_j$ then access input bit j, return its value and halt.

7) Invoke a type t state with q and i as the first and second auxiliaries. (See the discussion that follows for the case of $t = NOT$.)

8) Set $w := i$ with its last bit removed and $y :=$ that last bit.

; Lines 9 and 10 verify that gate w is in fact an input to gate g in the circuit.

; They differ in how they treat the case when it is not.

; Line 9 causes gates that are not actual inputs to be treated as 0's.

9) If $t = Existential$, $Mod\text{-}k$, or $y = 0$ and $t = Majority$ then universally

9a) verify $\langle n, g, 1, w \rangle \in DCL$

9b) go to line 11

; Line 10 causes gates that are not actual inputs to be treated as 1's.

10) If $t = Universal$, or $y = 1$ and $t = Majority$ then existentially

a) verify $\langle n, g, 1, w \rangle \notin DCL$

b) go to line 11

11) Set $g := w$.

12) Set $c := c + 1$.

13) If $c = d$ then Reject.

14) Go to line 5.

The program simply simulates the circuit by guessing all possible gates as successors to the currently simulated gate and verifying that they are actual inputs. Lines 9 and 10 deal with the tricky point mentioned earlier regarding how to handle the gates that are not actual inputs.

The program contains a loop (line 14). However, line 13 assures that the loop

will be traversed only $d = O(1)$ times. Since only $O(1)$ program steps are performed, there are $O(1)$ alternations. Thus, the program is *ucd*.

The situation regarding the simulation of NOT gates mentioned in line 6 can be handled by creating a new oracle language $L_{NOT} = \{x_1 x_2 \ldots x_n : \exists i \text{ s.t. } x_{2i-1} = 1 \text{ and } x_{2i} = 0\}$. The oracle input bits would be treated as pairs: the first (odd numbered) one would test a gate name to see if it is an actual successor, and the second (even numbered) one tests whether that gate is accepting. Since only one gate is a successor to a NOT gate, the simulation accepts if and only if that successor is rejecting. □

These last two lemmas combine to make the following theorem which asserts the equivalence of the uniform circuit classes defined here and the one defined in terms of the DCL.

Theorem 4.6: Let L_2 be $Majority$ or $Mod\text{-}k$. A language L_1 is recognized by a *Dlogtime*-uniform polynomial size constant depth circuit family of $\{AND, OR, NOT, L_2\}$-circuits if and only if $L_1 \leq_{ucd} L_2$.

Proof: By lemmas 4.4 and 4.5. □

Chapter V

Group Extensions

This chapter discusses the relationship between the computational complexity of the word problems of a group and an extension of that group. In the following we assume that $G = \langle g_1, \ldots, g_m \rangle$, $H \lhd G$, $H = \langle h_1, \ldots, h_t \rangle$ and $Q \cong G/H$. Also assume that the identity element is always included in any set of generators.

Lemma 5.1: $H \leq_{hom} G$. (Recall that the word problem for a group is notationally identified with the group.)

Proof: Each generator h_i of H has a representation as a finite string of the g_j. These can be made to be equal length by padding with appropriate numbers of the identity element providing a natural homomorphism. □

Theorem 5.2: If Q is finite then $G \leq_{ucd} \{H, Q\}$.

Proof: The algorithm uses the conjugate collection process (see the proof of theorem 3.4). The basis of the algorithm is the following observation. In the following string let the q_i be from a set R of representatives for G/H and the h_i be in H.

$$q_1 h_1 q_2 h_2 \ldots q_n h_n = [w_1(a_1 h_1 a_1^{-1}) w_2(a_2 h_2 a_2^{-1}) \ldots w_n(a_n h_n a_n^{-1})] a_n$$

$$\begin{aligned} \text{where } a_i &= \text{the representative of } q_1 q_2 \ldots q_i \text{ in } R \\ w_i &= a_{i-1} q_i a_i^{-1} \end{aligned}$$

(Let $a_0 = 1$, making $w_1 = q_1 a_1^{-1} = 1$.)

Clearly for each i $w_i = 1 \bmod H$, so $w_i \in H$. Furthermore, by the normality of H in G each of the conjugates $a_i h_i a_i^{-1}$ is in H. Thus the quantity in square brackets is in H. Then since a_n is in R, the entire string is in H if and only if $a_n = 1$. Thus the entire string is equal to 1 if and only if the quantity in square brackets is equal to 1 and $a_n = 1$.

To apply this to the problem at hand, let us assume that the set of generators for G consists of the set of representatives R and some finite subset of H. (By lemma 2.9 we can make such an assumption.) Let the input be denoted by $\vec{x} = x_1 x_2 \ldots x_n$. By padding as needed with the identity element, $\vec{x}$ can be put into the form $q_1 h_1 q_2 h_2 \ldots q_n h_n$ as previously discussed. Thus, it will be sufficient to create a *ucd* program recognizing G using oracles for H and Q. The program consists of two parts: verification that $a_n = 1$ and that the quantity in square brackets is equal to 1. Only sketches of the two algorithms are given. To write programs is straightforward but requires a lot of tedious bit manipulation.

Program 1: verify $a_n = 1$. To solve this problem we simply translate each input generator into the corresponding $G/H \cong Q$ generator and solve the Q word problem. This describes a homomorphic reduction and thus the problem is solvable by a $ucd(Q)$ program.

Program 2: verify the square-bracketed quantity is equal to 1. Such a program would begin with an H-oracle invocation. A fixed amount of space in the oracle's input string would be reserved for each $(a_i h_i a_i^{-1})$ and each w_i. The a_i's are computed using the Q oracle, (since Q is finite this is straightforward), and the conjugates and the w_i are then computed via a finite table look-up. □

Corollary 5.3: If G is finite and solvable then $G \in uACC^0$.

Proof: Recall that lemma 3.3 stated that a finite solvable group is polycyclic.

The proof will be by induction on the polycyclic chain length.

Base case: if $G = \{1\}$ then $G \in uACC^0$ trivially.

Induction step: suppose that G is an extension of polycyclic finite group H by $\mathbb{Z}_k$. Then by theorem 5.2 $G \leq_{ucd} \{H, \mathbb{Z}_k\}$. By induction hypothesis $H \leq_{ucd} \mathbb{Z}_p$ for some p and it is straightforward that $\mathbb{Z}_p \leq_{ucd} \mathbb{Z}_{p \cdot k}$ and $\mathbb{Z}_k \leq_{ucd} \mathbb{Z}_{p \cdot k}$. Application of lemma 4.1 completes the proof. □

Lastly, we address the situation of $Q \cong \mathbb{Z}$. It will later be shown that groups built up from the trivial group using extensions by finite and infinite cyclic groups (the polycyclic groups) are in TC^0. One might wonder whether if $Q \cong \mathbb{Z}$ then $G \leq_{ucd} \{H, \mathbb{Z}\}$, since this would be analogous to the finite extension case and would have the polycyclic result as a corollary. This question remains unanswered. Theorem 7.9 will show that a straightforward conjugate collection approach cannot lead to an NC algorithm because exponential length intermediate strings may have to be generated. The best we can show is a corollary to the following.

Definition: A group G is a semidirect product of H by Q if G contains subgroups H and Q with the properties:

i) $H \lhd G$

ii) $QH = G$

iii) $Q \cap H = \{1\}$.

Note that this makes G an extension of H by Q.

Theorem 5.4: If H and Q are finitely generated and G is a semidirect product of H by Q then $G \leq_{Pspace} H \times Q$.

Proof: Since the output of the reduction may be exponentially long, the Turing machine we use is equiped with a write-only output tape that is not subject to the polynomial space bound. The input tape will be assumed to be two-way infinite and it will be allowable to write on it. For both G and $H \times Q$ we can assume that all

generators are in either H or Q (and except for 1, not both).

The plan is to use the use the usual conjugate collection process. In order to not use exponential space, the algorithm starts at the left end of the input and starts to push a Q generator to the right. Whenever an H-generator is at the far left it is deleted and copied onto the output tape. When there are only Q-generators left they are all copied onto the output tape.

Let Q be generated by $\{q_1, q_2, \ldots, q_s\}$ and H be generated by $\{h_1, h_2, \ldots, h_t\}$. Then for each i and j we have a substitution rule of the form $q_i h_j = h_{k_1} h_{k_2} \ldots h_{k_r} q_i$. The algorithm is:

1) Move the tape head to the far left of the string on the input tape.

2) If the character scanned is

h_j: delete it and copy it onto the output tape;

blank: halt;

q_i: continue on to line 3

3) Move the tape head rightward until an H-generator h_j is reached. Let q_i be the last Q-generator scanned. Replace $q_i h_j$ by $h_{k_1} h_{k_2} \ldots h_{k_r} q_i$, shuffling the rest of the string to make enough space for it to fit.

4) If no such h_j is found in step 3 then copy the input tape contents onto the output tape and halt.

5) Go to 1

The algorithm converts the input to an equivalent string of the form hq, with $h \in H$ and $q \in Q$. If x is the input string, then $xH = qH$ (since $x = hq = qh'$ for some $h' \in H$). Since Q is a set of representatives for G/H, $xH = 1H$ if and only if $q = 1$. Thus $x = hq = 1$ if and only if $hq = 1$ in the group $H \times Q$.

It only remains to be shown that the amount of space used is polynomial. The first thing to observe is that the number of Q-generators on the tape always remains fixed until step 4, at which point it decreases. Now suppose that the number r is the maximum number of H-generators appearing on the righthand side of a substitution

rule. Let m be the maximum of r and the input length n. Then it is easy to see that at no point do more than m H-generators appear consecutively. Since the number of Q-generators is less than or equal to n, there can be at most $mn = O(n^2)$ characters on the tape at once. (Actually linear space is sufficient.) □

Corollary 5.5: If $Q \cong \mathbb{Z}$ then $G \equiv_{Pspace} H$.

Proof: Suppose G/H is generated by aH. Then $\{a^k : k \in \mathbb{Z}\}$ is a subgroup that plays the role of Q in the definition of semidirect product. Thus by theorem 5.4 $G \equiv_{Pspace} H \times \mathbb{Z}$ and since clearly $H \equiv_{Pspace} H \times \mathbb{Z}$ it follows that $G \equiv_{Pspace} H$. □

Chapter VI

Free Groups

As stated in the preliminaries, free groups are defined by presentations with no relators. In this case a word is equivalent to 1 if and only if by repeatedly cancelling adjacent inverse pairs one arrives at the empty string, which we identify with 1. For example:

$$abb^{-1}a^{-1}b^{-1}b \Rightarrow abb^{-1}a^{-1}(1) \Rightarrow abb^{-1}a^{-1} \Rightarrow a(1)a^{-1} \Rightarrow aa^{-1} \Rightarrow 1$$

These languages have also been called the two-sided Dyck languages. Observe that they are context free languages. If G is generated by $\{x_1, \ldots, x_m\}$ then the following is a context-free grammar for G:

$$S \longrightarrow SS$$
$$S \longrightarrow g_i S g_i^{-1} \mid g_i^{-1} S g_i \quad \text{for } i = 1, 2, \ldots m$$
$$S \longrightarrow e$$

This property of free groups and some others are proved in the following lemma. But first a definition is needed.

Definition: A word is called freely reduced if it has no trivial relator as a subword.

Lemma 6.1:

(a) If G is freely generated by $\{g_1, g_2, \ldots, g_n\}$ then the word problem for G is given by the above defined two-sided Dyck language.

(b) Every element of a free group has a unique freely reduced representation.

(c) (Nielsen–Schreier Theorem) Every subgroup of a free group is free.

(d) If G is freely generated by two elements then for all m G has a free subgroup H_m on m generators.

