INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106

Najarian, John Panos

INVESTIGATIONS OF BRAID GROUP ALGORITHMS

City University of New York PH.D. 1985

INVESTIGATIONS OF BRAID GROUP ALGORITHMS

by

John P. Najarian

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, the City University of New York

1985

This manuscript has been read and accepted for the Graduate Faculty of Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

4/29/85 — Michael Anshel

date — Chairman of the Examining Committee

4/30/85 — Paul R Kormanik

date — Executive Officer

Professor Michael Anshel, Mentor

Professor Leon F. Landovitz

Professor John Moyne

Professor Kenneth McAloon

Professor Peter Stebe

Supervisory Committee

City University of New York

Abstract

INVESTIGATIONS OF BRAID GROUP ALGORITHMS

by

John P. Najarian

Adviser: Professor Michael Anshel

The classical Artin algorithm for the word problem for braid groups is shown to have an exponential space and exponential time worst case. Monte Carlo experiments and enumerations show that the average case is non-linear (but appears to be a low order polynomial) for Artin's algorithm. Trends and patterns are analyzed.

Garside's algorithm for the word problem is then analyzed with respect to average and worst cases. Statistical evidence shows that the classical Garside algorithm has exponential space growth with respect to the number of braid strands. For braids with more than six strands, this algorithm easily exceeds the storage capacity of the present generation of mainframe computers. Many interesting properties are proven about the word diagrams of Garside's algorithm.

A variant of Garside's algorithm is proven to operate in non-deterministic linear space on a Turing Machine in the average and worst cases. In deterministic form, this algorithm requires at most quadratic space on a Turing Machine.

The Burau representation is used to construct other algorithms. A complexity-theoretic line of attack for the famous faithfulness conjecture of the Burau representation is demonstrated. The word problem for B(3) is proven to require logspace.

A variant of the Luginbuhl combing algorithm is designed for solving the word problem for braid groups. A brief analysis of it follows.

A new algorithm is designed for nearly solving the word problem for braid groups. This algorithm operates in log-space for all B(n) but it accepts some rare non-identity braids. Some counterexamples are shown.

The word problem for B(3) can be solved in linear time but with linear space usage.

Acknowledgements

I wish to express my appreciation to the Computer Science Faculty of City College and the Graduate School for my education.

Foremost recognition belongs to Professor Michael Anshel for the careful reading, criticism, and supervision of this thesis. Before the thesis, his profound understanding, analytic lectures, and elucidating conversations provided an atmosphere of inquiry and research. I only hope I could retain a fraction of it. In that respect, thanks go to Prof. William Gewirtz for introducing me to the first half of Computer Science.

I am greatly indebted to Professor Leon Landovitz for the support, guidance, and concern he has shown in these last few years.

Professors John Moyne, Ken McAloon, Peter Stebe, Daniel Gandel, and Jack Fenichel deserve my thanks in many respects, both for education provided and criticisms given.

I wish also to thank my brother for the long hours he waited to have supper with me. Finally, I wish to thank God for providing all our needs out of his abundant generosity.

This thesis had one casualty; a dear cousin of only half my age, named Lisa. She waited for its completion so I could visit her. A few months ago, divine providence saw it fit that I should be one of her pall bearers. To her, I dedicate the theorem bearing her name.

CONTENTS

I.0 Preliminaries

I.A Group Theoretic Concepts

Here we define a group in terms of generators and defining relators. A presentation P is a pair <A ; R> such that:

A is a set of generating symbols a[i] and their uniquely associated inverses (denoted a[i]'.)

R is a set of finite words (called defining relators) formed by elements of A.

The inverse of a word w=a[1]a[2]....a[i]..a[k] (where a[i] is an element of A) is a word w'=a[k]'...a[i]'...a[2]'a[1]'. We observe that a[i]''=a[i]. Implicitly assumed in R (but rarely written) are a set of relators g'g and gg' (called free relators), for every generator g. Under a presentation P and given two words u and v (from alphabet A), we say

Preliminaries

u is reducible to v iff u can be transformed into v after a finite sequence of insertions or deletions of relators and their inverses. The group G being represented has as its elements, the equivalence classes of words (from alphabet A) under the reducibility equivalence relation. Concatenation of representatives is the group operation. A complete discussion of groups expressed in terms of generators and defining relators can be found in Magnus, Karrass, and Solitar[1966].

The algebraic braid group was introduced by Artin[1925]. The braid group B(n+1) is the group corresponding to the presentation <s(1),s(2),...,s(n) ;

$$s(i)'s(i) = s(i)s(i)' = 1$$
$$s(i)s(j) = s(j)s(i) \quad \text{iff} \quad |i-j| > 1$$
$$s(i)s(i+1)s(i) = s(i+1)s(i)s(i+1) \quad \text{for all } i,j. \quad >$$

For alternate interpretations of B(n+1) as a topological structure, refer to Birman[1975], Magnus[1974], Stillwell[1980], or Moran[1984].

Preliminaries

A group of whose presentation has only free relators and n generating symbols is called a free group of rank n (denoted F(n).)

The fundamental algorithmic problem of group theory, the word problem, was defined and investigated by Dehn[1911].

The word problem for a group G (denoted WP(G)), given in terms of generators and defining relators, is the problem of determining if a word w in the generators can be reduced (ie. transformed by insertion and deletion of relators and their inverses) to the identity word. The set of such words which reduce to the identity word is called the group language of G (denoted GLA(G)) under that presentation.

It will be convenient to describe the group language of the free abelian group on one generator as LEQ, which is the language composed of all words w (from alphabet {0,1}) such that the number of zeros equals the number of ones.)

We remark that GLA(F(n)) have been investigated as the two-sided Dyck languages on n letters (ref. Harrison[1978], page 312-325.)

By the growth rate of a language, we mean the function f(m) whose value denotes the number of words of length m which are reducible to the identity element (under the given presentation.)

I.B Turing Machines and Complexity Concepts

The Turing Machine serves as a useful model of computation in that it use allows us to measure space and time utilization (ie. the computational complexity) of algorithms and, more abstractly, of problems. Storage space in our model of Turing Machines consists of a read-only input tape (containing the input data), an output (generally 2-valued), and a number of work tapes (for intermediate computations.) Each tape can only be accessed through a head which can read and write from one cell on that tape at any moment. In that moment, the head can move left one cell or move right one cell or stand still. This so far describes the storage and access method; the exact

Preliminaries

formal control system of a Turing Machine is described below.

A Turing Machine T is an octuple

T=<Q,I,W,O,D,NS,q0,qF> such that:

i. Q is a set of states

ii. I is a set of input symbols

iii. W is a set of work tape symbols

iv. O is a set of output symbols {T,F}

v. D is a set of directions in which to move a head {L,R,S}

vi. NS is a next move function

NS: QxIxW --> QxDxWxD

vii. q0 in Q is the start state

viii. qF is the final state

Preliminaries

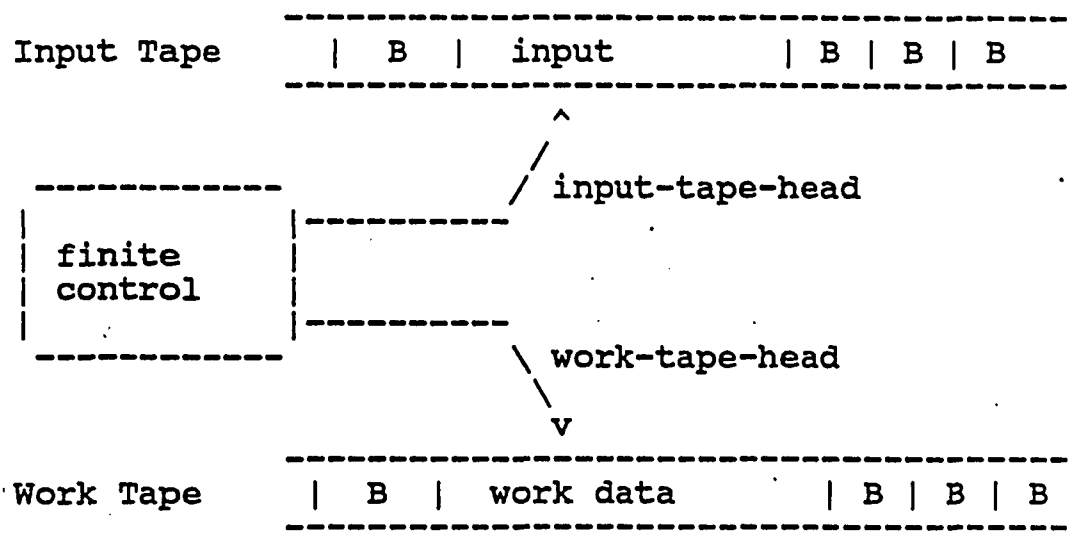
By the finite control of a Turing Machine T, we mean the function NS(q,i,w) which characterizes the rules of computation specific to T, expressing the algorithm by which T will operate.) The finite control can not be modified at execution time; likewise, it is generally not treated as a computational resource.

The Turing Machine starts in state q0, with input-tape-head pointing to the left boundary of the input, which is on the input tape. The work tape generally starts with blank symbols (denoted B) and the work-tape-head points to it.

The NS function (based on those of Hopcroft and Ullman[1968]) is generally described as a set of function values. For instance, NS(q,i,w)=<q2,di,w2,dw> signifies: if the Turing Machine is in state q and the input-tape-head points to symbol i (on the input tape) and the work-tape-head points to symbol w, then have the Turing Machine enter state q2, with i2, move input-tape-head one cell in direction di, replace the value of the work cell (ie. the cell pointed to by the work-tape-head) with w2, and move work-tape-head one cell in direction dw. This process is repeated until

the machine enters a final state. In a step before entering the final state, the output (ie. T or F) is placed on the work tape. T stands for true, F for false, L for "move left", R for "move right", and S for "stay still". The work and tape heads move independently.

Below is a diagram of this machine model:



An example of a Turing Machine for determining if string 1**k has even length is:

TPAR=<{q0,q1,qF}, {1,B}, {B}, {T,F}, {L,R,S}, NS,q0,qF>

such that function NS is defined by:

NS(q0, B , B) = < qF , S , T , S >
NS(q0, 1 , B) = < q1 , R , B , S >
NS(q1, B , B) = < qF , S , F , S >
NS(q1, 1 , B) = < q0 , R , B , S >

Preliminaries

Occasionally, we will use a natural programming language for Turing machines in the manner of Domanski[1982], as well as the octuple model given.

An alternate model of this Turing machine is the two-input-head Turing Machine. Formally, a two-input-head Turing machine is a Turing Machine T2 with the following octuple structure:

T2=<Q,I2,W,O,D,NS2,q0,qF> such that:

I2 = IxI where:

I is a set of input symbols

NS2 is a next move function
NS2: QxI2xW --> QxD2xWxD

D2 = DxD where:

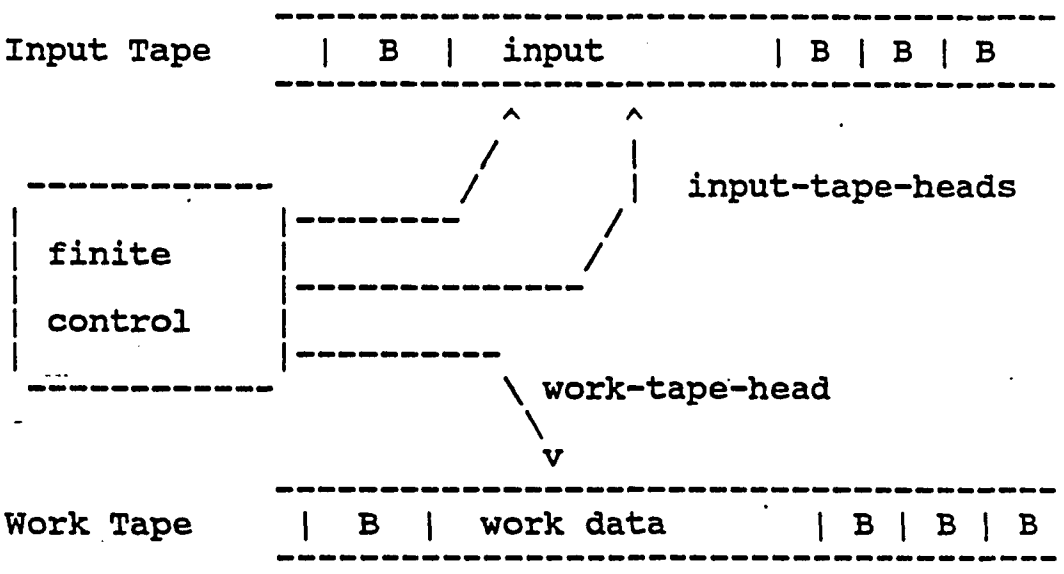
D is a set of directions in which to move a head {L,R,S}

and all other terms correspond to those of the single input head model.

The main difference between T2 and T is that T2 has two independently moving input heads. Each execution of the next move function NS2 is determined by the machine state, the work tape cell, and the two input cells being scan by the two input heads (ie. corresponds to I2.) The execution of the NS2 function

then results in the motion of both input heads in independent directions (ie. D2's role), as well as the standard Turing Machine operation on the work tape and next state.