Proof: (a) By the definition of group presentation $w = w_1 w_2 \ldots w_n = 1$ if and only if by repeatedly inserting and deleting trivial relators to and from w one can arrive at the empty string. The proposition then is that the same language is defined if only insertions are allowed and deletions are not.

It will be shown that any derivation of a string by insertions and deletions of trivial relators from the empty string can be converted to a derivation containing no deletions. Suppose the fact that $w = 1$ is demonstrated by the derivation $1 = v_0 \Rightarrow v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_q = w$ where v_{i+1} is obtained from v_i by insertion or deletion of a trivial relator. Let $v_i \Rightarrow v_{i+1}$ be the leftmost step that is a deletion. If cannot be that $i = 0$ since there is nothing to delete at that point, so it must be that $v_{i-1} \Rightarrow v_i$ is an insertion.

Case 1: the deletion step removes the same two letters that the insertion just inserted. Then both steps can be omitted and we still have a derivation of w from 1.

Case 2: the deletion step removes one letter that was already there. Then both steps can be omitted because the deleted letter that had previously been there is necessarily the same as the newly introduced letter that is not deleted.

Case 3: the deletion step removes neither of the two letters just inserted. Then the order of the two operations can be reversed while still leaving a valid derivation of w from 1.

Using these three cases we can convert any derivation from 1 to one that has no deletion preceded by an insertion. But since there cannot be any deletions before the first insertion this process must result in a derivation consisting only of insertions.

(b) From part (a) it is clear that the identity element 1 has such a unique representation (namely the empty string). Now suppose some other element has two distinct freely reduced representations v and w. We can assume that the last characters of v and w differ (or else just consider prefixes). Then $vw^{-1} = 1$ and is freely reduced, but as was just observed this requires that v and w^{-1} each be the empty word.

(c) The proof of this theorem is fairly involved. It can be found in most group theory texts (e.g. Rotman [29], pp. 242–245).

(d) Let $c_i = a^i b^i$ and $H_m = \langle c_1, c_2, \ldots, c_m \rangle$. From part (b) we know that H_m is free. It remains to show that $\{c_1, c_2, \ldots, c_m\}$ is a set of free generators.

Let $\Sigma_c = \{c_1^{\pm 1}, c_2^{\pm 1}, \ldots, c_m^{\pm 1}\}$ and $\Sigma_{ab} = \{a^{\pm 1}, b^{\pm 1}\}$. Suppose there is a freely reduced Σ_c-word equal to 1. Let $f : \Sigma_c^* \to \Sigma_{ab}^*$ map Σ_c-strings to their Σ_{ab} reduced form. The conclusion follows immediately from the following.

lemma: Let $k \geq 1, w = d_{i_1} d_{i_2} \ldots d_{i_k}$ with $d_{i_j} \in \Sigma_c$, and w be freely reduced. Then

i) if $d_{i_k} = c_j^{+1}$ then $f(w)$ ends in $a^{\pm 1} b^{+j}$

ii) if $d_{i_k} = c_j^{-1}$ then $f(w)$ ends in $b^{\pm 1} a^{-j}$.

proof of lemma: Use induction on k.

Base case: then $w = c_i^{\pm 1}$ for some i and thus $f(w)$ is either $a^i b^i$ or $b^{-i} a^{-i}$.

Induction step: suppose the proposition holds for the string $d_{i_1} d_{i_2} \ldots d_{i_{k-1}} = w'$, $d_{i_{k-1}} = c_{j_1}^{e_1}$, and $d_{i_k} = c_{j_2}^{e_2}$. Since w is freely reduced it cannot simultaneously hold that $j_1 = j_2$ and $e_1 = -e_2$.

If $e_1 = +1$ then by the induction hypothesis $f(w')$ ends in $a^{\pm 1} b^{\pm 1}$. By the uniqueness of reduced forms

$$f(w) = \begin{cases} \textit{the reduced form of } f(w') \cdot a^{j_2} b^{j_2} & \textit{if} \quad e_2 = +1 \\ \textit{the reduced form of } f(w') \cdot b^{-j_2} a^{-j_2} & \textit{if} \quad e_2 = -1 \end{cases}$$

In the first case the string is already reduced. In the second case since $j_1 \neq j_2$ the reduced form will end in $b^{\pm 1} a^{-j_2}$ as claimed.

The case of $e_1 = -1$ is similar. □

The two-sided Dyck languages are of course similar to the one-sided Dyck languages which are defined by grammars of the form

$$S \longrightarrow SS$$
$$S \longrightarrow g_i S g_i^{-1} \text{ for } i = 1, 2, \ldots, m$$
$$S \longrightarrow e$$

Usually in the one-sided case the "positive" letters are represented by open parentheses such as (, [, { and the "negative" letters by the corresponding close parentheses),], }. The languages are then the sets of balanced parentheses. In [23] it is proved that the one-sided Dyck languages are in *Alogtime*. This was later improved to:

Theorem 6.2 (Barrington and Corbett 1989 [4]): The one-sided Dyck languages are in uTC^0.

The proof is based on a generalization of the well known counting procedure used for the one-sided Dyck language on one letter. The next theorem shows that the two-sided Dyck languages are apparently more difficult.

Theorem 6.3: If G is a nonabelian free group then the word problem for G is hard for *Alogtime* with respect to *Dlogtime* reductions.

Proof: The reduction will have to code accepting computations of an *Alogtime* machine as identity strings and nonaccepting computations as nonidentity strings. The basic idea is contained in the following lemma, which shows how to simulate AND and OR.

Lemma: Let $\Sigma = \{a^{\pm 1}, b^{\pm 1}\}$, $w, x \in \Sigma^*$; further suppose $|w| = |x| = k$ and that the net occurence of b in both w and x is 0. If w and x are interpreted as elements of the free group generated by a and b then:

i) $(w = 1) \vee (x = 1) \iff b^{-3k} w b^{3k} x b^{-3k} w^{-1} b^{3k} x^{-1} = 1$

ii) $(w = 1) \wedge (x = 1) \iff b^{-3k}wb^{3k}xb^{-3k}wb^{3k}x^{-1} = 1$

Pf: Call the strings on the righthand side above $f_\vee$ and $f_\wedge$. In both cases the forward direction of implication ($\Rightarrow$) is apparent. As for the reverse direction:

Claim: if $w \neq 1$ then the reduced form of $b^{-k}wb^k$ begins with b^{-1} and ends with b^{+1}.

Pf of claim: Since the net occurence of b in w is 0, and $w \neq 1$, w must contain some occurences of $a^{\pm 1}$ in its reduced form. This implies that in obtaining the reduced form of $b^{-k}wb^k$ none of the left block of b^{-1} will be able to cancel with any of the right block of b^{+1}. Since w has $< k$ occurences of b^{-1} or b^{+1}, cancellation cannot remove the entirety of either the b^{-k} block or the b^{+k} block.

Similar reasoning shows:

1) if $x \neq 1$ the reduced form of b^kxb^{-k} begins with b^{+1} and ends with b^{-1};

2) if $w \neq 1$ the reduced form of $b^{-k}w^{-1}b^k$ begins with b^{-1} and ends with b^{+1};

3) if $x \neq 1$ the reduced forms of b^kx and b^kx^{-1} begin with b^{+1}.

Case i, (OR): Rewrite $f_\vee(x,w)$ as $b^{-2k}(b^{-k}wb^k)b^k(b^kxb^{-k})b^{-k}(b^{-k}w^{-1}b^k)b^k(b^kx)$. The preceding claims show that if $x \neq 1$ and $w \neq 1$ then the reduced form of $f_\vee(x,w)$ will be other than the empty string. (If one reduces the blocks in parentheses then the entire string will be in reduced form.)

Case ii, (AND): Reasoning similar to the OR case shows that if $x \neq 1$ and $w \neq 1$ then the reduced form of $f_\wedge(x,w)$ is not the empty string. Now suppose $x = 1$ and $w \neq 1$. Then $f_\wedge(x,w) = b^{-3k}wwb^{3k}$. Since no nonidentity string is its own inverse (see lemma 6.1(c)), ww is not the identity. Since the identity is conjugate to no nonidentity string it follows that $b^{-3k}wwb^{3k} \neq 1$.

Similarly, if $x \neq 1$ and $w = 1$ then $f_\wedge(x,w) = xx \neq 1$.

q.e.d. lemma

Now suppose L is recognized by an ATM M using time $O(\log n)$. The following can be assumed about M without loss of generality:

i) All computation trees are completely balanced. (I.e., for a given n all computation paths have the same length.)

ii) Each nonhalting state has exactly two successors which are distinguished as left and right.

To define the reduction $f(\vec{x})$ we start by associating a Σ-string $f_N(\vec{x})$ with each node N of M's computation tree on input $\vec{x}$. If N is a leaf node then:

i) If N is accepting, then $f_N(\vec{x}) = a^8a^{-8}$.

ii) If N is nonaccepting, then $f_N(\vec{x}) = a^{16}$.

If N is an internal node with left successor N_l and right successor N_r, and $|f_{N_l}(\vec{x})| = |f_{N_r}(\vec{x})| = k$, then:

i) If N is existential $f_N(\vec{x}) = b^{-3k}[f_{N_l}(\vec{x})]b^{3k}[f_{N_r}(\vec{x})]b^{-3k}[f_{N_l}(\vec{x})]^{-1}b^{3k}[f_{N_r}(\vec{x})]^{-1}$

ii) If N is universal $f_N(\vec{x}) = b^{-3k}[f_{N_l}(\vec{x})]b^{3k}[f_{N_r}(\vec{x})]b^{-3k}[f_{N_l}(\vec{x})]b^{3k}[f_{N_r}(\vec{x})]$

Note: because the computation tree is completely balanced it will necessarily occur that $|f_{N_l}(\vec{x})| = |f_{N_r}(\vec{x})|$.

The reduction function can now be defined as $f(\vec{x}) = f_R(\vec{x})$ where R is the root node of M's computation tree on input $\vec{x}$. It is clear from use of induction on the above lemma that $f(\vec{x}) = 1$ if and only if M accepts $\vec{x}$.

It only remains to show that f is a *Dlogtime* reduction. Let $n = |\vec{x}|$. First we confirm that $|f(\vec{x})| \leq p(n)$ for some polynomial p. Suppose M uses time $c_1 \log n + c_2$. Then since if $n = 1$ we have $|f(\vec{x})| = 16$ and increasing the depth by 1 increases $|f(\vec{x})|$ by a factor of 16 we have:

$$|f(\vec{x})| = 16^{depth+1} = 16^{c_1 \log n + c_2 + 1} = 16^{c_2+1} \cdot n^{4c_1}$$

which is a polynomial function of n.

Next we show that the i^{th} bit of $f(\vec{x})$ can be computed in time $O(\log n)$. The basic idea is that by viewing i as a base 16 number we can easily follow a path through the computation tree of M to the single leaf that $\hat{f}(x,i)$ depends on, or determine that $\hat{f}(x,i) = b^{\pm 1}$. For example, the first four bits of i tell which sixteenth of the output string i is in, (assume all i are padded with leading zeroes to make them of equal length). So i might occur in a block of b or b^{-1}, or it might occur in a substring

corresponding to the left of right subtree of the computation. In the latter case we simulate the computation one step and repeat the process with the next four bits.

In the following description of the *Dlogtime* algorithm, assume for technical reasons that the outputs are numbered 0 to $p(n) - 1$ instead of 1 to $p(n)$.

(A) *START UP*

A1 We have M in its initial configuration

state := the type (*existential*, *universal* or *halting*) of the current configuration of the ATM simulation. This variable is assumed to be updated automatically as the simulation proceeds.

A2 Get n in binary

A3 Compute $z = p(n) - 1$ —the length of the output string

A4 If $|i| \leq |z| - 4$ output b^{-1} —because then i is in the initial sixteenth

A5 If $|i| < |z|$ then output blank —because then i is too big

A6 If $|z| - 4 < |i| < |z|$ then pad i with leading 0's to make $|i| = |z|$

A7 Put the index tape head on the leftmost square

A8 Set *sign* := +1

(B) *LOOP*

B1 *code* := binary number encoded by the next 4 bits of i

B2 If *state* = *halting*

Case 1: configuration is accepting

If *code* < 8 output a else output a^{-1}

Case 2: configuration is rejecting

If *sign* = +1 output a else output a^{-1}

B3 Case 1: *sign* = +1 —must be in a nonhalting configuration here

If *code* =

0,1,2,8,9,10: output b^{-1}

4,5,6,12,13,14: output b^{+1}

3: simulate M one step taking the left branch; go to (B)

7: simulate M one step taking the right branch; go to (B)

11: if *state* is *existential* then set $sign := -sign$

Regardless of *state* simulate M one step taking the left branch

15: if *state* is *existential* then set $sign := -sign$

Regardless of *state* simulate M one step taking the right branch

B4 Case 2: $sign = -1$

If *code* =

1,2,3,9,10,11: output b^{-1}

5,6,7,13,14,15: output b^{+1}

0: if *state* is universal then set $sign := -sign$

Regardless of the *state* simulate M one step taking the right branch

4: if *state* is universal then set $sign := -sign$

Regardless of the *state* simulate M one step taking the left branch

8: simulate M one step taking the right branch; go to (B)

12: simulate M one step taking the left branch; go to (B)

The program looks at string i in four-bit blocks. Part A just makes sure that i is the right length: if it is too long, the program halts; if it is much too short, the program outputs b^{-1} since i is then in the initial sixteenth; and otherwise i is padded with an appropriate number of 0's.

Part B takes four bits of i and simulates the left or right branch of the ATM as appropriate. Some care is needed regarding the case when the subtree simulated is within the scope of an odd number of negative exponents. In that case the corresponding substring that is generated is the inverse of what would have been (i.e., the string is reversed and all exponents negated).

That the program runs in time $O(\log n)$ is clear: a constant amount of processing is done for each bit of i, and the i we are concerned with are $O(\log n)$ in length. □

As to the question of what complexity class the free group word problems are in, we have the following:

Theorem 6.4 (Lipton and Zalcstein, 1977 [25]): For any k, the problem of determining whether a product of m $k \times k$ integer matrices is equal to the identity is decidable in *Dlogspace*.

Proof: Given input $\vec{A} = A_1, A_2, \ldots, A_m$, $k \times k$ integer matrices we want to determine if their product is I. Let $d =$ the maximum absolute value of any entry in any of the input matrices. Let $n =$ the overall length of the input.

The proof is based on the Chinese remainder theorem, which states that if $p_1, p_2, \ldots, p_k$ are distinct primes and $|n| < \Pi p_i$ then $n = 0$ if and only if $n \equiv_{mod\ p_i} 0$ for all i.

Let $\mu(n) =$ the product of all primes $\leq n$. We need the following fact from number theory:

Lemma 1: For some constant $c > 0$, $\mu(n) > 2^{cn}$.

The proof can be found in [17].

We define a norm on matrices: if $A = (a_{ij})$ then $|A| = \Sigma_{i,j=1}^{k} |a_{ij}|$.

Lemma 2: $|A \cdot B| \leq k^2 |A||B|$.

Proof: Each entry of $A \cdot B$ has absolute value $\leq |A||B|$, and there are k^2 entries. *q.e.d. lemma 2*

Lemma 3: $|A_1 \cdot A_2 \cdot \ldots \cdot A_m| \leq k^{2(m-1)} |A_1| \cdot \ldots \cdot |A_m|$.

Proof: Use induction on m and Lemma 2. *q.e.d. lemma 3*

We are going to check whether $A_1 \cdot A_2 \cdot \ldots \cdot A_m - I \equiv_{mod\ p} 0$ for $p = 2, 3, 4, \ldots, z$. The question is, how large does z need to be as a function of n in order to assure that $A_1 \cdot A_2 \cdot \ldots \cdot A_m$ is in fact equal to the identity.

The Chinese remainder theorem says that z must be such that $\mu(z) \geq |A_1 \cdot A_2 \cdot \ldots \cdot A_m|$.

$$|A_1 \cdot A_2 \cdot \ldots \cdot A_m| \leq k^{2(m-1)} |A_1| \ldots |A_m| \quad \text{by lemma 3}$$

$$
\begin{aligned}
&\leq k^{2(m-1)}(k^2d)^m \quad \text{since } |A_i| \leq (k^2d) \\
&= k^{4m-2}d^m \\
&\leq k^{4n-2}(2^n)^n \text{ since } 2^n \geq d,\ n \geq m \\
&= 2^{n^2+(4n-2)\log k} \\
&\leq 2^{q(n)} \quad \text{for some polynomial } q \\
&\leq 2^{cz} \text{ for } z = q(n)/c \\
&\leq \mu(z) \quad \text{by lemma 1}
\end{aligned}
$$

So a polynomially large z will suffice. Now it is a simple matter to test the matrix product in space $O(\log z) = O(\log n)$. □

Corollary 6.5: The word problems for free groups are decidable in logspace.

Proof: The matrices $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$ generate a free group ([26], pages 100–101). By lemma 6.1(d), for any k the free group on k generators is a subgroup of the free group on 2 generators, so the result follows from lemma 5.1. □

All attempts to improve this to NC^1 have failed so far. The best we have attained is:

Theorem 6.6: The word problem for any finitely generated integer matrix group is decidable by polynomial size, bounded fanin circuit families of depth $O(\log n \frac{\log\log n}{\log\log\log n})$.

Proof: Recall that there exist polynomial size, $O(\log n)$ depth circuits for iterated addition, iterated multiplication and x mod m, (theorems 2.4 and 2.5).

Let the input be of length n and consist of the $k \times k$ matrices $A_1, \ldots, A_m$.

The basic idea is as in the proof of theorem 6.4. The circuit will compute the AND of $A_1 \cdot A_2 \cdot \ldots \cdot A_m \equiv_{mod_p} I$ for $p = 2,3,4,\ldots,z$. As seen in theorem 6.4, a polynomial size z will suffice. Note that since the A_i are chosen from a finite set the length of the input n is $O(m)$.

The general approach is to break the input into blocks and "hardwire in" the pattern of additions and multiplications, and reduce mod-p afterwards. For example, if we choose a block size of k we get:

$$\begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{1,3} & x_{1,4} \end{bmatrix} \begin{bmatrix} x_{2,1} & x_{2,2} \\ x_{2,3} & x_{2,4} \end{bmatrix} \cdots \begin{bmatrix} x_{k,1} & x_{k,2} \\ x_{k,3} & x_{k,4} \end{bmatrix} = \begin{bmatrix} f_1 & f_2 \\ f_3 & f_4 \end{bmatrix}$$

where $f_i = \sum_{z=1}^{2^k} \prod_{j=1}^{k} y_{z,j}$ with each $y_{z,j}$ being one of the input entries $x_{c,d}$.

Now let $k = \log\log n$, and consider how to compute $f_i \bmod p$. Each entry is $O(p)$ in value and thus $O(\log n)$ in size. Using the vector multiplication subcircuit each product in the sum can be computed by a $\log\log n$ depth, polylog size circuit. The resulting product is of size $O(\log n \log\log n)$ bits, which can then be reduced mod-p to size $O(\log n)$ by a polylog size, depth $O(\log\log n)$ subcircuit. The sum then consists of $2^{\log\log n} = \log n$ of these products which can be computed in polylog size and depth $\log\log n$, and then reduced mod-p in polylog size and depth $O(\log\log n)$. The total subcircuit is then $O(\log\log n)$ in depth and polylog in size.

The subcircuit that computes the entire matrix product mod-p consists of a balanced tree of the above subcircuit. The size of this tree is easily $O(n)$ and its depth d is such that $(\log\log n)^d = n$, which means that $d = \frac{\log n}{\log\log\log n}$. The total depth of the mod-p circuit is this d times the depth of a subcircuit, $\log\log n$. The total size is the product of the size of the tree and the size of the subcircuit, which is polynomial in n.

For each p the result must be tested for equality to $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. There are $z = poly(n)$ of these mod-p circuits (one for each integer $p \leq z$), and they are connected by a polynomial size $O(\log n)$ depth AND tree. The result is a circuit of the stated size and depth. □

As stated earlier, the word problem for free groups is a context free language. The following theorem discusses groups whose word problem is context free.

Theorem 6.7: (Muller and Schupp, 1983 [28]) Let G be a finitely generated group. G has a free subgroup of finite index if and only if G is context free.

Actually, Muller and Schupp proved that G has a free subgroup of finite index if and only if G is context and accessible (a term that need not be defined here). It was later pproved by Dunwoody [12] that all finitely presented groups are accessible. Since a context free group clearly must be finitely presented we can omit mention of accessibility in the theorem.

This allows us to state the following:

Theorem 6.8: If G is context free and F is a nonabelian free group, then $G \leq_{ucd} F$.

Proof: By theorem 6.7, G has a free subgroup H of finite index. By lemma 2.6, H has a finitely generated subgroup N such that $N \lhd G$ and N is of finite index in G. Let $Q \cong G/N$. Then by theorem 5.2 $G \leq_{ucd} \{N, Q\}$. Since any subgroup of a free group is also free (lemma 6.1(b)) we have that N is free. By lemma 6.1(d) N is isomorphic to a subgroup of F, so $N \leq_{ucd} F$. On the other hand the fact that F is hard for *Alogtime* with respect to *Dlogtime* reductions shows $Q \leq_{ucd} F$. Since trivially $G \leq_{ucd} \{N, Q, F\}$, we have $G \leq_{ucd} F$ by lemma 4.1. □

The free group $\langle a, b; \rangle$ is identical to the free product $\langle a; \rangle \otimes \langle b; \rangle$. We now address the more general problem of relating the complexity of a free product to the complexity of its free factors. First we show how free products relate to other classes of groups discussed in this dissertation.

Lemma 6.9: Let $G = H \otimes K$ with neither H nor K the trivial group.

(a) If $H \cong K \cong \mathbb{Z}_2$ then H is solvable but not nilpotent.

(b) If $H \ncong \mathbb{Z}_2$ or $K \ncong \mathbb{Z}_2$ then G has a nonabelian free subgroup and thus is nonsolvable.

Proof: (a) If $H \cong K \cong \mathbb{Z}_2$ then G has presentation $\langle a, b; a^2, b^2 \rangle$. Since $a^{-1} = a$, $b^{-1} = b$, and $a^2 = b^2 = 1$ every word is equivalent to one with no negative exponents and no consecutive occurences of a or b. So

$$G = \{b^{e_1}(ab)^k a^{e_2} : e_1, e_2 \in \{0,1\} \text{ and } k \geq 0\}$$

To determine $G^{(1)}$ first observe that

$$G^{(1)} = G_2 = \{(ab)^{2k} : k \geq 0\} \cup \{(ba)^{2k} : k \geq 0\} = \{(ab)^{2k} : k \in \mathbb{Z}\}$$

Containment is shown by the fact that every product of commutators will have counts of a and b that are both multiples of 2. To see the reverse containment observe that

$$\begin{aligned} [a][(ba)^k][a]^{-1}[(ba)^k]^{-1} &= a[(ba)^k]a(ab)^k \\ &= a[b(ab)^{k-1}a]a(ab)^k \\ &= (ab)(ab)^{k-1}(ab)^k \\ &= (ab)^{2k} \end{aligned}$$

Since $G^{(1)}$ is abelian $G^{(2)} = \{1\}$ and thus G is solvable. To see that it is not nilpotent observe that if $(ab)^p \in G_i$ then G_{i+1} contains

$$\begin{aligned} [(ab)^p][a][(ba)^p][a]^{-1} &= (ab)^p a \cdot b(ab)^{p-1}a \cdot a \\ &= (ab)^p(ab)(ab)^{p-1} \\ &= (ab)^{2p} \end{aligned}$$

(b) Suppose $K \ncong \mathbb{Z}_2$. Then K has nonidentity elements k_1 and k_2 such that $k_1 \neq k_2^{-1}$. (Note: it may be that $k_1 = k_2$.) Let $h \in H$, $h \neq 1$. The elements $x = k_1 h k_2$ and $y = h k_1 h k_2 h$ generate a free group of rank 2.