Below is a diagram of the two input head machine model:



Turing Machines which restrict input tape heads to move only in one direction are called on-line; otherwise, they are called off-line. For example, Turing Machine TPAR is on-line. Since input tapes can only be read, the space utilization of a machine is determined by the amount of work tape (ie. number of

work tape cells) used during the execution of that algorithm. Time utilization can respectively be defined as the amount of time required (expressed as the number of instructions executed or moments spent) to complete the execution of that algorithm. Both space and time complexity measures can be expressed as functions of the input length (or more broadly, the problem instance.) Furthermore, both measures can be qualified in terms of average utilization (ie. the resource utilization of an arbitrary case) and worst case utilization (ie. the case of maximal resource use.) Problems and algorithms are called f(m)-space (or f(m)-time) solvable iff their respective complexity measures can be expressed as (or bounded by) a linear function of f(m) where m is the length of the input (or problem instance.) For example, a problem is log-space solvable iff its space complexity is k*log(m) where k is a constant. Turing machines which exactly require time complexity m are said to operated in real time. Other types of Turing Machines and their complexity aspects are covered in Harrison[1978].

Preliminaries

The notation in this section was conceived out of the graphic constraints of the computer used. This has resulted in cumbersome adaptation for notation. x**y denotes exponential function x raised to the power y.

I.C Bounds on the Complexity of the Word Problem for Braid Groups

Sometimes properties can be proven about groups directly from the structure of the relators. In the case of the braid group B(n+1), the following theorems arise in such a manner.

Proposition 1: If w is a word in the braid group language GLA(B(n+1)), then the number of occurrences of inverse generators in w must be equal to the number of occurrences of positive generators in w.

Proof: All relators are of one of the following forms:

a'a , aa' , abab'a'b' , a'c'ac .

Starting with the trivial word (which obviously has as many inverse generators as positive generators), any insertion or deletion of relators will preserve the equality in number.

(QED)

Proposition 2: If w is a word in the braid group language GLA(B(n+1)), then w must be of even length.

Proof: By proposition 1, for a word to be trivial, it must have k inverted generators if it has k positive generators. So it has length 2k, hence even.

As an alternate proof, note that the relators are all of even length. Any insertion or deletion of them will conserve parity.

(QED)

Proposition 2 may seem weak relative to the Proposition 1; however, it is extremely useful in trivial-braid enumerations because it can skip a whole length category in the enumeration. Proposition 1 can't cause such clean jumps (unless much programming machinery is added, which will still prove very costly (in time complexity) without this proposition as a first programming construct.)

The recognition problem for LEQ is the problem of determining if an arbitrary word (composed of 0's and 1's) is in the language LEQ.

Preliminaries

Lemma 1: The recognition problem for LEQ:

i. is solvable on an on-line Turing Machine.

ii. requires at least log-space in (the worst case) on an on-line single-input-head Turing Machine.

Proof: First, we need to show that a single head on-line Turing Machine can solve this problem in log-space. The proof is by construction.

```
-------------------------------------------------
/* TM-input-head starts at leftmost on input tape */
/* The work tape will be treated as a COUNTER.    */

Put a '+' on the work tape as initial COUNTER ;
Loop Until ( TM-input-head points to right-end) ;
    If (TM-input-head points to '1')
    Then CALL INCREMENT( COUNTER) ;  /* by 1 */
    Else CALL DECREMENT( COUNTER) ;  /* by 1 */
   Move TM-input-head to right one step;
End Loop;

If (COUNTER = 0)
Then Print 'Word is in LEQ.';
Else Print 'Word is not in LEQ.';
```

```
Halt ;
```

For sake of completeness, the Increment and Decrement Functions can be implemented as shown below. These functions only increment and decrement the COUNTER (on the work tape) by one. Note that these functions are a simple but have complex Turing machine representations because the COUNTER is represented by an absolute value of a count, followed by a sign character (ie. + or -.)

```
/* TM-work-head starts at rightmost on input tape */
/* The work tape (called COUNTER) will start off  */
/* as a string of zeroes, enclosed in B's.        */

/* Increment by 1 Case                            */
If (Input-head points to '1'
   and COUNTER sign is negative)
Then If (COUNTER=0)
     Then do;
          Change sign of COUNTER to positive;
          COUNTER=1;
          RETURN to main program;
```

```
          end;
     Else do;
          CALL DECREMENT(COUNTER);
          RETURN to main program;
          end;
Else;
Loop Until ( TM-work-head points to '0') ;
    If (TM-work-head points to '1')
    Then Replace '1' by '0';
    Move TM-work-head to left one step;
End Loop;
Replace '0' by '1';
Move TM-work-head to right until end-marker;
/* End of Increment by 1 Case                          */

/* Decrement by 1 Case                                 */
If (Input-head points to '0'
    and COUNTER sign is negative)
Then If (COUNTER=0)
     Then do;
          Change sign of COUNTER to negative;
          COUNTER=-1;
          RETURN to main program;
          end;
```

```
     Else do;
          CALL INCREMENT(COUNTER);
          RETURN to main program;
          end;
Else;
Loop Until ( TM-work-head points to '1') ;
    If (TM-work-head points to '0')
    Then Replace '0' by '1';
    Move TM-work-head to left one step;
End Loop;
Replace '1' by '0';
Move TM-work-head to right until end-marker;
/* End of Decrement by 1 Case                    */

/* Counting done in binary.                      */
```

For the sake of rigorous formality, the above algorithm corresponds exactly to the following Turing Machine:

T=<Q,I,W,O,D,NS,q0,qF> such that:

Q ={q0,q1,qtest,qtestA,qnot0,qeq0,qdec,qinc,qmove,qF}

I = { 0 , 1 , B } where B signifies blank

Preliminaries

W = { 0 , 1 , B , + , - }

O = {T,F}

D = {L,R,S}

q0 in Q is the start state

qF is the final state

NS is a next move function (given below)

NS(q0, 1 , B) = < q1 , S , + , S >

NS(q0, 0 , B) = < q1 , S , + , S >

NS(q1, 1 , +) = < qinc , S , + , L >

NS(q1, 0 , +) = < qtestA , S , + , L >

NS(q1, B , +) = < qtest , S , + , L >

NS(q1, B , -) = < qtest , S , - , L >

NS(q1, 1 , -) = < qtestA , S , - , L >

NS(q1, 0 , -) = < qinc , S , - , L >

NS(qtest, B , 0) = < qtest , S , 0 , L >

NS(qtest, B , 1) = < qF , S , F , S >

NS(qtest, B , B) = < qF , S , T , S >

Comment: Below are the increment-decrement routines.

NS(qtestA, 0 , 0) = < qtestA , S , 0 , L >

NS(qtestA, 1 , 0) = < qtestA , S , 0 , L >

NS(qtestA, 0 , 1) = < qnot0 , S , 1 , R >

NS(qtestA, 1 , 1) = < qnot0 , S , 1 , R >

NS(qtestA, 0 , B) = < qeq0 , S , B , R >

NS(qtestA, 1 , B) = < qeq0 , S , B , R >

NS(qnot0, 0 , 0) = < qnot0 , S , 0 , R >

NS(qnot0, 0 , 1) = < qnot0 , S , 1 , R >

NS(qnot0, 1 , 0) = < qnot0 , S , 0 , R >

NS(qnot0, 1 , 1) = < qnot0 , S , 1 , R >

NS(qnot0, 0 , +) = < qdec , S , + , L >

NS(qnot0, 1 , -) = < qdec , S , - , L >

NS(qeq0, 0 , 0) = < qeq0 , S , 0 , R >

NS(qeq0, 0 , 1) = < qeq0 , S , 1 , R >

NS(qeq0, 1 , 0) = < qeq0 , S , 0 , R >

NS(qeq0, 1 , 1) = < qeq0 , S , 1 , R >

NS(qeq0, 0 , +) = < qinc , S , - , L >

NS(qeq0, 1 , -) = < qinc , S , + , L >

NS(qdec, 0 , 0) = < qdec , S , 1 , L >

NS(qdec, 1 , 0) = < qdec , S , 1 , L >

NS(qdec, 0 , 1) = < qmove, S , 0 , L >

NS(qdec, 1 , 1) = < qmove, S , 0 , L >

NS(qinc, 0 , B) = < qmove, S , 1 , R >

NS(qinc, 1 , B) = < qmove, S , 1 , R >

NS(qinc, 0 , 0) = < qmove, S , 1 , R >

NS(qinc, 1 , 0) = < qmove, S , 1 , R >

NS(qinc, 0 , 1) = < qinc , S , 0 , L >

NS(qinc, 1 , 1) = < qinc , S , 0 , L >

```
NS( qmove, 0 , - ) = < q1 , L , - , S >
NS( qmove, 0 , + ) = < q1 , L , + , S >
NS( qmove, 1 , - ) = < q1 , L , - , S >
NS( qmove, 1 , + ) = < q1 , L , + , S >
NS( qmove, 0 , 0 ) = < qmove , S , 0 , R >
NS( qmove, 0 , 1 ) = < qmove , S , 1 , R >
NS( qmove, 1 , 0 ) = < qmove , S , 0 , R >
NS( qmove, 1 , 1 ) = < qmove , S , 1 , R >
End-of-TM.
```

To demonstrate the operation of this Turing Machine, a simulation of its behavior (as a trace) follows:

Trace of Turing Machine behavior on input 1110.

Input Tape	State	Work Tape
v 1 1 1 0	q0	B B B B B B B
v 1 1 1 0	q1	B B B B + B B
v 1 1 1 0	qinc	B B B B + B B
v 1 1 1 0	qmove	B B B 1 + B B
v 1 1 1 0	q1	B B B 1 + B B
v 1 1 1 0	qinc	B B B 1 + B B
v 1 1 1 0	qinc	B B B 0 + B B
v 1 1 1 0	qmove	B B 1 0 + R B
v 1 1 1 0	qmove	B B 1 0 + B B
v 1 1 1 0	q1	B B 1 0 + B B
v 1 1 1 0	qinc	B B 1 0 + B B
v 1 1 1 0	qmove	B B 1 1 + B B
v 1 1 1 0	q1	B B 1 1 + B B

```
       v                                  v
1 1 1 0          qtestA      B B 1 1 + B B

       v                                    v
1 1 1 0          qnot0       B B 1 1 + B B

       v                                  v
1 1 1 0          qdec        B B 1 1 + B B

       v                                    v
1 1 1 0          qmove       B B 1 0 + B B

         v                                  v
1 1 1 0 B        q1          B B 1 0 + B B

         v                                  v
1 1 1 0 B        qtest       B B 1 0 + B B

         v                                v
1 1 1 0 B        qtest       B B 1 0 + B B

         v                                v
1 1 1 0 B        qF          B B F 0 + B B
```

Machine halts in final state qF.

Output symbol F signifies 1110 is not in LEQ.

Informal Argument of Lemma 1 (for On-line Case):

Informally, in the above Turing Machine, COUNTER requires log(m) space at worst where m is the length of the input string. This corresponds to the string 1**m. Note: 1**(m/2)0**(m/2) is the worst case of the accepted words. The proof is essentially reducing the recognition problem by the Turing Machine to a counter

principle.

The argument begins by showing log(j) space is the minimum space possible to record numbers 0 through j. Since the input string is scanned on-line, when the input head has passed over any prefix 1**(k), it must be capable recording the exact number of ones, namely k. Otherwise, if it did not have the capacity to distinguish between every k (where 0<k<m), then there would be two distinct values k and k1 such that for every word w :

(1**k)w would be in LEQ iff (1**k1)w is in LEQ

This would clearly violate the definition of LEQ because w would have COUNTER values of -k and -k1 simultaneously, which is clearly a contradiction. Hence, the on-line Turing Machine must be capable of counting upto m exactly.

Note: the contradiction condition would also have serious ramifications on the context-free nature of (1**k)(0**k).

Formal Proof of Lemma 1 for On-line Case:

Sublemma: On-line counting of a string 1**j requires log(j) space on a single-head on-line Turing Machine.

Proof of Sublemma: Assume a k-symbol alphabet for the work-space of such a Turing Machine. Assume X cells exist on that work tape. By definition, each cell can hold exactly one symbol.

The question arises: How many symbol patterns can be stored in X cells? From combinatorial theory (ref. Liu[1968]), k**X patterns can be stored. So in X cells, at most k**X different numbers can be stored. If we use integer 0 and require counting by one (a strict successor function constraint), then we can only represent numbers from 0 to k**X-1. Let i=k**X. Then we are saying, to count from 0 to i, we need log(i) cells. To count a sequence of j ones on a Turing Machine, we need log(j) space. The above log functions are in base k.

Note: In changing the base of counting (ie. from k to k'), we have log-sub-k'(j) = log-sub-k(j)/log-sub-k(k'), so this is really just dividing by a constant factor; hence, logspace remains.

(QED to Sublemma)

So far, we have shown that log(j) space is needed to count from 0 to j (with no gaps.) All that remains is to show that the count (j) must be as large as m.

So, in our on-line Turing Machine for 1**(m/2)0**(m/2), as the head sweeps over 1**(m/2), it must use:

log(m/2)=log(m)-log(2)=O(log(m)) space. By the above lemma, to have the correct count of ones, log-space is absolutely required. Any fewer cells would not be capable of recording this count. After the midpoint (of 1**(m/2)0**(m/2)) is past, the countdown would not cause any further growth. Note: for the 1**m case, the space requirement would be again O(log(m).)

This count information is necessary because if at any point in the input scan (say when 1**k has been passed over by the input) we should happen to lose any count information, then the final output for the Turing Machine would be the same for (1**k)w as (1**k2)w (where w is any word and k not equal to k2.) So any loss of count information would result in erroneous outputs.

QED to Lemma 1 On-line Case

Theorem 1: The word problem for braid groups requires at least log-space (in the worst case) on an on-line single-input-head Turing Machine.

Proof: Using the necessary condition (of Proposition 1), we can look at the problem in terms of the language LEQ, the language composed of all words w (from alphabet {0,1}) such that the number of zeros equals the number of ones. Clearly, any Turing Machine can interpret a braid word in terms of LEQ words (under the mapping s(i)-->1 and s(i)'-->0, where s(i) is a positive braid generator) and furthermore, this interpretation can be done in the finite store of any Turing Machine model. So now a necessary condition for

the word problem for braid groups reduces to the recognition problem for LEQ. Since LEQ is recognizable on an on-line single head Turing Machine in log-space, the word problem for braid groups requires at least log-space in the worst case (on that model.)

QED to Theorem 1 On-line Case

Discussion:

For a single-head off-line Turing Machine, the situation appears to be similar in terms of the worst-case. Counting can occur upto a certain value. By the time that the midpoint is reached, a count of the traversed side must be made. Assume we did not count all of the ones but every kth one (k a constant) in this one pass; then the space will be O(log(m/k))=O(log(m).) Assume each pass counted k**i where i is the pass number; then, for large k**i (ie. close to m/2), the remainder in such a count would require O(log(k**i)) which would be O(log(m/2).) If we left that remainder for further counting by smaller chunks (say k**p where p<i), then to record the address of that position where the remainder begins requires log-space.

Unfortunately, all this proves is that standard counting methods would fail to use less than log-space on an off-line Turing Machine for the worst case ; however, instead of k**i, there could possibly exist other functions f(i) which, after many successive passes, would cover all counts from 0 through m/2 (by a conjunction of cases) and yet use less than log-space. Such a system would very likely be nonoptimal in time usage, moving the input head over sufficient scans to collect sufficient partial results. No such class of functions seems to exist and there is much intuitive evidence of its' nonexistence but this remains to be proven.

For the average space complexity of the on-line single-head Turing Machine model, the situation has been analytically solved (below) but the exact function has not been resolved in terms of its' big-O class. Let w be an arbitrary word of length m over alphabet {0,1}. Let SP(w) = the maximal space used by word w on the work-tape (which is just a counter.) SP(w) is really the least integer greater than the log of the counter value at it's maximal point in the program run. For example, SP(1011011110000)= log-sub2(5) = 3. Let

r=the value of the counter in the maximal step (ie. r=2**SP(w).) The average case space usage of this system can be computed by an expected value:

$$\text{Average Space} = \sum_{\substack{\text{over} \\ \text{all words } w}} \text{SP(w)*Probability(w being the word)}$$

The maximal counter value r may be interpreted as the maximal value of the sequential sum of m independent binomial random variables (values +1 and -1). In other words, r is the maximum value of the random walk of length m. The probability that such a sum of length m would have a maximal value of r is:

$$\text{Prob}(m,r) = C(m,(m+r)/2)*2**(-m)$$

(according to Feller[1968], p74-75,87-89 or Renyi[1970], p233.)

At this point, the problem adopts a new twist; if the situation was as simple as presented so far, the sum may be repartitioned as follows:

$$\text{Average Space} = \sum_{r=-m}^{r=m} \log(|r|) * C(m,(m+r)/2)*2**(-m)$$

(for words of length m)

Preliminaries

This classical random walk approach cannot be used directly because a maximal value of r could still result in a negative count below -r (resulting in underestimation of space usage.) Instead, the problem should be reformulated in terms of the absolute value of the random walk. Let the probability that the absolute value of such a sum (ie. random walk) of length m (ie. m steps) would have a maximal value of r be denoted as PA(m,r). With the absolute value condition, the expected value becomes:

$$\text{Average Space (for words of length m)} = \sum_{r=0}^{r=m} \log(|r|) * PA(m,r).$$

Clearly, PA(m,0)=0, so the sum can be assumed to begin at r=1. Also the log(|r|) is really the least integer greater than log(|r|).

At this point, we could compute the exact value of PA(m,r) by a very lengthy derivation starting with the fundamental assumptions of random walks and adding conditions. For sake of brevity, this approach will be avoided. Another method, based on taking differences

Prob(m,r)-Prob(m,r-1) seems semantically correct at first but is erroneous because it neglects to consider the cases of negative sums that go below r and r-1 respectively. These negative cases are not equiprobable in the r and r-1 case. Therefore, this difference approach fails (as did all the other tested reformulations in terms of Prob.)

One alternative is to approximate, of course keeping track of whether the approximation is an over-estimate or under-estimate. We know that Prob(m,r) is greater than or equal to PA(m,r) because Prob is unbounded for negative r while PA is. So we can approximate the expected value in terms of Prob(m,r) as:

$$\text{Average Space} < 2 * \sum_{r=1}^{r=m} \log(|r|) * C(m,(m+r)/2) * 2^{(-m)}$$

(for words of length m)

No simple method of resolving this to be non-logspace has been found. Program runs show this is sub-O(log(m)) but a direct proof via series manipulation is not obvious.

Instead, using an approximation (due to Renyi page 234):

Probability(Maximum >=r) =< 2e**((-r**2)/(2m)).

As r grows, this function decreases exponentially. By substitution into the average space inequality, the average space of the single-head on-line Turing Machine (for LEQ) can be shown to be less than log(m)-space. Exactly what function it is has not been determined yet.

For the average space complexity of the off-line single-head Turing Machine, this issue remains open by the same arguments of the worst case for the off-line model (described above.) The issue is: can large numbers of repeated scans collect enough fragmentary information to completely determine membership in LEQ and yet use less than log-space in each pass.

Lemma 2: The necessary condition for WP(B(n+1)) in theorem 1 requires no work-tape space on a 2-input-head Turing Machine (even with the on-line condition.)

Proof: Proof by construction of an on-line 2-input-head Turing Machine which requires no workspace and can still recognize LEQ.

```
/* TM-input-heads starts at leftmost on input tape*/

Loop Forever;
    If (TM-input-head1 points to right-end)
    Then
        /* Need to test if an excess of zeros.*/
         Loop Until (TM-input-head2 points to '0');
            If (TM-input-head2 points to right-end)
            Then
                Print 'Word is in LEQ.';
                Halt;
            End-if;
            Move TM-input-head2 right once;
         End-loop-until;
         /* Have at least one excess zero. */
         Print 'Word is not in LEQ.';
         Halt;
    End-if;
    If (TM-input-head1 points to '0')
    Then ;   /* Head1 ignores input='0' */
```

```
    Else
         /*Since Head1 hits a '1', Head2 must      */
         /* find a '0'.                            */
         Loop Until (TM-input-head2 points to '0');
            If (TM-input-head2 points to right-end)
            Then
                Print 'Word is not in LEQ.';
                Halt;
            End-if;
            Move TM-input-head2 right once;
         End-loop-until;
         /* Since head2 hit a '0', all is ok.       */
         /* To prevent double counting, move again.*/
         Move TM-input-head2 right once;
    End-if;
    Move TM-input-head to right one step;
End Loop-Forever;
```

For the sake of rigorous formality, the above algorithm corresponds exactly to the following Turing Machine:

Preliminaries

T=<Q,I2,W,O,D,NS2,q0,qF> such that:

Q = {q0 , qfind1 , qfind0 , qF}

I2 = IxI where:

I = { 0 , 1 , B } where B signifies blank

W = { B }

O = {T,F}

D = {L,R,S}

q0 in Q is the start state

qF is the final state

NS2 is a next move function (given below)

NS2(q0, <B , B> ,B) = < qF , <S, S> , T , S>

NS2(q0, <1 , 1> ,B) = < qfind0 , <S, R> , B , S>

NS2(q0, <0 , 1> ,B) = < q0 , <R, R> , B , S>

NS2(q0, <1 , 0> ,B) = < q0 , <R, R> , B , S>

NS2(q0, <0 , 0> ,B) = < qfind1 , <R, S> , B , S>

NS2(q0, <0 , B> ,B) = < q0 , <R, S> , B , S>

NS2(q0, <B , 0> ,B) = < qF , <S, S> , F , S>

NS2(q0, <1 , B> ,B) = < qF , <S, S> , F , S>

NS2(q0, <B , 1> ,B) = < q0 , <S, R> , B , S>

NS2(qfind0, <1 , 1>,B) = < qfind0, <S, R> , B, S>

NS2(qfind0, <1 , 0>,B) = < q0 , <R, R> , B, S>

NS2(qfind0, <1 , B>,B) = < qF , <S, S> , F, S>

```
NS2( qfind1, <0 , 0>,B ) = < qfind1, <R, S> , B, S>
NS2( qfind1, <1 , 0>,B ) = < q0     , <R, R> , B, S>
NS2( qfind1, <B , 0>,B ) = < qF     , <S, S> , F, S>
```

The following example will demonstrate the behavior of this Turing Machine.

Trace of Turing Machine behavior on input 110100.

(Note: ^ is head 1 and v is head 2.)

```
Input Tape          State        Work Tape

v
1 1 0 1 0 0         q0           B
^                                ^

  v
1 1 0 1 0 0         qfind0       B
^                                ^

    v
1 1 0 1 0 0         qfind0       B
^                                ^

      v
1 1 0 1 0 0         q0           B
  ^                              ^

        v
1 1 0 1 0 0         qfind0       B
  ^                              ^

          v
1 1 0 1 0 0         q0           B
    ^                            ^

          v
1 1 0 1 0 0         qfind1       B
      ^                          ^

            v
1 1 0 1 0 0 B       q0           B
        ^                        ^

            v
1 1 0 1 0 0 B       q0           B
          ^                      ^

              v
```

```
1 1 0 1 0 0 B    qF          T
           ^                 ^
```

Machine halts in final state qF.
Output symbol T signifies 110100 is in LEQ.

(QED to Lemma 2)

Obviously, the average work-space usage of the 2-head Turing Machine is zero. So, this lowerbound is not a very useful one.

Note: The analysis of the 2-input-head approach does have one hidden flaw. In the real world, a head (or a pointer) requires log(memory used) space to record the position of an average cell. The assumption that the lower bound due to theorem 1 fails because of the addition of another head is not a real one; the extra head requires logspace on any real machine.

Previous work on random walks over groups was carried out by Kesten[1959]. His research was directed toward the recurrence problem for the identity, subgroups, and other events in random walks over groups. His methodology was more probabilistic than

combinatorial. Later results in the countably infinite Abelian case were developed by Kesten and Spitzer[1965]. Our approach is more combinatorial and concentrates on the maximal distances in random walks, with an ultimate goal in the complexity issues. Spitzer[1976] presented a theoretical but introductory approach to random walks.

I.D The Growth Rate of the Braid Group Languages

The previous theorems establish an upper bound result.

Lemma 3: There are at most C(m,m/2)*(n**m) identity braid words of length m in B(n+1).

Proof: By proposition 2, m/2 generators will be inverted (in an identity braid word of length m.) There are C(m,m/2) ways to chose which m/2 positions will be the inverted. Once inversion positions are selected, the remainder of the problem is: how many ways can m positions be filled (with replacements) using the (n) generators. There are (n**m) selections. Hence, C(m,m/2)*(n**m).

(QED to Lemma 3)

The above lemma demonstrates an upper bound on the set, not an exact value. Using this value, the probability of a word being an identity word is

```
PR(m,n) = C(m,m/2)*(n**m) / ((2*n)**m)
```

where (2*n)**m is the total number of braid words of length m. The expression reduces to:

```
PR(m,n) = C(m,m/2) / (2**m)
```

A binomial expansion of 2**m contains C(m,m/2) as the middle term, which means it is the largest in the summation. So at this point, approximate analytic methods may be too inaccurate; instead, a program proves more helpful in approximating PR(m,n). First computing the products term-by-term (using the expression

```
PR(m,n)= (m / (4*1))          * ((m-1) / (4*2)) * ... *
         ((m/2+1) /(4*(m/2-1))) * ((m/2)/(4*m/2))
```

resulted in overflows near the center of the product PR(m,n) for n=381; a far better approach is the computation of F(m,n) = log(PR(m,n)). After programming it, the function shows the following growth rate:

m =	2	10	1000	14000	37000	60000	70000
F(m,n)=	-.301	-.34	-1.29	-1.87	-2.08	-2.19	-2.22

Since F is log(PR), this is a terribly slowly converging function (of course bounded by 0 and 1 due to the binomial theorem.) Such a convergence is evident (and provable by monotonicity and boundedness.) These values are far above the actual values. These values correspond to the true values for free abelian groups. In conclusion, this corollary is far too weak to be of statistical value but can act as an upper bound.

Note: In the above analysis, PR(m,n) finally reduces to a function of m only. This is another property by which the above differs from the braid

group. In B(n), as n increases, the ratio of identity words to all words decreases.

As a lower bound on the number of identity words of length m, the free group can be used (again, a weak bound.) An identity word in the free group can be expressed as:

W = w1 w2 ... wk

where wi = a word composed only of generators

g and g' with an equal number of each.

Lemma 4: For k=1, there are n*C(m,m/2) identity words of length m.

Proof: There are n choices for generator g. Once g is chosen, there are C(m,m/2) ways to distribute the negative generators.