To see this, suppose w is a freely reduced word on $\{x^{\pm 1}, y^{\pm 1}\}$. Let w' denote the freely reduced word representing w in the alphabet $\{h^{\pm 1}, k_1^{\pm 1}, k_2^{\pm 1}\}$. It can readily be seen by induction on the length of w that if w ends in

x: then w' ends in k_2

x^{-1}: then w' ends in k_1^{-1}

y: then w' ends in $k_2 h$

y^{-1}: then w' ends in $k^{-1}h^{-1}$

Thus no nonempty freely reduced string is equal to 1 so $\langle x, y \rangle$ is free. □

Theorem 6.10: If $G = H \otimes K$ then the word problem for G is decidable by a family of polynomial size $O(\log n)$ depth unbounded fanin $\{AND, OR, NOT, H, K\}$-circuits.

Proof: Assume without loss of generality that n is a power of 2. The circuit will have $\log(n)$ layers. Each layer will be a sequence of n blocks of characters. Every block will contain either all H-generators, all K-generators, or only the identity element. (The identity element is included among the H- and K-generators.) A block filled completely with the identity element is denoted by I.

In the 0^{th} layer (the layer nearest the input) each block will consist of one character; the i^{th} block will contain the i^{th} input x_i. In general, the blocks on the j^{th} layer will be 2^j characters long. The product of the sequence of blocks will always be equal to the input $x = x_1 x_2 \ldots x_n$.

The sequence of blocks in each layer will be divided into subsequences. In the j^{th} layer the blocks will be grouped into subsequences of size 2^j. We now create a binary tree with the set of subsequences as nodes. The root of the tree is labeled by the one subsequence of the top layer (layer number = $\log(n)$), which is of length n. It has two children. The left child is the subsequence consisting of the first $n/2$ blocks of the $(\log n) - 1$ layer, and its right child consists of the remaining $n/2$ blocks on that layer. In this manner the tree extends down to the 0-layer at which each subsequence consists of one block.

Each block will have these properties:

1) A block is equal to the identity if and only if it is I.

2) Adjacent blocks within a subsequence do not contain the same type (H or K) of generator.

3) If the subsequence is a left child in the tree then no I block occurs to the right of

a non-I block.

4) If the subsequence is a right child in the tree then no I block occurs to the left of a non-I block.

Generally, circuit works by putting each subsequence into a reduced form – one in which blocks of nonidentity H and K blocks alternate, and these nonidentity blocks are pushed all the way to one end of the subsequence. Since there are $\log(n)$ layers it will be sufficient to show that the computation between two layers can be done in constant depth. In fact, since the computation of each subsequence in a layer can be done independently of the other subsequences on that layer we need only look at one such portion of a layer.

Let a_l and a_r be the left and right subsequences respectively that feed into subsequence a on the next higher level. Let $a_l = b_1 b_2 \ldots b_m$ and $a_r = c_1 c_2 \ldots c_m$ where the b_i and c_i are blocks. Further, we are given that the nonidentity blocks of a_l are pushed to the right and those of a_r are pushed to the left. Assume for the sake of definiteness that we need the nonidentity blocks of a pushed to the right. The resulting sequence of blocks, called $d_1 d_2 \ldots d_{2m}$ will have to be in reduced form and equal to the product $(b_1 b_2 \ldots b_m)(c_1 c_2 \ldots c_m)$. The plan is simply to cancel as many b-c pairs as possible and push the remaining blocks to the left.

Define a set of propositions:

$$\begin{aligned} P_i &\cong (b_{m-i+1} c_i = 1) \\ Q_k &\cong (P_1 \wedge P_2 \wedge \ldots \wedge P_k) \wedge \neg P_{k+1} \\ R &\cong \text{“}b_m \text{ and } c_1 \text{ come from different free factors”} \\ S_i &\cong (b_i = I) \wedge (b_{i+1} \neq I) \end{aligned}$$

Then if $R \wedge S_i$ holds we have

$$d_j = \begin{cases} b_{i+j} & \text{if } j \leq m - i \\ c_{i+j-m} & \text{if } j > m - i \end{cases}$$

If $\neg R \wedge S_i \wedge Q_k$ holds then

$$d_j = \begin{cases} b_{i+j} & \text{if } j < m-k-i \\ b_{i+j} \circ c_{k+1} & \text{if } j = m-k-i \\ c_{j-m+2k+i+1} & \text{if } j > m-k-i \\ 1 & \text{if } j \geq 2m-2k-i \end{cases}$$

Using these relations, we can determine in constant depth which b- or c-block should fill a given d-block. It is a simple matter then to transfer the appropriate set of bits and fill in the back half with I. The one exception is when the d-block is filled with the concatenation $b_{i+j} \circ c_{k+1}$. In that case b_{i+j} is placed in the front half and c_{k+1} in the back half.

In the end the top layer will either contain a sequence of n I-blocks or it will contain some non-I block. The circuit will test for the former condition and return the result as the output. □

Chapter VII

Nilpotent Groups and Extensions

This chapter addresses the class of (finitely generated) nilpotent groups and their finite extensions. In section 7.1 it is shown that the word problems of nilpotent groups are in uTC^0. The algorithm is based entirely on the commutator collection process. In section 7.2 the results of Chapter 5 are applied giving classifications to the groups that are extensions of nilpotent groups by finite solvable and finite nonsolvable groups. Both of these classes turn out to have interesting alternative characterizations based on the "growth function" of the group, which are discussed. Lastly, it is shown that straightforward application of conjugate collection cannot give NC circuits for any other groups than these.

VII.A Nilpotent Groups

We begin by repeating the definition of nilpotent given in Chapter 2. The lower central series is defined recursively by

$$\begin{aligned} G_1 &= G \\ G_{i+1} &= [G_i, G] \end{aligned}$$

G is nilpotent if there exists an α such that $G_\alpha = \{1\}$.

The following lemma expresses the critical property of nilpotent groups upon

which the algorithm is based. As before, for any m $\Sigma_m = \{g_1^{\pm 1}, g_2^{\pm 1}, \ldots, g_m^{\pm 1}\}$.

Lemma 7.1: It is possible to choose a sequence of generators $g_1, g_2, \ldots, g_m$ of G such that for every $x \in G$ there exist integers $p_1, \ldots, p_m$ such that $x = g_m^{p_m} g_{m-1}^{p_{m-1}} \ldots g_1^{p_1}$, and further such that for all $e, s \in \{\pm 1\}$ and all i, j with $i > j$ we have $g_j^{-e} g_i^s g_j^e g_i^{-s} \in \Sigma_{j-1}^*$.

Proof: Suppose $G_\alpha = \{1\}$. It is a well known fact about finitely generated nilpotent groups that for all j the quotient group G_j/G_{j+1} is finitely generated abelian. It is straightforward to prove the lemma by induction on i, but a less formal presentation is a bit easier to follow.

Suppose for example that the set $\{h_1 G_2, h_2 G_2, \ldots, h_k G_2\}$ generates G_1/G_2. Since G_1/G_2 is abelian it follows that each element is equal to $h_1^{e_1} h_2^{e_2} \ldots h_k^{e_k} \gamma_2$ for some sequence of exponents $\vec{e}$ and some $\gamma_2 \in G_2$. Similarly we can write g_2 in the form $q_1^{d_1} q_2^{d_2} \ldots q_z^{d_z} \gamma_3$ for some set $\{q_1, q_2, \ldots, q_z\}$ of coset representatives of G_2/G_3, some sequence of exponents $\vec{d}$ and some $\gamma_3 \in G_3$. We can continue in this fashion until we reach G_α, the trivial group. The sequence (in reverse order) is then $g_m, g_{m-1}, \ldots, g_1 = h_1, h_2, \ldots, h_k, q_1, q_2, \ldots q_z, \ldots$.

To see that $i > j$ implies $g_j^{-e} g_i^s g_j^e g_i^{-s} \in \Sigma_{j-1}^*$ simply observe that if $g_j \in G_\beta - G_{\beta-1}$ then any commutator involving g_j is in $G_{\beta+1}$ and thus can be represented by generators all with indices lower than j. □

The important thing to notice is that the commutator is shown to be in Σ_{j-1}^*, not just in Σ_{i-1}^*.

The general idea is to use commutator collection to eliminate all occurences of the symbol g_m. For readability let $z = g_m$. For each $y = g_i^{\pm 1}$ with $i < m$ and $s = \pm 1$ we have "replacement rules" of the form

$$z^s y z^{-s} = y \vec{w}$$

where $\vec{w}$ is a string in the alphabet $\Sigma_{m-1} \cup \{1\}$. Since each string can be padded with the character 1 without changing its value, we can assume that for all y and s,

$|\vec{w}| = p$ for the same integer p.

We need to convert $z^k y z^{-k}$ to a string in the alphabet $\Sigma_{m-1} \cup \{1\}$, and it has to be done in such a way that the length of the new string is bounded by a polynomial in k. It turns out that the most straightforward approach of simply applying the replacement rules repeatedly works. For example, if we have the rule $zyz^{-1} = yw_1w_2 \ldots w_p$, then (assuming for this example that $k > 0$)

$$\begin{aligned} z^k y z^{-k} &= z^{k-1}(zyz^{-1})z^{-(k-1)} \\ &= z^{k-1} y w_1 w_2 \ldots w_p z^{-(k-1)} \\ &= (z^{k-1} y z^{-(k-1)})(z^{k-1} w_1 z^{-(k-1)})(z^{k-1} w_2 z^{-(k-1)}) \ldots (z^{k-1} w_p z^{-(k-1)}) \end{aligned}$$

Thus the exponent k is reduced by one and this process can be continued until the exponents reach zero.

Now we want to formalize this by defining the function $R(i, e, s, k) =$ the transformation of $z^{sk} g_i^e z^{-sk}$, where $e, s \in \{\pm 1\}$ and $k \geq 0$. For each $e, s \in \{\pm 1\}$ and $1 \leq i < m$ we have a replacement rule

$$z^s g_i^e z^{-s} = g_i^e \vec{w}(i, e, s)$$

where $\vec{w}(i, e, s)$ is a string of length p in the alphabet $\Sigma_{m-1} \cup \{1\}$. To these we add

$$z^s 1 z^{-s} = 1$$

for $s \in \{\pm 1\}$. Note that the righthand side of this rule is not of length p. For the sake of exposition let $g_0 = 1$.

Now we can formally define the function R recursively by

$$\begin{aligned} R(0, e, s, 1) &= 1 \qquad --(\text{i.e., } z^s 1^e z^{-s} = 1) \\ R(i, e, s, 0) &= g_i^e \qquad --(z^0 g_i^e z^0 = g_i^e) \\ R(i, e, s, k+1) &= [R(i, e, s, k)] \circ ([R(j_1, e_1, s, k)] \circ \ldots \circ [R(j_p, e_p, s, k)]) \end{aligned}$$

where $\vec{w}(i, e, s) = g_{j_1}^{e_1} g_{j_2}^{e_2} \ldots g_{j_p}^{e_p}$.