QED

Lemma 5: For k=2, there are

n*C((m-2),(m-2)/2)*(n-1)*C(2,1) +
n*C((m-4),(m-4)/2)*(n-1)*C(4,2) +
... +

n*C(2,1)*(n-1)*C((m-2),(m-2)/2)

identity words of length m.

Proof: Follow the same process as k=1 but split into two subwords. Note that the n-1 prevents the generator in w2 from being the same as in w1.

QED to Lemma 5

Note: for cases k=3,4,...m-1, the sums become very complex and while still describable, the expressions neither provide any deep insight, nor are they apparently reducible to simpler ones.

...

For k=m, there are n*C(2,1)*((n-1)*C(2,1))**((m-2)/2).

By summing over k=1 to m, we get the total number of identity words of length m. Unfortunately, for k=3 there are approximately (m/2)*(m/2-1)/2 terms, making the expressions too complex to deal with.

Preliminaries

For early work on group languages, consult Anisimov[1973]. More recent progress in the area is in Muller and Schupp[1983].

II.0 Artin's Algorithm

Artin [1925-1926] defined B(n+1) (the braid group on n strands) and demonstrated that braids can be characterized as automorphisms of the free group F(n).

Artin's algorithm takes a braid word W (expressed in generators s(i) and s(i)'), converts each generator into an action, applies those actions on the vector <1,2,...,n>, and freely reduces that vector of words. The actions that these generators create are:

```
s(i) : x(i)    --> x(i+1)
       x(i+1)  --> x(i+1)' x(i) x(i+1)
       x(k)    --> x(k)              for all k>i+1 or k<i

s(i)': x(i)    --> x(i) x(i+1) x(i)'
       x(i+1)  --> x(i)
       x(k)    --> x(k)              for all k>i+1 or k<i
```

A simplified version of this algorithm would be:

Artin's Algorithm

```
Step 1 : READ BRAID WORD W ;
Step 2 : INITIALIZE TUPLE = <1,2,...,N> ;
Step 3 : LOOP UNTIL (W IS EMPTY) ;
Step 4 :   G = FIRST_GENERATOR( W ) ;
Step 5 :   W = W BUT WITH FIRST GENERATOR DELETED ;
Step 6 :   /* APPLY G AS AN ACTION ON TUPLE */
           IF (G is a positive generator) THEN
            TUPLE(G)   =TUPLE(G+1);
            TUPLE(G+1)=TUPLE(G+1)' TUPLE(G) TUPLE(G+1);
           ELSE
            TUPLE(G)    =TUPLE(G) TUPLE(G+1) TUPLE(G)' ;
            TUPLE(G+1) =TUPLE(G) ;
           ENDIF;
Step 7 :   /* FREELY REDUCE TUPLE */
           CANCEL ALL OCCURRENCES OF a'a AND
                  aa' in TUPLE for all generators a ;
Step 8 : END LOOP ;
Step 9 : IF (TUPLE=<1,2,..,n>) THEN PRINT('IDENTITY') ;
                  ELSE PRINT('NOT IDENTITY') ;
Step 10: HALT ;
```

Artin's Algorithm

The above algorithm would appear to run in exponential time on the average because the tuple's words appear to grow exponentially. This, however, is no guarantee; in general, L-systems appear to have an exponential growth behavior and yet many cases exist which grow linearly. Likewise here, the free-reductions will demonstrate some clearly non-exponential cases.

For example, taking braid word s(2)s(1)'s(2)s(1)s(2), the algorithm would compute as follows:

```
Initial Tuple : < 1   ,   2     ,   3     ,   4   >
Apply s(2) :
          < 1     ,     3     ,   3'23  ,   4   >
Apply s(1)':
          < 131'  ,     1     ,   3'23  ,   4   >
Apply s(2) :
          < 131'  ,     3'23  , (3'2'3)1(3'23) ,4>
Apply s(1) :
        < 3'23 , (3'2'3)131'(3'23) , (3'2'313'23) ,4>
Apply s(2) :
```

```
<3'23 ,(3'2'313'23) ,(3'2'31'3'23)(
                        3'2'3131'3'23)(3'2'313'23),4>
```

Free reduction gives:

```
          <  3'23 , (3'2'313'23) , (3'2'3)(23), 4  >
```

An even better algorithm would not do a complete free reduction over the TUPLE words but only cancellations on the boundaries of the words being concatenated at each step. This version of the algorithm was implemented on the VAX 11/780 in PL\I. This version of the algorithm will be used in all further analysis. No further improvements are evident; "folding" the words down their "midpoint" generators would only cut space use in half.

II.A Exponential Worst Case for Artin's Algorithm

An analysis of growth rates of the freely-reduced words formed by applying generator pairs will constitute the core of the following proof:

Artin's Algorithm

Theorem 1: Artin's algorithm has an exponential time and space worst case.

Proof 1: First, we will need some lemmas, definitions, and cases.

Partition the set of all braid words of length two into seven sets:

Set A = { s(i)s(i)' or s(i)'s(i) }

Set B = { s(i)s(i) or s(i)'s(i)' }

Set C = { g(i)g(k) where
g(m) = s(m) or s(m)' and |i-k|>1}

Set D = { s(i)'s(i-1)' or s(i)s(i+1) }

Set E = { s(i)'s(i-1) or s(i)s(i+1)' }

Set F = { s(i)s(i-1)' or s(i)'s(i+1) }

Set G = { s(i)'s(i+1)' or s(i)s(i-1) }

Def: A power-word of a set S = { a, b, ...} is a word of the form a**n.

Artin's Algorithm

Lemma A: Artin's algorithm runs in constant space and linear time for power-words of set A.

Proof A: Every time relator s(i) is applied (giving a TUPLE word of length 3 (like 3'23)), the relator s(i)' cancels the effect to give the initial TUPLE. At most one word will have length of at most three.

End of Proof A

Lemma B: Artin's algorithm runs in linear space and quadratic time for power-words of set B.

Proof B: We need to prove this for only s(i)s(i); the inverse case will hold by symmetry. With no loss of generality, we can assume i=1, (all other cases will be isomorphic except for position.) For (s(i))**n where n<4, TUPLE words grow in the following manner:

Initial Tuple : < 1 , 2 , 3 , 4 >

Apply s(1) :

< 2 , 2'12 , 3 , 4 >

Apply s(1) :

< 2'12 , (2'12)'(2)(2'12) , 3 , 4 >

which reduces to:

< 2'12 , 2'1'212 , 3 , 4 >

Apply s(1) :

<2'1'212, (2'1'212)'(2'12)(2'1'212), 3, 4>

This looks exponential but note the reduction:

< 2'1'212 , (2'1'2'12)(12) , 3 , 4 >

From this point, an induction proof can start:

Sublemma B1: For n>2, TUPLE words are of the form:

<((2'1')**k)2((12)**k), ((2'1')**k)(2'12)((12)**k),3,4>

for k odd, where n=2k+1

and

<((2'1')**k)(2'12)((12)**k),

((2'1')**(k+1))2((12)**(k+1)),3,4>

for k even, where n=2k+2.

Proof of B1:

Assume TUPLE words:

<((2'1')**k)2((12)**k),

((2'1')**k)(2'12)((12)**k),3,4> for k odd,

(the odd step of the induction, n=2k+1 > 2)

(i.e. TUPLE for s(i)**n.)

Then, for s(i)**(n+1), we have by applying s(i):

```
<((2'1')**k)(2'12)((12)**k) ,
    ((2'1')**k)(2'12)((12)**k)'((2'1')**k)2((12)**k)((
        2'1')**k)(2'12)((12)**k) , 3 , 4  >
```

The second TUPLE reduces to :

```
((2'1')**k)(2'1'2)((12)**k)((2'1')**k)2)((
                                    2'12)((12)**k)))
```

which further reduces to:

```
    ((2'1')**k)(2'1'212)((12)**k))
```

which factors to:

((2'1')**(k+1))(2)((12)**(k+1))) , which is the even case.

So the even condition of the induction arises and is proven from the odd case, whose proof is completed below:

For s(i)**(n+2), we have by applying s(i) on the above case's first TUPLE word: ((2'1')**(k+1))2((12)**(k+1)).

For the second TUPLE word:

```
((2'1')**(k+1))2((12)**(k+1))'((2'1')**k)(
          2'12)((12)**k))((2'1')**(k+1))2((12)**(k+1))
```

which reduces to:

```
((2'1')**(k+1))2'((12)**(k+1))((2'1')**k)2')(
                                  2(12)**(k+1))
```

reducing further to:

```
      ((2'1')**(k+1))(2'12))((12)**(k+1)).
```

This completes the odd n case induction and so the even case will also
hold.
End of Proof B1.

In the above proof, TUPLE grows by 4 generator-terms at each step, so s(i)**n will cause space requirement O(4n).

Before each reduction is completed, at most twice the space is needed, so it requirement is O(8n). In either case, the function is linear space. Since at each step we have linearly many reductions and linearly many steps, the time required is quadratic.

End of Proof B.

Lemma C: Artin's algorithm runs in linear space and quadratic time for power-words of set C.

Proof C: In this case, braid words (g(i)g(k))**n are considered where g(m) = s(m) or s(m)'

and | i - k | > 1.

g(i) has a growth effect on TUPLE positions i and i+1.
g(k) has a growth effect on TUPLE positions k and k+1.

Since | i - k | >1, g(i) and g(k) operate independently and produce independent complexity contributions. So,
Space-Complexity((g(i)g(k))**n)

= Space-Complexity(g(i)**n)+ Space-Complexity(g(k)**n)

= 2(O(8(n/2))), according to lemma B.
So we have linear space.

Similarly, the time requirement is quadratic.

End of Proof C.

Lemma D: Artin's algorithm runs in linear space and quadratic time for power-words of set D.

Proof D: We need to prove this for only s(i)s(i+1); the other case (ie. s(i)'s(i-1)') will hold by braid-symmetry (ref. Garside [1965], the mirror-images of braids.) With no loss of generality and to save notational space, we can assume i=1, (all other cases will be isomorphic except for position.) For (s(1)s(2))**m, TUPLE words grow in the following manner:

```
Initial Tuple : <  1   ,    2    ,  3   , 4 >
Apply s(1) :
  < 2  ,   2'12 ,  3   ,   4  >
Apply s(2) :
  < 2  ,  3   ,  3'2'123 , 4  >
Apply s(1) :
  < 3 , 3'23 ,   3'2'123 , 4  >
Apply s(2) :
```

< 3 , 3'2'123 , (3'2'123)'(3'23)(3'2'123) , 4>

which reduces to:

< 3, 3'2'123 , 3'2'1'2123, 4 >

Apply s(1) :

< 3'2'123 , 3'2'1'232'123 , 3'2'1'2123, 4 >

Apply s(2) :

< 3'2'123 , 3'2'1'2123 ,

(3'2'1'2123)'(3'2'1'232'123)(3'2'1'2123) ,4>

which reduces to:

< 3'2'123, (3'2'1')2(123), (3'2'1')3(123), 4 >

Apply s(1) :

<(3'2'1')2(123), (3'2'1')2'12(123), (3'2'1')3(123), 4>

We needed all the above to reach the first step of the induction. From this late point, our induction starts:

Assume the following TUPLE:

< ((3'2'1')**k)(w)(123)**k, ((3'2'1')**k)(x)(123)**k,

((3'2'1')**k)(y)(123)**k, 4 >

Applying s(2) produces:

< ((3'2'1')**k)(w)(123)**k , ((3'2'1')**k)(y)(123)**k ,

(((3'2'1')**k)(y)(123)**k)'((3'2'1')**k)(

x)(123)**k((3'2'1')**k)(y)(123)**k , 4 >

which reduces to:

```
< ((3'2'1')**k)(w)(123)**k , ((3'2'1')**k)(y)(123)**k ,
   ((3'2'1')**k)(y'xy)(123)**k    , 4 >
```

Applying s(1) produces:

```
< ((3'2'1')**k)(y)(123)**k ,
   (((3'2'1')**k)(y)(123)**k)'((3'2'1')**k)(
                w)(123)**k((3'2'1')**k)(y)(123)**k    ,
     ((3'2'1')**k)(y'xy)(123)**k ,    4 >
```

which reduces to:

```
<((3'2'1')**k)(y)(123)**k, ((3'2'1')**k)(y'wy)(123)**k,
     ((3'2'1')**k)(y'xy)(123)**k    , 4 >
```

So applying s(2)s(1) preserves the (3'2'1')**k left boundary and (123)**k right boundary. The growth of TUPLE depends on the "central" words (in w, x, y) and so we focus on them in the following table (starting where we left off in our initial step (ie. ((s(1)s(2))**3s(1)):

Artin's Algorithm

w=2	x=2'12	y=3
Apply s(2)		
2	3	3'2'123
Apply s(1)		
3	3'23	3'2'123
Apply s(2)		
3	3'2'123	(3'2'123)'(3'23)(3'2'123)
which reduces to		
3	3'2'123	3'2'1'2123
Apply s(1)		
3'2'123	(3'2'123)'3(3'2'123)	3'2'1'2123
which reduces to		
3'2'123	(3'2'1')232'(123)	3'2'1'2123
Apply s(2)		
3'2'123	(3'2'1')2(123)	(3'2'1'2123)'(3'2'1')(232')(123)(3'2'1'2123)
which reduces to		
3'2'123	(3'2'1')2(123)	(3'2'1'2'123)(3'2'1'23)(123)
which further reduces to		
3'2'123	(3'2'1')2(123)	(3'2'1')3(123)
Apply s(1)		
and reduce to		

(3'2'1')2(123) (3'2'1')(2'12)(123) (3'2'1')3(123)

At this point, we note the "loop structure" and so the induction step-size would be (s(2)s(1))**3 (ie. six.) So, we have:

For m > 1, the TUPLE structure will be:

<((3'2'1')**m)2(123)**m , ((3'2'1')**m)(2'12)(123)**m ,
((3'2'1')**m)3(123)**m, 4 >

given braid word ((s(1)s(2))**(3m))s(1).

This implies O(6(m/3)) growth, which means linear space growth and quadratic time. End of Proof D.

Lemma E: Artin's algorithm runs in exponential space and time for power-words of set E.

Proof E: We need to prove this for only s(i)s(i+1)' ; the other case (ie. s(i)'s(i-1)) will hold by braid-symmetry (ref. Garside [1965], the mirror-images of braids.) With no loss of generality and to save notational space, we can assume i=1, (all other cases will be isomorphic except for position.) For (s(1)s(2)')**m, TUPLE words grow in the following manner:

Artin's Algorithm

Initial Tuple : < w1 , w2 , w3 , w4 >

Apply s(1) :

< w2 , w2'w1w2 , w3 , w4 >

Apply s(2)':

< w2, (w2'w1w2)w3(w2'w1'w2), w2'w1w2, w4>

Apply s(1) :

< w2'w1w2w3w2'w1'w2 ,

(w2'w1w2w3w2'w1'w2)'(w2)(w2'w1w2w3w2'w1'w2) ,

w2'w1w2 , w4>

Only w2 cancels in the middle of the second TUPLE word:

<w2'w1w2w3w2'w1'w2 ,

(w2'w1w2w3'w2'w1'w2)(w1w2w3w2'w1'w2) ,

w2'w1w2, w4>

Apply s(2)':

<w2'w1w2w3w2'w1'w2 ,

(w2'w1w2w3'w2'w1'w2w1w2w3w2'w1'w2)(w2'w1w2)(

w2'w1w2w3'w2'w1'w2w1w2w3w2'w1'w2)' ,

w2'w1w2w3'w2'w1'w2w1w2w3w2'w1'w2 >

Only w2'w1w2 cancels in the middle of the second TUPLE word.

The growth pattern of the second TUPLE word (ie. the largest)

is doubling the previous one and deleting the one from three

steps behind. So the recurrence relation for the length of

the second TUPLE is:

$$a(n) = 2(a(n-1)) - a(n-3).$$

Since w1, w2, w3 are words, this argument is an induction because the final tuple words above can be resubstituted as initial words and the process repeated. As for the initial part of the induction, we need only delete the 'w's from the above expressions (of case E); the proof is there.

All that remains is the complexity issue. a(n)=2(a(n-1))-a(n-3) is a homogenous linear difference equation. Using classical math.,

$$a(n) - 2(a(n-1)) + a(n-3) = 0$$

We have characteristic equation: b**3 - 2b**2 + 1 = 0.
This gives eigenvalues:

b=1 , b=(1+(5)**(.5))/2 , b=(1-(5)**(.5))/2

In fact, the last two roots correspond to the Fibonnaci series roots. The general solution is:

```
a(n) = A1 + A2((1+(5)**(.5))/2)**n
          + A3((1-(5)**(.5))/2)**n
```

Using boundary conditions:

a(0) = 1 , a(1) = 3 , a(2) = 7,

We have to solve:

```
1 = A1 +A2 + A3
3 = A1 +A2((1+(5)**(.5))/2)    + A3((1-(5)**(.5))/2)
7 = A1 +A2((1+(5)**(.5))/2)**2 +A3((1-(5)**(.5))/2)**2
```

The solutions are:

```
A1 = -3
A2 = 2 + ((4/5)*(5)**(0.5))
A3 = 2 - ((4/5)*(5)**(0.5))
```

Then, we need only plug back into our general solution above.

So,

```
a(n) = -3 +
```

(2 + ((4/5)*(5)**(0.5)))*((1+(5)**(.5))/2)**n +

(2 - ((4/5)*(5)**(0.5)))*((1-(5)**(.5))/2)**n

is the exact growth rate. This would be the end of this lemma's proof but we are also concerned with what this exact answer means in a more clear (or real-world) manner.

Approximately,

a(n) = -3 + (3.788)*(1.618)**n + (.21114)*(-0.618)**n .

The first and last terms drop out as insignificant.

We can adopt and alternate approach for understanding the growth rate relative to others (as follows.) Note that the solution will be bigger than the Fibonnaci series (ie. a(n)=a(n-1)+a(n-2)) but smaller than 2**n (ie. a(n)=2(a(n-1)), a very close approximation.) However this is only the second (and largest word.) The word on one side has length a(n-1) and the other has a(n-2). So we have a total TUPLE length of

a(n) + a(n-1) + a(n-2) < 2(a(n)).

So 2Fibonnaci(n) < Length(TUPLE) < 2**(n+1).
So case E requires exponential space and exponential time; really, at each step, (3a(n-1)+a(n-2)) copies and a(n-3) reductions are executed. The growth rate remains the same.
End of Proof E.

Lemma F: Artin's algorithm runs in exponential space and time for power-words of set F.

Proof F: We need to prove this for only s(i)s(i-1)' ; the other case (ie. s(i)'s(i+1)) will hold by braid-symmetry (ref. Garside [1965], the mirror-images of braids.) With no loss of generality and to save notational space, we can assume i=2, (all other cases will be isomorphic except for position.) For (s(2)s(1)')**m, the following table exemplifies the growth of TUPLE words:

```
Initial Tuple : <  w1  ,  w2    ,  w3   ,   w4   >
Apply s(2) :
                <  w1  ,  w3    , w3'w2w3 ,  w4  >
Apply s(1)':
                <  w1w3w1' , w1 , w3'w2w3 ,  w4  >
```

Artin's Algorithm

Apply s(2) :

< w1w3w1' , w3'w2w3 , (w3'w2w3)'w1(w3'w2w3) , w4 >

Note: No cancellation will occur because s(2) will cause concatenation to occur between words with boundary w1 and w3 (likewise for s(1)'.) Again, string induction causes a property (such as boundary words) to hold. Since w1 and w3 are initially 1 and 3, no cancellation can occur. As noted above, the boundary words repeat after each (s(2)s(1)') (ie. (two w1 boundary and one w3 boundary) followed by (one w1 boundary and two w3 boundary.)

The growth pattern is:

Applying s(2) implies increase total TUPLE space used by twice the third word.

Applying s(1)' implies increase total TUPLE space used by twice the first word.

The exact difference equations are:

---

a(n+1) = a(n) for n even

```
b(n+1) = c(n)                    for n even

c(n+1) = 2*c(n) + b(n)           for n even

------------------------------

a(n+2) = 2*a(n+1) + b(n+1)       for n odd

b(n+2) = a(n+1)                  for n odd

c(n+2) = c(n+1)                  for n odd

------------------------------
```

To resolve the problem with the two sets of equations (even and odd), we concentrate on the growth aspect and use the "telescoped" (or composite equations:

```
------------------------------

a(n+2) = 2*a(n) + c(n)

b(n+2) = a(n)

c(n+2) = 2*c(n) + b(n)

------------------------------
```

Now we rewrite as homogenous equations:

Artin's Algorithm

$$a(n+2) - 2*a(n) - c(n) = 0$$

$$b(n+2) - a(n) = 0$$

$$c(n+2) - 2*c(n) - b(n) = 0$$

From this point, we will treat the two steps (in n+2) as one. We now apply the shifting operator used in the calculus of finite differences.

$$(E-2)a(n) - c(n) = 0$$

$$(E)b(n) - a(n) = 0$$

$$(E-2)c(n) - b(n) = 0$$

Substituting $(E)b(n) = a(n)$, we get:

$$(E-2)(E)b(n) - c(n) = 0$$

(E-2)c(n) - b(n) = 0

Substituting b(n) = (E-2)c(n) will give:

(E-2)(E)(E-2)c(n) - c(n) = 0.

Reduction results in:

(E**3-4E**2+4E-1)c(n) = 0.

This equation has solutions:

E=1 , E=(3-(5)**(0.5))/2 , E=(3+(5)**(0.5))/2

Now we recombine this with the fact that (n/2) is the real power to use instead of n (due to the two step approach) and we have:

c(n) = C1 + C2*((3-(5)**(0.5))/2)**(n/2)
+ C3*((3+(5)**(0.5))/2)**(n/2)

Note: if you feel uncomfortable with the intuitive way of resolving the two step problem, the formal way would be to use E**2 as the operator but this would give you the same result.

Returning back to the central problem, since a(n)

and b(n) are E-linearly related to c(n), they too have the form:

```
b(n) = B1 + B2*((3-(5)**(0.5))/2)**(n/2)
          + B3*((3+(5)**(0.5))/2)**(n/2)

a(n) = A1 + A2*((3-(5)**(0.5))/2)**(n/2)
          + A3*((3+(5)**(0.5))/2)**(n/2)
```

In a big-O analysis, these functions grow more slowly than those in Case E. Note that this is primarily true because of the (n/2) exponents.

End of Proof F.

Lemma G: Artin's algorithm runs in linear space and quadratic time for power-words of set G.

Proof G: We need to prove this for only s(i)s(i-1) ; the other case (ie. s(i)'s(i+1)') will hold by braid-symmetry (ref. Garside [1965], the mirror-images of braids.) With no loss of generality and to save notational space, we can assume i=2, (all other cases will be isomorphic except for position); therefore, we concentrate on (s(2)s(1))**m. The proof

is similar to that of lemma D. The following table exemplifies the initial growth:

Initial Tuple : < 1 , 2 , 3 , 4 >

Apply s(2) :

< 1 , 3 , 3'23 , 4 >

Apply s(1) :

< 3 , 3'13 , 3'23 , 4 >

Apply s(2) :

< 3 , 3'23 , 3'2'123, 4 >

Apply s(1) :

< 3'23 , 3'2'323 , 3'2'123 , 4 >

Apply s(2) :

< 3'23 , 3'2'123 , 3'2'1'(3)123, 4>

Apply s(1) :

< 3'2'123, (3'2'1')2(123), (3'2'1')3(123), 4 >

Apply s(2) :

< 3'2'123, (3'2'1')3(123) ,(3'2'1')3'23(123), 4 >

Apply s(1) :

<(3'2'1)3(123), (3'2'1')3'13(123), (3'2'1')3'23(123),4>

From s(2)s(1) on, the loop structure becomes apparent: every (s(2)s(1))**3 will produce another (3'2'1') on the left of every TUPLE word (below 4) and (123) on the right. To prove this, first we show that previous (3'2'1') and (123)'s don't affect the situation.

spl

Assume <(3'2'1')w(123),(3'2'1')x(123), (3'2'1')y(123)>.

By applying s(2), we have:

<(3'2'1')w(123) , (3'2'1')y(123) , (3'2'1')y'xy(123)>.

By applying s(1), we have:

<(3'2'1')y(123),(3'2'1')y'wy(123), (3'2'1')y'xy(123)>.

So by induction here, (s(2)s(1))**n has no growth from (3'2'1') or (123).

So the growth pattern is due to the central words w, x, y. However, the example above demonstrates that w=3 , x=3'13, and y=3'23 will recur after (s(2)s(1))**3 and, in doing so, will generate (3'2'1') and (123) on the left and right (respectively) of three TUPLE words. So the growth rate is (O(9(n/6)) for space and

quadratic time due to the linear number of cancellations in each step.
End of Proof G.

The Remainder of the Theorem's Proof

Sledge-Hammer Argument: To prove that exponential-time and space are the worst possible, note that the transformation resulting in growth is x(i) --> x(i+1)'x(i)x(i+1). Assuming the worst (really impossibly bad) case that the iterations were applied to the same string each time and further assume that no cancellation occurs (even worse impossibility), then the largest words would be of length 3**n, which is exponential and not too far from case E. Now that we have shown 3**n is the absolute worst, cases E and F are actual cases that exponential one do exist.

Below is a proof that case E is the absolute worst, not 3**n.

Now to close the proof, we apply a "greedy" argument: Since type E patterns produce the worst exponential growth, we need only apply them to produce the maximal growth pattern. So words such as

(s(1)s(2)')**n are the worst braid words.

Note: one may argue that a combination of types will possibly produce a worse case but this is false because take any word of length n. By using other than the exponential type patterns, growth is linear. If exponential types are used, we note: the proofs above assume initial words don't cancel; however, by combining types this can't be guaranteed and so cancellations could occur (making the growth very small.) If no cancellation occurs, even then the intermediate words will be smaller than in the exponential case and so the growth will never catch up with the all exponential case. The greedy argument before this note holds, but I hope this note removes any doubts.

Note: the above paragraph seems too sketchy, here is one of four equivalent cases that remain to be proven (brevity and redundance prevent me from further exposition.)

Artin's Algorithm

Start with <w1 , w2 , w3 , w4 , w5 , >.

Applying 1 will give:

<w2 , w2'w1w2 , w3 , w4 , w5 , >.