As stated before, it is a necessary condition to do this reduction that the length of the string produced from a conjugate of the form $z^{sk} g_i^e z^{-sk}$ be bounded by

a polynomial in the variable n. (Note that k will always be $\leq n$.) The following function defined for $0 \leq i < m$ and $k \geq 0$ will serve as that bound.

$$\begin{aligned} f(0,k) &= 1 \\ f(i,k) &= \binom{k}{i-1}p^{i-1} + \binom{k}{i-2}p^{i-2} + \ldots + \binom{k}{0}p^0 \quad \text{for } i \geq 1 \end{aligned}$$

Note that for fixed i and p, f is a polynomial in the variable k.

Lemma 7.2: The function f has the following properties:

1. $\forall 0 \leq i < m \quad f(i,0) = 1$

2. $\forall i \forall k \geq 1 \quad f(i+1,k+1) = f(i+1,k) + pf(i,k)$

Proof: Property 1 follows from the fact that $\binom{0}{a}$ is 0 if $a \neq 0$ and 1 if $a = 0$. As to property 2 we show that for all $j \geq 0$ the coefficient of p^j on the left side is equal to the corresponding coefficient on the right side. The coefficient of p^0 is 1 on both sides. For $j > 0$ the coefficient of p^j on the right side is $\binom{k}{j} + \binom{k}{j-1}$ while the coefficient on the right side is $\binom{k+1}{j}$. Since $\binom{k}{j} + \binom{k}{j-1} = \binom{k+1}{j}$ is a binomial coefficient identity, both sides are equal.□

Lemma 7.3: The length of the string $R(i,e,s,k)$ is bounded by $f(i,k)$.

Proof: Induction on k.

If $k = 0$ then $|R| = 1$ and $f = 1$, so the bound holds.

For the induction step, observe that if $i \neq 0$ then $R(i,e,s,k+1)$ is a concatenation of $p+1$ strings. The induction hypothesis states that the length of first string is bounded by $f(i,k)$ and that the others are bounded by functions bounded by $f(i-1,k)$. Thus the total length is bounded by $f(i,k) + pf(i-1,k) = f(i,k+1)$. For the special case of $i = 0$ we have $R(0,e,s,k+1) = 1$, and the bound holds. □

So the conversion of $z^{sk}g_i^e z^{-sk}$ will have length $\leq f(i,k) \leq f(i,n)$ since we will always have $k \leq n$. The function f is the length of the string that would be obtained

if all of the replacement rules were of the form $z^s g_i^e z^{-s} = g_i^e (g_{i-1})^{\pm p}$: that is, if the indices of the elements of $\vec{w}(i,e,s)$ were always the largest possible $(i-1)$.

We are now ready to state and prove the main theorem:

Theorem 7.4: If G is nilpotent then the word problem for G is in uTC^0.

Proof: The proof will be by induction on the m of lemma 7.1. The subgroup $H = \langle g_1, \ldots, g_{m-1} \rangle$ of G is nilpotent and the elements $\{g_1, \ldots, g_{m-1}\}$ form a set of $m-1$ generators of H with the properties described in lemma 7.1. Thus, after the base case is proved it will be sufficient to show that $G \leq_{ucd} \{H, \mathbb{Z}\}$. To be more explicit: we need to show that $G \leq_{ucd} \{H, \mathbb{Z}\} \Longrightarrow G \leq_{ucd} Majority$ and this can be done with the following chain of implications:

$$\begin{aligned} G \leq_{ucd} \{H, \mathbb{Z}\} &\Longrightarrow G \leq_{ucd} \{H, \mathbb{Z}, Majority\} \\ &\Longrightarrow G \leq_{ucd} \{H, Majority\} \text{ since } \mathbb{Z} \leq_{ucd} Majority \\ &\Longrightarrow G \leq_{ucd} Majority \text{ by induction hypothesis} \end{aligned}$$

First the base case:

Claim 1: If $m = 1$ then $G \in uTC^0$.

proof: If $m = 1$ then G is a cyclic group and by example 2 of Chapter 4 $G \in uTC^0$. *q.e.d. claim 1.*

Next the induction step.

Claim 2: $G \leq_{ucd} \{H, \mathbb{Z}\}$.

proof: If the index of H is finite (i.e., G is a finite extension of H) then $Q \cong \mathbb{Z}_p$ for some p. By theorem 5.2 $G \leq_{ucd} \{H, \mathbb{Z}_p\}$. Thus $G \leq_{ucd} \{H, \mathbb{Z}_p, \mathbb{Z}\}$ and since $\mathbb{Z}_p \leq_{ucd} \mathbb{Z}$ we have $G \leq_{ucd} \{H, \mathbb{Z}\}$ (lemma 4.1). The rest of the proof deals with the case of G/H being infinite; i.e., $G/H \cong \mathbb{Z}$.

The algorithm uses the conjugate collection process, which is based on the fact that the input $x = x_1 x_2 \ldots x_n$ can be rewritten in the form

$$(z^{s_1 k_1} y_1 z^{s_1 k_1})(z^{s_2 k_2} y_2 z^{-s_2 k_2}) \ldots (z^{s_n k_n} y_n z^{-s_n k_n}) z^{s_n k_n}$$

where

$$y_i = \begin{cases} x_i & \text{if } x_i \neq z^{\pm 1} \\ 1 & \text{if } x_i = z^{\pm 1} \end{cases}$$

k_i = the absolute value of the net count of z in $x_1 x_2 \ldots x_i$,

s_i = the sign of the net count of z in $x_1 x_2 \ldots x_i$.

Each of the conjugates is in H so $x \in H$ if and only if $z^{s_n k_n} \in H$, which is to say, if and only if $k_n = 0$.

The oracle Turing machine that computes G from oracles for H and $\mathbb{Z}$ is outlined as follows:

1. Universally verify

   1a) that $k_n = 0$, and

   1b) the conjugate string $= 1$ in H.

These two parts will be treated separately. As for part 1a let $\mathbb{Z}$ be represented by the alphabet $\{a, a^{-1}, 1\}$. Then if S = the set of inputs that make $k_n = 0$ then $S \leq_{hom} \mathbb{Z}$ via the mapping

$$z^s \longmapsto a^s$$

$$x_i^s \longmapsto 1 \text{ for } x_i \neq z$$

So by lemma 4.3 there is a $ucd(\mathbb{Z})$ program for part 1a.

The problem of verifying that the conjugate string $= 1$ in H requires more explanation. Generally, what needs to be done is quite simple. The conjugate string is equivalent to a string in $\Sigma_{m-1} \cup \{1\}$ of length polynomial in n and thus so is the product of all the conjugate strings. The Turing machine will simply make a polynomial size invocation using oracle H and present that polynomial length string to the oracle. The only question is whether we can determine easily enough what the i^{th} input to the oracle should be.

Earlier we defined a mapping $z^{sk} g_i^e z^{-sk} \longmapsto R(i, e, s, k)$ taking conjugates to strings in Σ_{m-1} (in fact for a given i the mapping is to strings in Σ_i). Then the

function $f(i,k)$ was defined as an upper bound on the length of $R(i,e,s,k)$. In the algorithm the exact string $R(i,e,s,k)$ will not be used but rather a string of length exactly $f(i,k)$ that is $R(i,e,s,k)$ with 1's inserted in various places. Recall that

$$R(i,e,s,k) = [R(i,e,s,k-1)] \circ ([R(j_1,e_1,s,k-1)] \circ \ldots \circ [R(j_p,e_p,s,k-1)])$$

a concatenation of $p+1$ blocks. The first $f(i,k-1)$ bits are reserved for the first block, the next $f(i-1,k-1)$ bits are reserved for the second block, and for each of the remaining blocks $f(i-1,k-1)$ bits are reserved as well.

The most straightforward way to determine the t^{th} character of the conversion of $z^{sk}g_i^e z^{-sk}$ would be to 1) determine which block t should fall into, 2) reset t to the relative position of the character within the block, and 3) reset s, e, i, and k to the values corresponding to the particular block and loop. In particular k would be reset to $k-1$. Since k can be as large as n this is a problem: this process would take linear time. Our salvation lies in the observation that for all $0 \leq v \leq k$ the conversion of $z^{sv}g_i^e z^{-sv}$ is a prefix of the conversion of $z^{sk}g_i^e z^{-sk}$. The algorithm we will use begins by finding v such that t is in the space allocated for $z^{sv}g_i^e z^{-sv}$ but not in that for $z^{s(v-1)}g_i^e z^{-s(v-1)}$. So the character we are looking for is the t^{th} in the conversion of

$$z^{sv}g_i^e z^{-sv} = [z^{s(v-1)}g_i^e z^{-s(v-1)}] \circ [z^{s(v-1)}w_1 z^{-s(v-1)}] \circ \ldots \circ [z^{s(v-1)}w_p z^{-s(v-1)}]$$

and it must be in a block other than the first. The happy fact is that all of the other blocks are conjugates of generators with indices strictly less than i. So if we apply this process iteratively we will reach a point in which we are taking conjugates of $g_0 = 1$ and we can fill that whole block with the character 1, and this point will be reached in $\leq m = O(1)$ steps.

We now present the algorithm formally. It is split into two parts: subroutine 1b performs the conjugate conversion and the main program 1c does the computation needed before 1b is called.

Subroutine 1b: determine the t^{th} character of the conversion of $z^{sk}g_i^e z^{-sk}$

Input: $s, e \in \{\pm 1\}$, $0 \leq i \leq m$, $0 \leq k \leq n$

Output: $h \in \Sigma_{m-1} \cup \{1\}$; that is, in the end the correct t^{th} character will
; be on worktape h

1) If $t > f(i,k)$ output $h = 1$; the space beyond the end of the conversion is filled with 1's

2) If $k = 0$ or $t = 1$ then output $h = g_i^e$; the "base cases" are handled

; The next two lines determine the smallest prefix of the form $z^{sv} g_i^e z^{-sv}$ that t
; is in.

3) Invoke $Existential(k, v)$

4) If $f(i, v-1) < t \leq f(i,v)$ then continue else Reject

5) $t := t - f(i, v-1)$; t is reset to its relative position after the first block

6) $b := \lceil t / f(i-1, v-1) \rceil$; t is in the b^{th} block

7) $t := t - (b-1) f(i-1, v-1)$; t is reset to relative position within the b^{th} block

; Now suppose $w(i,e,s) = g_{j1}^{e_1} g_{j2}^{e_2} \ldots g_{jp}^{e_p}$

; Then the character we are seeking is the t^{th} character of the conversion

; of $z^{s(v-1)} g_{j_b}^{e_b} z^{-s(v-1)}$.

; Now reset variables and loop

8) $k := v - 1$

9) $e := e_b$

10) $i := j_b$

11) go to line 1

To see that this is a *ucd* program first observe that most of the steps are basic arithmetic operations on $O(\log n)$-bit numbers. The determination of $w(i,e,s)$ is just a finite table look-up. As for the loop, on each pass the value of i is decreased, so there can be at most m passes.

We now give program 1c, which basically just invokes the H oracle and then for each oracle input bit determines the proper values of i, e, s, and k for the call to subroutine 1b. There is a slight complication in that the input to the oracle is

a string of generators encoded in binary, so after the subroutine call it is necessary to make that transformation. Let c be the length of the binary codes of the generators.

Program 1c: determine whether the conjugate string $= 1$ in H.