Assume that the next generator g is nonadjacent (ie. |g-1| >1.) In that case, only strings to the left of position w2 would be affected, resulting in very little growth. If we jumped like this, in the long run, it would decrease the exponent of our grow expressions (cutting it in half.) Since 3**n would be the absolute impossible worst case, 3**(n/2) would be the result of this, which is a function less than that of lemma E. Since the absolute impossible worst is "ruined" by this policy (picking |g-1|>1), then the real strings would also be smaller than those of lemma E.

Instead let us assume the policy is not true. If |g-1|=0, then we get linear growth (according to the lemmas.) If |g-1|=1, then we either get our previous policies (in the lemmas, and hence covered) or we get rightward motions, 12'3 or 12'3'. However, these produce less of a growth because the larger expressions

remain on the left as we pick generators further right. So non-pairwise generator selection will result in short strings (ie. we produce a trail of partial growths as we move.) Since this principle holds whether we select generators moving right or left, the case is proven.

End of Proof of Theorem 1.

Note: All the above lemmas were verified by computer for braid words of length up to nine.

Note: The braid (s(1)s(2)')**n is interesting in that no-unraveling can occur. In this sense, the TUPLE words sizes represent a "good" measure of real topological complexity. This opens up a totally new area of inquiry. Topological complexity of knots could be an interesting second step.

II.B Experiments in the Average Case for Artin's Algorithm

Before proceeding further with a formal analysis of Artin's algorithm, experiments demonstrated it's average behavior. By attaching a

Artin's Algorithm

braid-word-enumerating subroutine, Artin's algorithm was run over all braid words upto a certain length. A statistical subroutine collected the results. In the resulting tables,

let: n = number of strands to the braids

m = length of braid-word

f(m,n)= average length of free-relator word

(ie. TUPLE word) (given m and n)

	n=2	n=3
m= 1 :	2.0000000000	1.6666666667
2 :	2.2500000000	2.0000000000
3 :	2.5000000000	2.3680555556
4 :	2.7187500000	2.7617187500
5 :	2.9250000000	3.1893229167
6 :	3.1145833000	3.6532118056
7 :	3.2946428570	4.1601097470
8 :	3.4638671875	4.7151311239
9 :	3.6258680556	5.3256615533
10 :	3.7800781250	5.9989243825
11 :	3.9286221591	6.7438931032

	n=4	n=5
m= 1 :	1.5000000000	1.4000000000
2 :	1.7638888889	1.6125000000
3 :	2.0555555556	1.8432291667
4 :	2.3771219136	2.0953613281
5 :	2.7335390947	2.3723095703
6 :	3.1292509717	2.6775390625
7 :	3.5696034094	
8 :	4.0604543687	

	n=6	n=7
m= 1 :	1.3333333333	1.2857142857
2 :	1.5100000000	1.4365079365
3 :	1.6991111111	1.5962301587
4 :	1.9031166666	1.7666515101
5 :	2.1243333333	

	n=8	n=9
m= 1 :	1.2500000000	1.2222222222
2 :	1.3813775510	1.3385416667
3 :	1.5194363460	1.4600332755
4 :	1.6654616566	1.5876481798

Note: these results were computed using a braid enumeration routine coupled with Artin's algorithm.

Clearly, the function f(m,n) is not linear in m (use any difference method.) By differences, it also does not appear to be quadratic. The growth rate appears to be less than quadratic and greater than linear. If it is a complicated expression with exponential terms, then it is starting extremely slowly as such. Warning: the n=2 case is totally deceptive because there is only one transformation and it's inverse (which would produce a linear to sublinear growth.) Use the tables for n>2.

Interpreting f(m,n)'s growth with respect to n, (for small m) as n increases, with more TUPLE positions, the growth effect of allowing one more braid generator to be used in constructing braid words will result in smaller TUPLE words because the larger TUPLE words could have moved to a larger set of alternate positions. In other words, the effect of the braid transformations is distributed among more positions. The exponential growth pairs would have a higher probability of being applied on short TUPLE words, showing insignificant growth (for small m.) For example, braid words in B(4) of length three will have smaller TUPLE words than braid words of B(3) of length three.

Unfortunately, for larger m, the growth behavior of relative to n becomes more complex. At m=5, the cut of curve f(m,n) (as n varies) has a peak at n=3 because n=3 has the capacity for exponential transformations (and since n=3 will receive the most concentrated effect from them, f(m,3) will grow faster than the tables with larger n.) These trends might not hold for m larger than these tables represent because the second differences for n=4 are larger than n=3. Extrapolating

this, f(m,4) will eventually out grow f(m,3). This may occur because f(m,4) will have more exponential cases than f(m,3) even though they may be more concentrated with f(m,3).

One may argue that program ARTESIAN (in the appendix) should be run longer to produce larger f(m,n) tables; however, all the above tables required a continuous week of run time on a VAX-11/780. So over a few billion braids were tested in these runs. Further experimentation would require the next generation of computers.

II.C Analytical Results of Artin's Algorithm's: Average Case

At first, a Markov matrix approach would seem appropriate for this problem. Let each row represent a generator and each column a corresponding generator. Start with an initial TUPLE vector and repeatedly take matrix products. Assume stationary conditions (in a constant state or steady growth) and solve for them. All this works fine for studying L-systems and stochastic processes but with this system such a

stationary state does not exist. As m grows, the cancellations produce drastic contractions and the exponentiations produce larger explosions. In a sense, the ahistorical aspect will neglect the long chains of cancellations and exponentiations (unless the rows and columns represent products of generators but then our matrix increases in size exponentially while it's expressive capacity for the chain-effects (ie. historical effects) grows linearly, making this modification unfeasible.) The median cases may be well represented but the total distribution would be ignored; therefore, this approximation will get worse as m increases beyond 2. So L-system methods are useless here.

Next, by computing the contribution of the exponential cases to the general case, an answer may emerge. A combinatorial view would give:

4(n-1) = number of exponential generator-pairs

There are approximately O(4(n-1)) words of length m with k=m/2 exponential pairs of one kind. They contribute O(4(n-1)2**(m/2)) at most according to lemma E. Dividing by the total number of words (2n)**m, we

have a contribution of: O(4(n-1)(2**(m/2))/((2n)**m))
which reduces to:

$$O\left(\frac{4(n-1)}{(2^{**}(m/2))(n^{**}m)}\right)$$

As m grows, this contribution is less than one. Now we look at words with k occurrences of an exponential pair (where k<m/2). There are at most C(m,k) ways to choose where to place the k pairs (in fact, this is a rough upper bound.) Assuming no cancellation in these cases (to simplify the terms), those k pairs will cause at most 2**k growth (by lemma E.) Since there are m-2k generators remaining to fill in the braid word, we multiply by (2n)**(m-2k) to give all the cases. This also is a rough upper bound because we should disregard the exponential pair mentioned above but that would give ((2n)**2 - 1)**((m-2k)/2) which asymptotically is the same. So we have the following contribution to the general case due to the exponent pair occurring k times cases (assuming all other action is linear):

$$O\left(\frac{4(n-1)C(m,k)2^{**}k((2n)^{**}(m-k))}{(2n)^{**}m}\right)$$

which reduces to:

$$O\left(\frac{4C(m,k)}{(n^{**}(k-1))}\right)$$

This approximation does express the growth pattern for very small m (such as how f becomes smaller as n grows while m remains fixed.) For larger m (such as m>10), this approximation is totally absurd because we left out the partial-commutativity effect, the effects of cancellation (anti-exponential on exponentially large strings), and finally that even if the exponential-pairs exist, if they cause growth in one string, then the next braid transformation might (very probably) switch that string out of the TUPLE slots which will be the range of the next exponential pair, thus the 2**k is a tremendous over-estimation. These three factors (ie. partial-commutativity, cancellation, and slot-switching) would probably give the sub-quadratic growth behavior. Apparently, no analytic mathematical model lends itself adequately to expressing all these three properties simultaneously.

Artin's Algorithm

The combinatorial approach used above could be refined to approximate the slot-switching but the other two properties are much harder.

Warning: at first, recurrence relations could be written which match the pattern of the relators. After characterizing all of them, the problem unfolds: any recurrence system assumes a unique path (or at least equally valued paths) to an initial value. Unfortunately, cancellation (primarily, it's non-monotonic effect) almost always negates this path criterion. Hence, this ostensibly general method of combinatorially dealing with presentations fails. Even if this was not the case, the recurrence relations would almost always be too complex to prove useful. The same holds true for a conditional probabilistic version of the above (unless very rough approximations are taken which, in turn, would become worthless estimates.) Computations were carried out along these lines with no success.

II.D An Interesting Note

Garside's algorithm's behavior is strongly dependent upon the number of strands n (or generators.) It blows up very quickly because of it.

Artin's algorithm has the same worst case for all n>2. Not only does n not affect it's worst case but Artin's average case behavior requires less storage as n increases (due to the spreading of the growth effect.)

III.0 Garside's Word Problem Algorithm

Garside's algorithm for the word problem for braid groups was first presented in Garside[1965] and later published in Garside[1969].

III.A Definitions for the Garside Algorithm

Def: The product of successive generators is denoted as pi(m) = a1*a2*...am.

Def: The fundamental word of braid group B(n+1) is the word FW=pi(n)*pi(n-1)*...*pi(1).

Example: FW for B(4) is 123121.

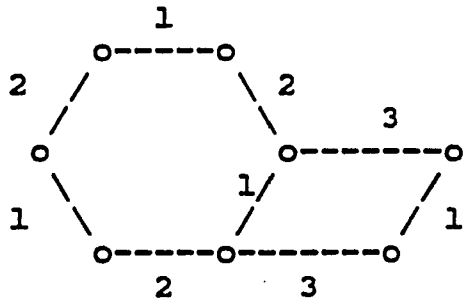
Def: The reflection of a generator g(k) is a generator Ref(g(k)) = g(n+1-k).

Def: Two positive braid words (w1 and w2) are positively equal iff w1 can be transformed into w2 by applying relations with only positive relations (ie. relations of the form a(i)a(j)=a(j)a(i) and a(i)a(i+1)a(i)=a(i+1)a(i)a(i+1).)

Garside's Word Problem Algorithm

Def: A word-graph or word-diagram of a positive braid word W is a subgraph of the Cayley graph which characterizes all words positively equal to W. The word graph has a source (or origin) and a sink. The wordgraph is a directed acyclic graph. Its' edges (or links) are labelled with the positive generators of B(n+1).

Example: The positive word 2123 has word-graph:



The three positively equal words are 2123, 1213, 1231. By convention, the source is the leftmost node and the sink is the rightmost.

Def: The spine of a word diagram is the path from source to sink labelled by the initial braid word. Occasionally, according to the context, the term spine will also be used in reference to any specific path with the property being discussed at the time.

III.B The Xi-Theorem for the Garside Algorithm

A part of the Garside algorithm is missing in all previous descriptions of it. In the segment for handling non-positive words, Garside states that for any braid generator ai, there exists a positive word Xi such that:

ai' = (Xi) * (FW') Garside proves such an Xi exists but his proof is not constructive. Later papers don't even make a suggestion as to what Xi is. The following theorem should resolve this situation:

Thm: If ai is the ith braid generator, then ai'=(Xi)*FW' has (as a uniform, positive word) solution:

Xi = pi(n)*pi(n-1)*...*pi(n-i+2)*a2
a3...a(n-i+1)*p(n-i)*...*p(1).

Proof: Let Xi= ai'*FW.

By successive factoring, we get:

Xi= ai'*pi(n)*pi(n-1)*...*pi(n-i+2)*
a1*a2*...*a(n-i+1)*...*p(1).

Xi= ai'*pi(n)*pi(n-1)*...*pi(n-i+3)*

a1*a2*a3*...*a(n-i+2)*a1*a2*...*a(n-i+1)*...*p(1).

Now we let a1 commute over a3*...*a(n-i+2), giving:

Xi= ai'*pi(n)*pi(n-1)*...*pi(n-i+3)*
a1*a2*a1*a3*...*a(n-i+2)*a2*...*a(n-i+1)*...*p(1).

But by the relator a1'*a2'*a1'*a2*a1*a2, we have

Xi= ai'*pi(n)*pi(n-1)*...*pi(n-i+3)*
a2*a1*a2*a3*...*a(n-i+2)*a2*...*a(n-i+1)*...*p(1).

which compacts to:

Xi= ai'*pi(n)*pi(n-1)*...*pi(n-i+3)*
a2*pi(n-i+2)*a2*...*a(n-i+1)*...*p(1).

This proves the initial step of an induction proof of the following lemma:

Lemma: pi(n-i+k+1)*ak = a(k+1)*pi(n-i+k+1). This lemma is Lemma 6 on page 20 of Garside's thesis[1965]. The proof is similar to the one above. This process of moving ak to the left and incrementing k by 1 finally gives:

Xi= ai'*ai*pi(n)*pi(n-1)*...*pi(n-i+3)*

pi(n-i+2)*a2*...*a(n-i+1)*...*p(1).

Xi= pi(n)*pi(n-1)*...*pi(n-i+3)*
pi(n-i+2)*a2*...*a(n-i+1)*...*p(1).

which is the desired result.