1) Compute n in binary

2) Compute $f := cnf(m,n)$; space for n conjugates with generators coded by c ; bits each

3) Invoke $H(f,q)$

4) Compute $r := \lceil q/cf(m,n) \rceil$; oracle input q comes from the r^{th} conjugate

5) Compute $s := q-(r-1)cf(m,n)$; oracle input q is the s^{th} bit of the conversion ; of the r^{th} conjugate

6) Compute $t := \lceil s/c \rceil$; input q belongs to the t^{th} character of the conversion of ; the r^{th} conjugate

7) Compute $u := s - c(t-1)$; it is the u^{th} bit of the character code

8) Compute $k :=$ the absolute value of the net occurence of z in $x_1x_2 \ldots x_r$; see ; examples 3 and 5 of Chapter 4

9) Compute $s :=$ the sign of the net occurence of z in $x_1x_2 \ldots x_r$

10) Find i and e such that $x_r = g_i^e$; Thus the relevant conjugate is $z^{sk}g_i^e z^{-sk}$

11) Run subroutine 1b ; output $h = t^{th}$ character of the conversion

12) Accept of Reject according to whether the u^{th} bit of the binary code of h is 1 or 0 resp.

As in program 1b most steps here are $O(\log n)$-bit arithmetic operations. Lines 8 and 9 can be performed easily using the counting subroutine of chapter 4. There are no loops in the program, so the number of alternations is bounded. Each step is *ucd*, so the program is *ucd*.

That completes the proof of claim 2, and thus the proof of the theorem. □

VII.B Groups of Polynomial Growth Rate

For $G = \langle g_1, \ldots, g_m \rangle$ let $a(n) =$ the number of elements of G definable by strings of length less than or equal to n in the alphabet $\{g_1^{\pm 1}, \ldots, g_m^{\pm 1}\}$. This function a is called the growth function of G (with respect to the given set of generators). In [34] and [27] it is proved that for finitely generated solvable groups a grows either polynomially or exponentially, and that which of these two categories a group's word problem falls into is not affected by the choice of generators. Thus the finitely generated solvable groups can be divided into two classes: the polynomial growth and the exponential growth. The following result of Wolf allows us to classify the complexity of the polynomial growth solvable groups.

Theorem 7.5 (Wolf, 1968 [34]): A solvable group G has a polynomial growth function if and only if G is an extension of a finitely generated nilpotent group by a finite solvable group.

This leads to an easy extension of Theorem 7.4.

Theorem 7.6: If G is a solvable group of polynomial growth rate, then the word problem for G in in uTC^0.

Proof: By theorem 7.5 G has a nilpotent subgroup H of finite index. Lemma 2.6 tells us that there exists a subgroup N of H such that $N \lhd G$ and N is of finite index in G. Since N is a subgroup of a nilpotent group it is nilpotent as well, so by theorem 7.4 $N \in uTC^0$. Thus, by 5.2 $G \leq_{ucd} \{N, Q\}$ and since both N and Q are *ucd*-reducible to *Majority* it follows that $G \in uTC^0$. □

A result of Gromov which generalizes the Wolf-Milnor theorem allows us to classify the nonsolvable polynomial growth rate groups.

Theorem 7.7 (Gromov, 1981 [14]): If G has a polynomial growth function then G has a nilpotent subgroup of finite index.

Theorem 7.8: If G is a group with polynomial growth function then $G \in$ *Alogtime*.

Proof: By reasoning similar to that of the proof of theorem 7.6 G is an extension of a nilpotent group by a finite (nonsolvable) group. Thus by theorem 7.4 and theorem 5.2 its word problem is in *Alogtime*. □

The next theorem shows roughly that the sort of algorithm used in the proof of Theorem 7.4 will only work for polynomial growth groups.

Theorem 7.9: Let G be a finitely generated group. Suppose there exist generators $\{g_1, \ldots, g_m\}$ and a polynomial $F(k)$ such that for all $i < j$ and all k there exists a word v in the alphabet Σ_i of length $\leq F(k)$ such that $g_j^k g_i g_j^{-k} = v$. Then G is of polynomial growth rate.

Proof: By manipulating conjugates as in the proof of theorem 7.4, any word of length n can be shown equal to $g_1^{p_1} g_2^{p_2} \ldots g_m^{p_m}$ for some $p_1, p_2, \ldots p_m$ such that $|p_1| + |p_2| + \ldots + |p_m| \leq q(n)$ for a polynomial q. On the other hand the number of elements of the form $g_1^{p_1} g_2^{p_2} \ldots g_m^{p_m}$ with $|p_1| + |p_2| + \ldots + |p_m| \leq n$ is bounded by a polynomial in n, (call it $r(n)$). Thus every w of length n is equivalent to one of $r(q(n))$ elements, giving a polynomial bound on the growth rate of G.□

Theorem 7.9 shows that if G is not of polynomial growth rate, then trying to eliminate one variable at a time using conjugate collection will inevitably cause an exponential blow-up in the size of the intermediately produced strings on some inputs.

The last theorem of this chapter addresses lower bounds for polynomial growth groups. A language L is defined to be *ucd*-hard for a class X if for all $L' \in X$,

$L' \leq_{ucd} L$. This is not the same as hard with respect to some many-one reduction but it has most of the same interesting properties: e.g., if *Majority* is *ucd*-hard for *Alogtime* then $uTC^0 = Alogtime$.

Theorem 7.10: If G is nonsolvable and of polynomial growth then the word problem for G is *ucd*-complete for *Alogtime*.

Proof: Let $G/H \cong Q$ with Q finite and nonsolvable and H nilpotent. If $L \in Alogtime$ we need to find a $ucd(G)$ program that accepts $x = x_1 x_2 \ldots x_n$ if and only if $x \in L$.

Since $L \in Alogtime$ and Q is nonsolvable we know that $L \leq_{Dlogtime} Q$, (by theorem 3.2). Let $f(x)$ be the *Dlogtime* function that maps strings to strings with the property that $f(x) = 1$ in Q if and only if $x \in L$. Now let R be a set of coset representatives for G/H. If we define $f'(x)$ to be the same as $f(x)$ except with the corresponding elements of R in place of those of Q then we have $f'(x) \in H$ if and only if $x \in L$. The basic idea is to guess all the possible H-elements that $f'(x)$ could be, multiply $f'(x)$ by the inverse of that, and present this concatenation of strings to the oracle. There exists some such guess that causes the oracle to output 1 if and only if $x \in L$.

The main task then is to produce these strings of H-elements. The first thing to notice is that the number of strings needed is polynomially bounded. The reduction $f(x)$ produces a string of length $p_1(n)$ for some polynomial p_1. Should one perform conjugate collection on $f(x)$ the resulting string is increased in size by another polynomial p_2. So if $x \in L$ then $f'(x)$ is equal to an H-string of length $p_2(p_1(n))$. Since H is nilpotent it has a growth function p_3 that is polynomial. Thus $f'(x)$ could have at most $p_4 = p_3(p_2(p_1(n)))$ values. This is sufficient to show that $L \leq_{cd} G$ because the inverses of these $p_4(n)$ strings could all be separately appended to $f'(x)$ and the resulting string tested for membership in G. To see that this can be done uniformly, recall lemma 7.1 which shows that the elements of H can be put into a nice normal form $h^{q_1} h^{q_2} \ldots h^{q_m}$. For the H-elements we are interested in, the sum of the lengths

of the q_i will be $O(\log n)$. The *ucd* algorithm would existentially guess sequences $\langle q_1, q_2, \ldots, q_m \rangle$ in binary and then make a G-oracle invocation. The input to the oracle would be $f'(x)$ followed by q_1 copies of h_1, q_2 copies of h_2, etc. To determine exactly what the i^{th} input to the oracle should be is a matter of binary arithmetic, details of which are omitted.□

Chapter VIII

Polycyclic Groups

In this chapter it will be shown that the polycyclic groups, a class that contains all of the finitely generated nilpotent groups, have their word problem in TC^0. In contrast to the theorem for nilpotent groups, the circuits given here are not shown to be uniform. We begin by recalling the definition of the polycyclic groups.

Definition: A group G is polycyclic if it has a subnormal series $G = H_1 \rhd H_2 \rhd \ldots \rhd H_n = \{1\}$ such that for all i H_i/H_{i+1} is cyclic.

That the finitely generated nilpotent groups are contained in the class of polycyclic groups is shown by lemma 7.1. The group $\langle a, b; a^2, b^2 \rangle$ discussed in lemma 6.9 gives an example of a nonnilpotent solvable group with polynomial growth function. In order to assure the reader that the polycyclic groups are in fact a new class of groups we give an example of a polycyclic group with an exponential growth function.

Lemma 8.1: The group with presentation

$$\langle a, b, c; a^{-1}ba = b^2c^3, a^{-1}ca = bc^2, bc = cb \rangle$$

is polycyclic and has exponential growth function.

Proof: It will be convenient to write the element $b^p c^q$ in the form $\begin{bmatrix} p \\ q \end{bmatrix}$ because conjugating by a

$$a^{-1}b^pc^qa = (a^{-1}ba)^p(a^{-1}ca)^q = (b^2c^3)^p(bc^2)^q = b^{2p+q}c^{3p+2q}$$

has the same effect as the following matrix multiplication.

$$\begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 2p+q \\ 3p+2q \end{bmatrix}$$

A theorem of Higman, Neumann and Neumann [20] tells us that when the element a is adjoined in this manner to a previous group (the free abelian group $\langle b, c; bc = cb\rangle$ in this case) the subgroup corresponding to the previous group is unchanged. Specifically, in the case of this group it says that b and c are free generators of a free abelian subgroup, and thus for all p and q, the vectors $\begin{bmatrix} p \\ q \end{bmatrix}$ are distinct.

Claim: Let $e = e_1e_2\ldots e_n \in \{0,1\}^n$. (Call e a length n exponent vector.) For distinct e the elements $w_e = (b^{e_1}a^{-1})(b^{e_2}a^{-1})\ldots(b^{e_n}a^{-1})bca^n$ are distinct.

Pf: Note that $w_e = \begin{bmatrix} p \\ q \end{bmatrix}$ for some p and q. Let $\begin{bmatrix} p_1 \\ q_1 \end{bmatrix} < \begin{bmatrix} p_2 \\ q_2 \end{bmatrix}$ mean $p_1 < p_2$ and $q_1 < q_2$. Since all the components of the matrix $\begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}$ are positive, it is evident that a maximum vector derived from a length n exponent vector is obtained by setting $e = 11\ldots1$. Call this maximum vector m_n.

subclaim: if e is any exponent vector of length $n+1$ then $m_n < w_e$.

pf of subclaim: By induction on n.

Base case: $m_0 = bc = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. On the other hand if $|e| = 1$ then either

$$w_e = a^{-1}bca = \begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \end{bmatrix} \text{ if } e_1 = 0$$

$$\text{or } w_e = ba^{-1}bca = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix} \text{ if } e_1 = 1$$

Induction step: suppose the subclaim holds for $|e| = n$. It needs to be shown that if w_e is obtained from any exponent vector e of length n then $b^0a^{-1}w_ea$ and $ba^{-1}w_ea$ are both greater than m_n. (That they are distinct from each other is obvious.)

By induction hypothesis $w_n > m_{n-1}$, so $a^{-1}w_na$ exceeds $a^{-1}m_{n-1}a$ by at least 2 in each component. (This is easily seen from inspection of the matrix.) Thus $b^0a^{-1}w_na$ and $ba^{-1}w_na$ exceed $ba^{-1}m_{n-1}a = m_n$ by at least 1 in each component. *q.e.d. subclaim/claim*

Thus there are at least 2^n distinct elements representable by strings of length $\leq 3n + 2$. □

Several interesting characterizations of the polycyclic groups exist, (see [34] for a concise summary). One that will be of use is given in the following theorems.

Theorem 8.2 (Auslander, 1967 [2]): A solvable group G is polycyclic if and only if it is isomorphic to an integer matrix group.

The Auslander characterization of polycyclic groups allows us to conclude the following immediately from the Lipton–Zalcstein theorem (6.4).