End of Proof

Example: 2' = (1234)(23)(12)(1)

III.C The Garside Algorithm

Step 1: Read Braid Word W = a1a2...am
where each ai is a generator;

Step 2:

```
/*Loop to convert negative generators into       */
/*     positive words with negative               */
/*     fundamental words and shift fundamental    */
/*     words to the left (as a counter, count)    */
/*Record number of negative generators in count   */

COUNT = 0;
WORDPASS: DO  I = LENGTH( BWORD) TO 1 BY -1;
```

```
   IF THE GENERATOR I IN BWORD IS AN INVERSE GENERATOR,
   THEN
        INVCASE : DO;
        IF (COUNT IS ODD)
        THEN REPLACE GENERATOR I WITH POSITIVE WORD Xi;
        ELSE
          DO;
          SET Xi = POSITIVE WORD CORRESPONDING
                                   TO GENERATOR I ;
          SET K = POSITION OF FIRST INVERSE GENERATOR
                  OCCURRING IN BWORD BEFORE I ;
          SUBSTR(BWORD,K+1,I-K) = REFLECTION(
             CONCATENATE(SUBSTR(BWORD,K+1,I-K-1), Xi) );
          END;
        COUNT = COUNT - 1;
        END INVCASE;
   ELSE IF (COUNT IS EVEN) THEN DELETE
SUBSTR(BWORD,I,1);
            /* IN THE ABOVE, WE DO DELETIONS TO AVOID */
            /* DUPLICATION OF GENERATORS MOVED IN THE */
            /* REFLECTION STEP */
END WORDPASS;
```

```
Step 3: COMPUTE G = WORDDIAGRAM( BWORD );

Step 4:
/*  SELECT A PATH WITH THE MAXIMAL NUMBER OF*/
/*       FW's, STARTING AT THE CONSECUTIVE   */
/*       ORIGIN  OF WORDDIAGRAM OF BWORD     */

SET POINTER PTR TO THE ORIGIN OF G.


LOOP
    SCOUT = PTR ;/*SCOUT will scout ahead to see if a*/
                 /*   whole FW starts at node PTR    */
    MATCH = TRUE;
    DO  I = 1 TO LENGTH( FW ) WHILE ( MATCH );
         IF A LINK EXISTS WITH LABEL = GENERATOR I OF
FW
           THEN SCOUT = LINK(SCOUT);
           ELSE MATCH= FALSE;
    END ;
    IF (MATCH = TRUE) THEN
       DO;
        PTR = SCOUT;
        COUNT = COUNT + 1 ;
       END;
```

```
UNTIL ( NOT( MATCH) );
```

Step 5:

```
/* SELECT PATH STARTING AT PTR AND ENDING AT THE END */
/*     WITH THE SMALLEST LABELS AT EACH STEP .       */
/* THE LABELS OF THIS PATH CONSTITUTE THE BASE WORD  */
/*     OF THE REMAINDER                              */
MINPTR = PTR;
BASEWORD = '';
DO WHILE (MINPTR NOT-EQUAL-TO 0);
     SELECT LINK (STARTING AT PTR) WITH SMALLEST LABEL;
     NAME THAT LINK  MINLINK AND THAT LABEL MINLABEL;
     MINPTR = MINLINK(MINPTR);
     BASEWORD = CONCATENATE( BASEWORD , MINLABEL);
END;
```

Step 6:

```
PRINT 'THE GARSIDE FORM OF WORD IS  FW**',
                    . COUNT,BASEWORD;
IF (COUNT = 0  AND  BASEWORD='')
      THEN  PRINT 'THE WORD IS THE IDENTITY.';
      ELSE  PRINT 'THE WORD IS NONTRIVIAL.';
```

III.D Turing Machine of Variant of Garside's Algorithm

A variant of Garside's algorithm can be constructed which will be proven to operate in non-deterministic linear time on a Turing Machine. Instead of computing the whole word-diagram, start with the initial word and by calling oracles, transform it into a word with the maximal number of copies of the fundamental word in front.

The Turing Machine is:

```
TURING_MACHINE: PROCEDURE;
Step1:
Declarations: Tapes:  Input , Output , Work , Work2
                      Oracle-Substitution (or OS) , FW
              Counters:  FWCOUNT , Oracle-Position (OP)

Comment: After each call to the Oracle, OP will have
    the position of the desired substitution, while OS
    will have the substitution string (such as 212.)
Comment: Input will have the format:  X input-word Y.
Comment: Tape FW will have only a copy of the
    fundamental word, enclosed in X and Y. X and Y are
```

```
    end-markers.
Comment: Input-head will start at rightmost generator.

Step2:
/* Put positive version of input word on Work tape.*/
       DO  UNTIL  (Input-cell=X)
           IF (Input-cell = negative-generator) THEN
              /* Next step computes Xi-word.        */
              Copy fundamental word from FW to Work2
               except for the Input'th occurrence of 1;
              Move FW head left until X;
              Copy Work2 word to
                        the left side of Work word;
              IF (Counter is even) THEN
                 Move Input head left;
                 DO UNTIL (Input-cell =
                           negative generator or X);
                   Copy Reflection(Input
                              generator) to Work;
                   Move Work  head left;
                   Move Input head left;
                 END DO;
                 IF (Input-cell = X) THEN;
```

```
            ELSE
                Move Input head right.
            END IF;
            COUNTER = COUNTER - 1;
        ELSE
            Comment: positive generator case.
            IF (Counter is odd) THEN
                Copy Input-cell generator to Work;
                Move Work  head left;
            END IF;
        END IF;
        Move Input head left;
    END UNTIL;

Step3: /*Repeatedly call the oracle in main loop.*/
       /*This will simulate the creation of      */
       /*                    a word diagram.     */
       Place X and Y markers to the
             left and right of the Work tape word;
       CALL ORACLE(OS,OP);
       /* Oracle returns values on     */
       /*       OP counter and OS tape.*/
       DO UNTIL (OP = 0);
```

```
        /*OP = 0 means no more substitutions needed.*/
        Move Work head to position OP.
        Copy OS word starting in
                    that position on Work tape;
        /*The above statement will write over    */
        /* previous contents in those positions.*/
        /*So, tape OS may have 212 to */
        /*              overwrite 121. */
        CALL ORACLE(OS,OP);
      END UNTIL;

Step4&5:  Comment: Pattern Match with FW
      Reset FW head to leftmost;
      Reset Work head to leftmost;
      DO UNTIL (OUTPUT-cell = F   OR
                WORK head points to Y) ;
          DO UNTIL (FW-cell = Y OR  OUTPUT-cell = F);
             IF (FW-cell generator =
                 Work-cell generator)
             THEN
                  Move FW head right;
             ELSE
                  OUTPUT-cell = F;
```

```
            END IF;
        END UNTIL;
        Reset FW head back to leftmost generator;
    END UNTIL;

Step6: IF (OUTPUT-cell not = F) THEN OUTPUT-cell = T;
       Comment: T means word is identity.
       HALT.

END TURING-MACHINE;
```

III.E Analysis of the Garside Algorithm

We will analyze the Turing machine model (for preciseness's sake.)

Step 1 is the initial input assumption.

Step 2 makes a pass over the input tape and produces a positive word on the Work tape. In this mapping, each positive generator in the input produces one positive generator in the work tape. Each negative generator will be replaced by a positive word of length $L = n(n+1)/2 - 1$. Since n doesn't vary with length m

(of the input), L is constant relative to m. Hence, the input generators produce a work tape word requiring linear space. Note: Counter counts the number of negative generators, therefore only requiring log space.

Step 3 depends on requires no extra space because all substitutions are of the same length as what they are substituting.

Note: The standard Garside algorithm differs with the variant TM version only at this main point (with respect to complexity.) The remainder of this chapter is devoted to this point: How large can the word diagrams grow!

Steps 4 and 5 are pattern matchs along the length of the work tape. In the Turing version, one matching process tests to see if the work tape is really just multiple copies of the fundamental word. In the standard version, Step 5 has an extra pattern match to compute the minimal tail (or remainder) word; however, this pattern match requires linear time and no extra space.

Step 6 is the one character output.

Note: Except for Step 3, the whole algorithm (in either version) would run in linear time.

We have just proven:

Theorem: A variant of the Garside algorithm (for solving WP(B(n+1))) runs in non-deterministic linear space on a Turing Machine.

Savitch proved for Turing Machines that nondeterministic linear space problems can be solved in deterministic quadratic space (ref. Harrison[1978], page 286.) Hence, the following corollary:

Corollary: A variant of the Garside algorithm (for solving WP(B(n+1))) runs in deterministic quadratic space on a Turing Machine.

Note: this problem could still very likely take exponential time because the decisions (that are made via oracle) are complex ones, some substitutions appearing to have no effect till many steps later.

III.F Analysis of Word-Diagram Growth

III.F.1 Three Complexity Measures

Three measures are of primary concern in characterizing the complexity of a word-diagram:

1. number of nodes (of the graph), termed NUMNODE.
2. number of edges, termed NUMEDGE.
3. number of paths from the initial node to the final node, termed NUMEQ. Equivalently, NUMEQ is the number of positively equal words.

III.F.2 Relationships between NUMEQ and the other two

Theorem: There exists a sequence of words (in B(n+1), for n>3) which have exponentially growing NUMEQ and yet their word-diagrams have NUMNODE values only growing linearly (in the length of the word.)

Proof: Take braid words of the form w(k)=(1322)**k. Note that 13 has an equivalent in 31 but neither commutes over 2. Since two 2'a occur together, no production like 323=232 or 121=212 can be applied. Hence, the word-diagram (resembling an open

necklace) has a linear NUMNODE growth rate but NUMEQ=2**k. (QED)

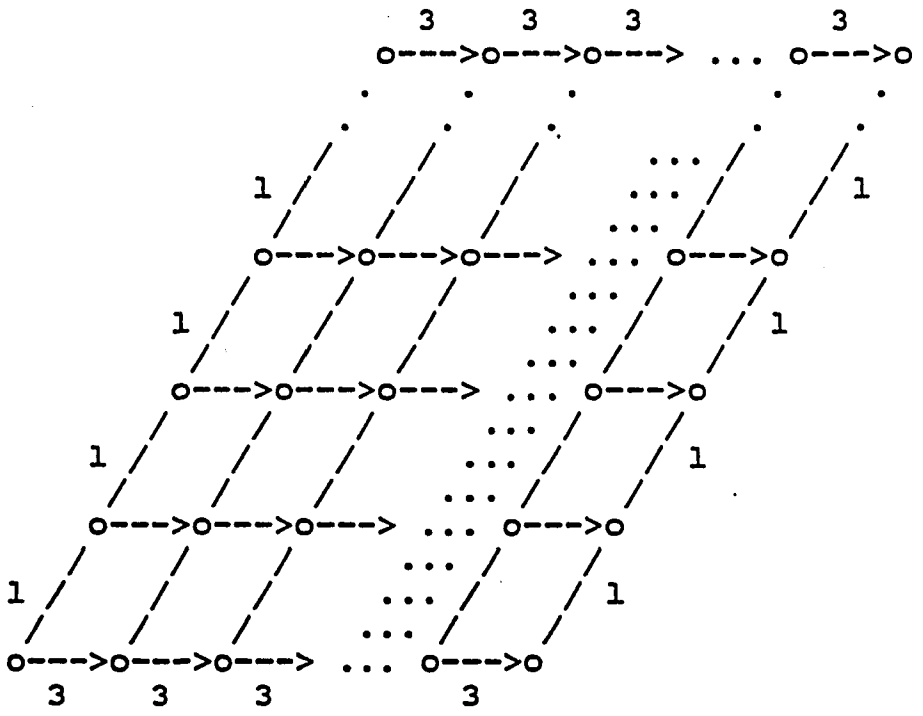
Theorem: There exists a sequence of words (in B(n+1), for n>3) which have exponentially growing NUMEQ and yet their word-diagrams have NUMNODE values only growing quadratically (in the length of the word.)

Proof: Take braid words of the form w(k)=(13)**k. The length of w(k) grows linearly in k.

NUMEQ(w(k+1)) = NUMEQ(w(k)13) > (2)*NUMEQ(w(k)) because, at the least, by partial commutativity, 13 can have the form 13 and 31. Note that this assumes independence of 13-pairs, which weakens the bound; the exact value of NUMEQ(w(k)) is C(2k,k).

Now for NUMNODE, we can draw the word-diagram:

Word Diagram for w(k)



Every extra 13 in w(k) will add a new row and column, giving NUMNODE(w(k))=(k+1)**2 and NUMEDGE=2k(k+1).

QED

So far, we have only shown a linear and quadratic NUMNODE with exponential NUMEQ (using commutativity.) Now let's exhaust the power of commutativity and see how large NUEQ can become by it. Note: First we will

remove a restriction by allowing n to also vary as well, giving the full result.

Theorem: Let

V = 1**(m/j)*3**(m/j)*...*(2j-1)**(m/j). We then get NUMEQ(V) > (j!)**(m/j) where 2j-1 < n+1.

Proof: There are j! ways of permuting (1,3,...,2j-1). If we treat V as (m/j) independent copies of (1,3,...,2j-1), the results follow immediately. (QED)

So NUMEQ(V) can produce larger exponential functions provided n can increase.

If we remove the independence constraint, we have the exact value:

Theorem: NUMEQ(V) = (m!)/ (((m/j)!)**j).

Proof: Refer to page 12 of [Liu, C.L.] (ie. j categories of objects, (m/j) of each category, distributed into n distinct cells (alternately, can use multinomial theorem or induction on formula.))

(QED)

As a side-theorem, we get from the above two: m!>((m/j)!)**j*(j!)**(m/j).

III.F.3 The relation between NUMNODE and NUMEDGE

We have the bound following bound in general:

NUMNODE-2 < NUMEDGE < NUMNODE*n

where n = number of generators.