Theorem 8.3: If G is polycyclic then $G \in$ *Dlogspace.*

Lemma 8.4: If G is polycyclic with chain length m and $H \lhd G$ then G/H is polycyclic with chain length $\leq m$.

Proof: This follows straightforwardly from the third isomorphism theorem, (see Rotman [29] pp. 24–27). To give a more combinatorial proof, we can view the statement that G is polycyclic with chain length $\leq n$ as equivalent to saying that G has a presentation with certain sorts of relations. For example, if H_1/H_2 is generated by element a then relations such as $a^{-1}ya = R(\vec{x})$ should hold, where y is any string, and $R(\vec{x})$ is a string representing an element of H_2. On the other hand we can consider G/H to be G with more relators added to its presentation (namely the infinite set of strings $\{w : w \in H\}$). Thus the same set of relations that show G to be polycyclic with chain length $\leq m$ show the same for G/H. □

Theorem 8.5: If G is a finitely generated polycyclic group then the word problem for G is in TC^0.

Proof: Assume G and the H_i are as given in the definition of polycyclic group.

It will be convenient to use the characterization from theorem 8.2 and assume that the generators are given as $k \times k$ integer matrices.

Let the input be a string of G-generator matrices $A_1 A_2 \ldots A_n$ and let A be their product. From the proof of theorem 6.4 we know that $A = 1$ (the identity matrix) if and only if $A \cong 1 \bmod p$ for $p = 2, 3, \ldots, f(n)$ for some polynomial f. The circuit consists of the conjunction of these predicates. For a fixed p the usual collection process will be used. Note that by lemma 8.4 $G \bmod p$ (or more precisely G/G_p where G_p is the elements of G equal to 1 mod p) is polycyclic of chain length m. Thus for each i we can translate a string of $H_i \bmod p$ matrices into an equivalent string of $H_{i+1} \bmod p$ matrices (if such a string exists; if not, then set a "reject" flag).

It will be sufficient to show that a string of $H_1 \bmod p$ matrices (not necessarily from a restricted set of generators) can be transformed into an equivalent sequence of $H_2 \bmod p$ matrices (or a reject flag is set) in constant depth and polynomial size. (This is sufficient because the process of converting from H_2 to H_3, H_3 to H_4, etc. is the same.) It is important to note that we need the same constant depth to work for all p simultaneously.

Given: n $k \times k$ integer matrices $A_1 A_2 \ldots A_n$ with entries in the range $[0, p)$ and all in the group $H_1 \bmod p$. Let A denote their product mod p.

Let a represent a generator of $(H_1 \bmod p)/(H_2 \bmod p)$.

Step 1: translate each A_i into $a^{q_i} B_i$ for some integer q_i encoded in binary and some $B_i \in H_2 \bmod p$.

Step 2: for $j = 1, 2, \ldots, n+1$ compute $r_j = \Sigma_{i=1}^{j} q_i$.

Step 3: if $a^{r_{n+1}} \notin H_2 \bmod p$ then set the reject flag.

Step 4: Replace the conjugates and the leftover portion ($a^{r_{n+1}}$) with equivalent matrices $C_1 C_2 \ldots C_{n+1}$ with entries in the appropriate range. Each C_i is in $H_2 \bmod p$ as desired (assuming the reject flag did not have to be set, in which case it doesn't matter).

Before analyzing this for size and depth we need to observe that the size of the largest finite group dealt with is polynomially bounded. A group of $k \times k$ (mod

p)-matrices can have at most p^{k^2} elements. Since p is bounded by a polynomial in n and since k is fixed the group is polynomially bounded. We have then that each matrix encountered here will be represented by $O(\log n)$ bits, and that the length of the q_i found are likewise bounded. Now to analyze the algorithm:

Step 1 just transforms $O(\log n)$ bit blocks into $O(\log n)$ bit blocks. This can be done with a depth 3 DNF circuit of polynomial size.

Step 2 is $n+1$ iterated addition circuits set up in parallel. Since each addend is $O(n)$ in size, (in fact $O(\log n)$), corollary 2.4 says this can be computed with constant depth polynomial size threshhold circuits.

Steps 3 and 4 both deal with only $O(\log n)$ bits at a time and so can be done by depth 3 DNF circuits.

Putting all this together, each mod-p subcircuit is polynomial size and constant depth. By stacking m such circuits on top of each other the input can step by step be translated into the empty string. The reject flags of each level are linked by an unbounded fanin OR. This is then negated giving us an output of 1 if and only if the input is equal to 1 mod p. There are polynomially many of these mod subcircuits and thus they can be linked by one unbounded fanin AND, adding 1 to the depth.

An additional bit of preprocessing is needed: the input matrices have to be reduced mod p before this process can start. This is a matter of transforming $O(1)$ length strings to $O(1)$ length strings (in almost all cases this will be the identity transformation) and does not present a problem. □

A class of groups similar in definition to the polycyclic are the M-groups.

Definition: A group is an M-group if it has a subnormal series $G = H_1 \rhd H_2 \rhd \ldots \rhd H_m = \{1\}$ such that for all i H_i/H_{i+1} is either infinite cyclic or finite.

Lemma 8.6: If G is an M-group then G is an extension of a polycyclic group by a finite group.

Proof: See Scott [32], p. 153. □

Theorem 8.7: If G is an M-group then $G \in NC^1$.

Proof: By theorem 5.2 and lemma 8.6 $G \leq_{ucd} \{H, Q\}$ for some polycyclic group H and some finite group Q. We can simply view the *ucd*-reduction as a regular constant depth reduction, and then the fact that the word problems for H and Q are in NC^1 gives the theorem.□

Chapter IX

Other Groups

The previous two chapters constitute the main part of the thesis. This chapter contains miscellaneous results that point in the direction of extending the main results, and provide a starting point for continued research.

The first theorem shows that the polycyclic groups are not the only groups in TC^0. The proof will make use of a classic theorem due to Dehn, the proof of which is quite long and can be found in [26].

Definition: If a word is freely reduced and its first and last characters are not inverses of each other, then the word is called cyclically reduced.

Theorem 9.1 (The Freiheitsatz): Let $R(a_1, a_2, \ldots, a_n)$ be a cyclically reduced word in $a_1, a_2, \ldots, a_n$ which involves a_n. Then the subgroup of

$$G = \langle a_1, a_2, \ldots, a_n; R(a_1, a_2, \ldots, a_n) \rangle$$

generated by $a_1, a_2, \ldots, a_{n-1}$ is freely generated by them; in other words, every non-trivial relator of G must involve a_n.

Also, a characterization of the polycyclic groups due to Hirsch will be needed.

Definition: A group G satisfies the maximal condition if it has no infinite ascending chain of subgroups $H_1 \subset H_2 \subset \ldots$.

Theorem 9.2 (Hirsch, 1938 [21]): If G is solvable then G is polycyclic if and only if it satisfies the maximal condition.

Theorem 9.3: The group with presentation $G = \langle a, b : b^{-1}ab = a^2 \rangle$ is solvable, but not polycyclic, and its word problem is in TC^0.

Proof: We first establish that G is solvable.

Claim 1: $G^{(1)} \subseteq \{b^n a^k b^{-n} : n \geq 0, k \in \mathbb{Z}\}$.

Pf: Every element of $G^{(1)}$ is a product of commutators and thus has a representation in which the net number of occurences of b, (b counting positively and b^{-1} counting negatively), is 0. The relation $b^{-1}ab = a^2$ implies the relations $ab = ba^2, a^{-1}b = ba^{-2}, b^{-1}a = a^2b^{-1}$, and $b^{-1}a^{-1} = a^{-2}b^{-1}$. Using these and the trivial relation $b^{-1}b = bb^{-1}$ we can push all occurences of b to the left and all occurences of b^{-1} to the right without changing the net occurences of b.

Claim 2: $G^{(2)} = \{1\}$.

Pf: It will be shown that two arbitrary elements of $G^{(1)}$ commute. Let $n \geq 0$. Then

$$\begin{aligned}(b^{n_1}a^{k_1}b^{-n_1})(b^{n_2}a^{k_2}b^{-n_2}) &= b^{n_1}(a^{k_1}b^{n_2})(b^{-n_1}a^{k_2})b^{-n_2} \\ &= b^{n_1}(b^{n_2}a^{k_1\cdot 2^{n_2}})(a^{k_2\cdot 2^{n_1}}b^{-n_1})b^{-n_2} \\ &= b^{n_1+n_2}a^{k_1\cdot 2^{n_2}+k_2\cdot 2^{n_1}}b^{-(n_1+n_2)}\end{aligned}$$

The result here is clearly the same as if the two factors were reversed.

Thus G is solvable. The next claim establishes that G is not polycyclic.

Claim 3: Let $H_i = \{b^n a^k b^{-n} : 0 \leq n \leq i, k \in \mathbb{Z}\}$. Then for all $i \geq 0$:

(a) H_i is a subgroup of G,

(b) $H_i \subset H_{i+1}$.

Pf: (a) The identity element is in H_i: $b^0a^0b^{-0} = 1$. To establish that each

element has an inverse: $(b^n a^k b^{-n})(b^n a^{-k} b^{-n}) = 1$. Lastly, to show that H_i is closed under multiplication, let $b^{n_1} a^{k_1} b^{-n_1}$ and $b^{n_2} a^{k_2} b^{-n_2}$ be arbitrary elements of H_i, (so $0 \leq n_1, n_2 \leq i$).

Case 1: $n_1 \leq n_2$. Then

$$\begin{aligned}(b^{n_1} a^{k_1} b^{-n_1})(b^{n_2} a^{k_2} b^{-n_2}) &= b^{n_1}(a^{k_1} b^{n_2-n_1}) a^{k_2} b^{-n_2} \\ &= b^{n_1}(b^{n_2-n_1} a^{k_1 \cdot 2^{n_2-n_1}}) a^{k_2} b^{-n_2} \\ &= b^{n_2} a^{k_2 + k_1 \cdot 2^{n_2-n_1}} b^{-n_2}\end{aligned}$$

Case 2: $n_1 > n_2$. Then

$$\begin{aligned}(b^{n_1} a^{k_1} b^{-n_1})(b^{n_2} a^{k_2} b^{-n_2}) &= b^{n_1} a^{k_1}(b^{n_2-n_1} a^{k_2}) b^{-n_2} \\ &= b^{n_1} a^{k_1}(a^{k_2 \cdot 2^{n_1-n_2}} b^{n_2-n_1}) b^{-n_2} \\ &= b^{n_1} a^{k_1 + k_2 \cdot 2^{n_1-n_2}} b^{-n_1}\end{aligned}$$

Since both n_1 and n_2 are less than i, in either case the result is in H_i.

(b) It will be shown that $b^{i+1} a b^{-(i+1)}$ is not in H_i. If it were then it would have to have an inverse $b^n a^k b^{-n} \in H_i$. But as in claim 2:

$$(b^{i+1} a b^{-(i+1)})(b^n a^k b^{-n}) = b^{n+i+1} a^{2^n + k \cdot 2^{i+1}} b^{-(n+i+1)}$$

Making use of claim 6 ahead, the latter is the identity if and only if $2^n + k \cdot 2^{i+1} = 0$. But since $n < i + 1$ this equation has no solution for k an integer.

Thus G does not have the maximal condition and by theorem 9.2 is not polycyclic.

Next we show that $G \in TC^0$. Let $w \in \{a^{\pm 1}, b^{\pm 1}\}^*$.

Claim 4: if $w = 1$ then the number of occurences of b is 0.

Pf: Going back to the definition of a group presentation we have that $w = 1$ if and only if we can obtain the empty string by insertion and deletion of $b^{-1}aba^{-2}$ and the trivial relators aa^{-1}, bb^{-1}, etc. None of these operations change the net occurences of b so if we are to possibly transform w into the empty string (which has no occurences of b, of course), w must start with a net of 0 b's.

Claim 5: If w has 0 net occurences of b then $w = b^n a^k b^{-n}$ for some $n \geq 0$.

Pf: as explained in claim 1.

Claim 6: $b^n a^k b^{-n} = 1$ if and only if $k = 0$.

Pf:

$$\begin{aligned} b^n a^k b^{-n} = 1 &\Rightarrow b^{-n}(b^n a^k b^{-n})b^n = b^{-n}(1)b^n = 1 \\ &\Rightarrow a^k = 1 \\ &\Rightarrow k = 0 \text{ by the Freiheitsatz} \end{aligned}$$

The algorithm to decide the word problem for G will 1) check that the net occurences of b is 0, and 2) if so, put the string into the form $b^n a^k b^{-n}$, $n \geq 0$ and test for $k = 0$.

Claim 7: Let $g = g_1 g_2 \ldots g_n \in \{a^{\pm 1}, b^{\pm 1}\}^*$. Then $g = b^{r_0} a^k b^{-s_0}$ where

r_i = the total # of occurences of b^{+1} to the right of g_i.

s_i = the total # of occurences of b^{-1} to the left of g_i.

$k = \Sigma_{i=1}^{n} e_i p_i q_i$ where

$$p_i = \begin{cases} 2^{r_i} & \text{if } g_i = a^{\pm 1} \\ 0 & \text{if } g_i = b^{\pm 1} \end{cases}$$

$$q_i = \begin{cases} 2^{s_i} & \text{if } g_i = a^{\pm 1} \\ 0 & \text{if } g_i = b^{\pm 1} \end{cases}$$

$$e_i = \begin{cases} +1 & \text{if } g_i = a^{+1} \\ -1 & \text{if } g_i = a^{-1} \end{cases}$$

Proof: As in claim 1, all occurences of b^{+1} will be pushed to the left and all occurences of b^{-1} will be pushed to the right.

We first deal with the b^{+1}. Whenever b^{+1} is pushed past an $a^{\pm 1}$, the $a^{\pm 1}$ doubles. (I.e., ab is replaced by ba^2 and $a^{-1}b$ is replaced by ba^{-2}.) Thus after all the b^{+1} are pushed, we have b^{r_0} on the far left, each $g_i = a^{e_i}$ is replaced by $a^{e_i p_i}$, and the b^{-1} are still in place. Similarly, when b^{-1} is pushed to the right past an $a^{\pm 1}$,

the $a^{\pm 1}$ doubles. Since $b^{-1}a^{e_i p_i}$ is equal to $a^{2e_i p_i}b^{-1}$, ultimately $a^{e_i p_i}$ is replaced by $a^{e_i p_i 2^{s_i}} = a^{e_i p_i q_i}$, and we have b^{-s_0} at the far right.

Claim 8: $G \in TC^0$.

Pf: We only need to be able to compute the above k. The r_i and s_i are obtained by counting. Since these will be $O(\log n)$ in length, the p_i and q_i can be obtained via a look-up table. Finally, as p_i and q_i are $O(n)$ in length, k can be computed using multiplication and vector addition circuits. As stated previously, $g \in G$ if and only if $k = 0$. □

Lastly, we mention some recent results on hyperbolic groups, (see [15] or [13] for a general exposition on the topic).

Hyperbolic groups are defined by viewing a group as a metric space. Let G be a group generated by a finite set Σ, and assume that for all s in Σ, s^{-1} is also in Σ. For $g \in G$ define

$$\|g\| = \text{the length of the shortest word over } \Sigma \text{ that defines } g$$

Then we define the distance between two elements g and h by

$$d(g,h) = \|g^{-1}h\|$$

A path between two elements g and h is a sequence of group elements $g_0, g_1, \ldots g_m$ such that $g_0 = g$, $g_m = h$ and for all i, $g_{i+1} = g_i s$ for some $s \in \Sigma$. A path is a geodesic between g and h if there exists no shorter path linking these two points. The length of such a geodesic is necessarily $d(g,h)$.

For $\delta \geq 0$, a δ-neighborhood of a path is the set of all group elements within distance δ of some point on the path.

Definition: A group G is hyperbolic if there exists a $\delta > 0$ such that for any three group elements g_1, g_2 and g_3 and any three geodesics p_{12}, p_{13} and p_{23} linking them, each path is contained in the union of the δ-neighborhoods of the other two.

Examples include free groups, context free groups and the small cancellation groups (see [13], Appendix). Examples of nonhyperbolic groups include any group with a free abelian subgroup of rank 2, (i.e., $\mathbb{Z} \times \mathbb{Z}$).

Theorem 9.4 (Cai, 1992 [9]): The word problem for any finitely generated hyperbolic group is in NC^2.

In looking to extend Cai's result, a natural next class to examine would be the Markov groups. That class contains the hyperbolic groups and the polycyclic groups (and thus $\mathbb{Z} \times \mathbb{Z}$) and is closed under extensions and free products ([13], Ch. 9). The Markov groups are also interesting because their definition relates to formal language theory.

Definition: A Markov grammar Γ is a Deterministic Finite Automaton (see [22]) with the restriction that no arrow points from a nonaccepting state to an accepting state. The language accepted by Γ in denoted by $L(\Gamma)$.

Definition: A group G is a Markov group if there exists a set of generators Σ for G and a Markov grammar Γ with alphabet Σ such that the natural map from $L(\Gamma)$ to G is a bijection.

Thus $L(\Gamma)$ is a set of unique normal forms for G with respect to Σ. It is not hard to see that a free group is Markov: using a set of free generators and their inverses as the alphabet, the DFA needs only to assure that no symbol is followed immediately by its inverse. It is equally easy to see that polycyclic groups are Markov by using the fact that each element can be written uniquely in the form $x_1^{e_1} x_2^{e_2} \ldots x_m^{e_m}$ for some set of generators $\{x_1, x_2, \ldots, x_m\}$.

The Markov groups then provide a candidate for a larger class to show to be in NC.

Chapter X

Conclusion

Many interesting open problems remain to be looked at in this area. Some, such as determining exactly which groups have their word problem in TC^0, entail solving major open questions in circuit theory. In the case of that example, just determining which finite groups are in TC^0 would answer the question of whether $TC^0 = NC^1$ since the finite nonsolvable groups are complete for NC^1. But many good problems without such large-scale implications exist as well.

Many of the results of this dissertation leave room for improvement. Some feasible-looking questions are:

1) Are the free group word problems in $Alogtime/NC^1$?

2) Are the polycyclic word problems in uTC^0?

3) Are the polynomial growth nonsolvable groups complete for *Alogtime* with respect to *Dlogtime* reductions?

4) Is the word problem for an extension of a group H by an infinite cyclic group uTC^0-reducible to the word problem for H?

5) Can the free products theorem be improved? A possible conjecture is that $G \otimes H \leq_{ucd} \{G, H, F\}$ for F a nonabelian free group.

6) Can the Markov groups be shown to be in NC or hard for P?

Another general question that could be explored is which groups' word problems are hard for NC^1. One might suspect that all nonsolvable groups word problems

would be hard for NC^1 since they seem "harder" than the finite nonsolvable group word problems. Of course, it is possible that a group's word problem could be neither in NC^1 nor hard for NC^1. A theorem of J. Tits [33] may be helpful here; it states that a nonsolvable linear group over a field of characteristic 0 either has a nonabelian free subgroup or is an extension of a solvable group by a finite nonsolvable group.

Most of the groups looked at here have been such that they can be built up from less complex groups by extensions, and the algorithms have been based on this structure. In [19] and [31] examples of an infinite simple groups are given, (groups with no nontrivial normal subgroup). Approaching such groups would require techniques different from any employed here.

Bibliography

[1] A. V. Anisimov. On group languages. *Cybernetics*, 7:594–601, 1971.

[2] Louis Auslander. On a problem of Philip Hall. *Annals of Math*, 86:112–116, 1967.

[3] David A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1986.

[4] David A. Mix Barrington and James Corbett. On the relative complexity of some languages in NC^1. *Information Processing Letters*, 32:251–256, 1989.

[5] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41:274–306, 1990.

[6] Paul W. Beame, Stephen A. Cook, and H. James Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15:994–1003, 1986.

[7] W. W. Boone. Certain simple unsolvable problems of group theory. *Indig. Math.*, 16,17 and 19, 1955.

[8] Samuel R. Buss. The Boolean formula value problem is in ALOGTIME. *19th ACM STOC Symp.*, pages 123–131, 1987.

[9] Jin-yi Cai. Parallel computation over hyperbolic groups. *24th Annual ACM Symposium on the Theory of Computing*, pages 106–115, 1992.

[10] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, pages 114–133, 1981.

[11] Ashok K. Chandra, Larry Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13:423–439, 1984.

[12] M.J. Dunwoody. The accessibility of finitely presented groups. *Inventiones Mathematicae*, 81:449–457, 1985.

[13] E. Ghys and P. de la Harpe, editors. *Sur les Groupes Hyperboliques d'apres Mikhael Gromov.* Birkhauser, 1990.

[14] M. Gromov. Groups of polynomial growth and expanding maps. *Publ. Math. I.H.E.S.*, 53:53–73, 1981.

[15] M. Gromov. Hyperbolic groups. In S.M. Gersten, editor, *Essays in Group Theory*. Springer–Verlag, 1987.

[16] Philip Hall. Some word problems. *J. London Math. Society*, 33:482–496, 1958.

[17] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, London, 4th edition, 1959.

[18] O. G. Harlampovič. A finitely presented solvable group with unsolvable word problem. *Math. USSR Izvestija*, 19:151–169, 1982.

[19] Graham Higman. A finitely generated infinite simple group. *Journal of the London Mathematical Society*, 26:61–64, 1951.

[20] Graham Higman, B. H. Neumann, and Hanna Neumann. Embedding theorems for groups. *J. London Math. Society*, 24:247–254, 1949.

[21] K. A. Hirsch. On infinite soluble groups, I. *Proc. London Math. Soc.*, 44:53–60, 1938.

[22] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Inc., 1979.

[23] Oscar H. Ibarra, Tao Jiang, and Bala Ravikumar. Some subclasses of context-free languages in NC^1. *Information Processing Letters*, 29:111–117, 1988.

[24] Richard J. Lipton. Model theoretic aspects of computational complexity. *IEEE* 19^{th} *Annual Symposium on Foundations of Computer Science*, pages 193–200, 1978.

[25] Richard J. Lipton and Yechezkel Zalcstein. Word problems solvable in logspace. *Journal of the ACM*, 24:522–526, 1977.

[26] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial Group Theory: presentations of groups in terms of generators and relations.* Dover Publications, Inc., 1976.

[27] John Milnor. Growth of finitely generated solvable groups. *Journal of Differential Geometry*, 2:447–449, 1968.

[28] David E. Muller and Paul E. Schupp. Groups, the theory of ends, and context-free languages. *Journal of Computer and System Sciences*, 26:295–310, 1983.

[29] Joseph Rotman. *Theory of Groups.* Allyn and Bacon, Inc., 1965.

[30] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.

[31] E. A. Scott. A tour around finitely presented infinite simple groups. In G. Baumslag and C. F. Miller III, editors, *Essays in Group Theory*. Springer–Verlag, 1987.

[32] W. E. Scott. *Group Theory*. Prentice-Hall, Inc., 1964.

[33] Jacques Tits. Free subgroups in linear groups. *Journal of Algebra*, 20:250–270, 1972.

[34] Joseph A. Wolf. Growth of finitely generated solvable groups and curvature of riemannian manifolds. *Journal of Differential Geometry*, 2:421–446, 1968.