Proof: Out-valence of word-diagram is at most n.

(QED)

The thinnest (or smallest) word-diagrams are the linear graphs. They have the smallest ratio, between NUMNODE and NUMEDGE. For these graphs, NUMEQ=1.

Theorem: The only words with linear graphs as word diagrams are of the following forms:

1**k, 2**k, ..., n**k, and any prefix or suffix (or center-word) of the words

u = (1**(k1)2**(k2)3**(k3)...n**(kn)) or

w = (n**(kn)...3**(k3)2**(k2)1**(k1)) or

u(1)w(1)u(2)w(2)...u(p)w(p) where p,k,kn,k1 are greater than or equal to one and adjacent u's and w's have a common generator at their boundaries.

Proof: By exhaustion of prefix and suffix cases. QED

III.F.4 Some Computed Combinatorics of Word-Diagrams

FW	121	123121	1234123121
NUMNODE	6	24	120
NUMEDGE	6	36	240
NUMEQ	2	16	768

Note: 12312 had only 12 nodes, 15 edges, and 5 paths.

Note: 123412312 had only 60 nodes, 108 edges, and 168 paths. Compare these with the FW statistics above. The statistical patterns of Xi word-diagrams merits further analysis. Complexity of the positive word translation depends upon this.

Asymptotics of FW**k in B(3)

Word-Diagram Statistics for (121)**k

k	NUMNODE	NUMEDGE	NUMEQ	RATIO
1	6	6	2	3.1
2	19	24	8	2.5
3	48	66	38	2.3
4	109	156	196	2.16
5	234	342	1062	2.08
6	487	720	5948	2.04
7	996			2.025
8	2017			2.0138
9	4062			2.007
10	8155			

RATIO means: $\dfrac{\text{NUMNODE((121)**(k+1))}}{\text{NUMNODE((121)**k)}}$

Note that the asymptotic behavior of RATIO tends toward 2.00 and that even the fractional residues appear to be decreasing by a factor of 2. No reason has been found. This is one of very few examples of exponential growth of word-diagrams with respect to word length.

III.F.5 A Worst Case for Word-Diagrams: Fundamental Words

Lengthy enumerations show that the fundamental words result in the largest word diagrams. For greater word-lengths, multiple copies of the fundamental word are the worst case. Though these tests were carried out upto word lengths of 20 (in B(3)) and less in B(n+1) (n < 10), the behavior seems to hold true; however, this remains a conjecture for the untested cases.

Conjecture: The word-diagram for the fundamental word FW(n) has growth rate NUMNODE(FW(n)) = Factorial(n).

In fact, the nodes of the word-diagram for the fundamental word can be labelled with permutations, exactly matching the symmetric group S(n). The generators labelling the edges correspond to transpositions exactly. Generator i corresponds to transposition (i i+1).

The distance from the origin corresponds exactly to the number of transpositions in each permutation. One can show this from the patterns and factorizations.

The final node of the word-diagram has the permutation:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & \dots & n \\ n & n-1 & n-2 & n-3 & \dots & 1 \end{pmatrix}.$$

As expected, this last node will have length (n)(n+1)/2, the distance of the word-diagram.

Very Important Note: A large word-diagram doesn't just affect the word-diagram size as is; every positively equal word represented will each contribute the whole size also. So, for the above, the effect of this theorem will be Factorial(n)*(NUMEQ(FW)), the latter term being extremely large.

III.F.6 Partial Results Toward Factorial Conjecture

Theorem: The node with distance=1 (from the source) include 1,2,3,...,n (ie. the single transpositions.)

Proof: In the last step of the Xi theorem (constructing the positive word corresponding to a negative generator), FW is expressed with potentially any generator in front.

(QED)

Theorem: The word graph for the fundamental word is symmetrical (with respect to origin and final node.)

Proof: Garside's thesis, theorem 3(ii) states that "Rev(fw) is positively- equal to fw". (QED)

Other partial results can be developed likewise. Note that even though Coxeter and Moser describe the relationship between the braid group and S(n), the word-diagram's restrictions prevent any apparent application of such results.

An incremental word diagram theory seems like a fruitful endeavor but it is too complex due to the large number of cases. Perhaps using recurrence relations or inclusion exclusion may help. No operator theory for graphs has been developed yet. Prefixes and suffixes don't help much in terms of the strings. String transformations seem to resonate throughout a

string rather than remain local or affect one direction.

III.F.7 Important Corollaries to the Factorial(n) Conjecture

Note: If the Factorial(n) conjecture holds for the fundamental word, then the breadth (or fatness) of the word-diagram will be at least Factorial(n)/((n)(n+1)/2), which is about 2*Factorial(n-2). This will have considerable importance with the next subchapter on asymptotics.

Note: The above breadth condition would have strong ramifications upon the complexity of the variant algorithm we gave for Garside's algorithm. If we combine the next result (in the asymptotics subchapter) with the above results on 2*Factorial(n-2), then the number of non-deterministic decisions needed to transform one arbitrary path through the word-diagram into the maximally-FW prefixed path will be the average time complexity of the variant-Garside algorithm. Furthermore, the maximal number of steps to do this transformation will be the worst case behavior of the

variant-Garside algorithm. In fact, the Garside algorithm's complexity will also be determined by it. Many powerful results just depend upon proving the conjecture (ie. Factorial(n).)

Note: Attempting to write a backtracking algorithm to simulate the oracle seems impossible unless much space is wasted (upto quadratic space) or time (exponential at worst) or both. Even on small examples, the only way found was brute force.

III.F.8 Asymptotics of Word Diagrams

Asymptotic Theorem: As word length tends toward infinity then the word-diagram has very high probability of having the breadth of at least the fundamental word (throughout the whole diagram.)

Proof: As word length tends toward infinity, the probability of a word having at least one copy of the fundamental word is one. This is especially true because the transformation of the negative generators into positive words results in words which are almost the fundamental word.

By commutativity, one copy of the fundamental word can commute the whole length of the word-diagram. Therefore, the word diagram will have have the breadth of the word-diagram for the fundamental word throughout the length of the word-diagram. (QED)

This is not a very sharp bound because it is linear with respect to the length of the braid word. It does not express the "central bulge" of most word diagrams.

Theorem: If we fix the length of a positive word, then (after a point), changing the generator structure of the word will produce no larger word-diagrams.

Proof: If we produce the largest word diagram of a given length, then it can have finitely many positive-relator substitutions with the number of distinct generators it is using. This is true because no positive relation (or substitution) can create a new generator. In simpler terms, if we have 143413133443, then by using positive substitutions, we can never get generator 2 in there.

(QED)

So for a given length category, there is a maximally-sized word-diagram. We know this is certainly not true for most algebraic systems.

III.F.9 The Word Diagram as a Poset

Theorem: A word diagram (of a positive word in B(n+1)) is the graph of a partial order.

Proof: It suffices to show that the graph is an acyclic digraph (via the following proof by contradiction.) If the word graph has a cycle, then it would characterize an infinite set of words. Since all words in the word graph have equal length, there can only be finitely many of them. Hence the contradiction arises. (QED)

Conjecture: A word diagram (of a positive word in B(n+1)) is a lattice.

Though this is a conjecture, a large sample of computer generated word diagrams were tested with positive results. Weaker theorems can be proven by the relators but are too localized.

Note: One possible proof of this could be developed by the permutational representation of the word diagram in conjunction with the factorization conditions needed for a lattice.

III.F.10 Construction of the Worst Case Complemented Poset with Bounded Maximal Valence

In the process of studying word-diagrams, I have produced a beautiful example of recursively defined category of worst-case complemented posets with bounded maximal valence. The beauty of it lies in the proof that it is the worst case. The proof emerges recursively as the poset grows.

Further details can be provided upon notice.

III.G The Word Diagram: Computational Aspects

There are a number of possible ways to compute the word diagram of a positive word. In this chapter we deal with some of them.

Garside's Word Problem Algorithm

III.G.1 Simple Closure Algorithm

Pack the initial spine of the word diagram (ie. the initial braid work in the lowest (or first) nodes in the linked list. Then, sequentially traverse this list repeatedly until no new nodes can be created. At each node traversed, apply all the relevant generators that have not produced an out-link.

This algorithm's space utilization grows monotonically, hence space and time complexity are very closely related. It appears to be fast and efficient in space usage. For two generators, it works perfectly and is extremely fast. Unfortunately, for larger n, it wastes much space and (due to the looping for closure) wastes much time too. In fact, it becomes unnecessarily exponential. The key to this is that it generates nodes in one loop before it computes the edge-closure of the other nodes previously created in the same traversal pass.

The real cause of this redundancy is a link that is duplicated. Once a link is duplicated, the subgraphs to be generated out of it are duplicated. So it grows exponentially as each single duplicate link

reproduces whole subgraphs. On long strings, the effect is devastating.

This algorithm is the first one implemented and therefore the one used in the appendix of this text. In this program, the words 14364 and 14346 differ by one redundant node, due to creation of a node before an edge-closure test. This is the first case of it. Though a little too large to duplicate, the diagram actually shows where the redundancy occurs.

To remove the redundancy, node deletion and merging should occur but the tests for duplication are to expensive in time for this model and the redundancy is found out too late.

Statistics for the Redundant Word-Diagram Size

(using the Simple Closure Algorithm)

	NUMSTRAND 4	5	6	7	8
Length=5 AVE.	9.7	11.09	12.30	13.59	14.68
MAX.	12	18	18	25	25
SAMPLE	2100	3800	10000	4000	8000
Length=10 AVE.	38.7	45.0	55.6	68.4	81.6
MAX.	112	129	203	247	289
SAMPLE	2300	5000	11900	6000	8000
Length=15 AVE.	121.3	133.9			
MAX.	620	>1000			
SAMPLE	2500	1000			

Note: Length refers to word length.

III.G.2 Simple Creation-Closure Phase Algorithm

This is another very time wasteful policy, far worse than the previous one (appearing to be quadratic in the previous one.) No space is wasted and the exact word-diagram is produced. This version of the algorithm was also programmed.

Basically, take the previous version but each time a node is created, a whole edge-closure traversal must be done. After that only can a next node be created.

Though this seems like brute force, the consistency of the algorithm is retained with minimal space usage.

Statistics for the Exact Word-Diagram Size

(using the Simple Closure Algorithm)
NUMSTRAND=3

(Word length=1)

	l=5	l=10	l=15	l=20	l=25
AVE.	9.05	35.8	108.7	292	745
MAX.	12	63	234	741	2528
SAMPLE	5600	3800	1250	2500	21000

III.G.3 Backpath-Closure and Creation Algorithm

Same as the first policy but allow for the merging of backward relation paths. By merging them, some redundancies can be avoided but I don't know if all of them can be removed as in the second policy. This has not been tested as a program.

III.G.4 Creation-Deletion Algorithm

This is a non-monotonic approach where the nodes are created as in the first process except where a redundancy occurs, a redundant path is selected for deletion. Which one of the paths is up to the programmer; however, be very careful not to enter an infinite loop of regenerating and deleting the same path. Though great for efficiency, this algorithm is hard to test for correct behavior.

IV.0 Burau Representation Algorithm(?) for B(n+1)

IV.A Burau Representation

Def: A group R is a representation of a group G if there exists a homomorphism h:G-->R.

Def: A representation is faithful if the homomorphism is one-to-one. Burau[1936] demonstrated a potentially faithful matrix representation for B(4) given below.

The Burau representation is a matrix group consisting of matrices with entries from the integer coefficient polynomials in variable t and 1/t where t is a rational. The group's elements are formed by taking all matrix products of the following six matrices:

Burau representation of B(4) over Z[t,t**(-1)]

where t is in Q (ref. Birman(1974)

$$s(1) \longrightarrow \begin{vmatrix} -t & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \qquad s(1)' \longrightarrow \begin{vmatrix} -1/t & 1/t & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

$$s(2) \longrightarrow \begin{vmatrix} 1 & 0 & 0 \\ t & -t & 1 \\ 0 & 0 & 1 \end{vmatrix} \qquad s(2)' \longrightarrow \begin{vmatrix} 1 & 0 & 0 \\ 0 & -1/t & 1/t \\ 0 & 0 & 1 \end{vmatrix}$$

$$s(3) \longrightarrow \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & t & -t \end{vmatrix} \qquad s(3)' \longrightarrow \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & -1/t \end{vmatrix}$$

where s(i) = ith standard braid generator

An open problem that remains outstanding (despite many efforts) is to show that the mapping is faithful. As a matter of convenience, it shall henceforth be called the Burau Conjecture.

To prove it true, we need to show that a braid word is the identity if it's representation (a matrix product) is the identity matrix.

Example: 1'3'13 = identity would then require the following product in the matrices of polynomials in the Burau representation:

$$\begin{vmatrix} -1/t & 1/t & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & -1/t \end{vmatrix} \begin{vmatrix} -t & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & t & -t \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

IV.B Burau Representation Algorithm(?)

Burau Algorithm for B(4):

```
Step0: Burau: Procedure;

Step1: Declare (MATRIX,WORK) as a 3 by 3 matrix,
          each entry of which is a 1-dim. array
          of coefficients of a polynomial in
          t and t**-1; these coefficients have
          indices (-P to +P).
       READ BRAID WORD W=a1a2...am;

Step2: MATRIX = IDENTITY-MATRIX;

Step3: MAINLOOP: DO  IPOS = M TO 1 BY -1;
          GEN = GENERATOR IN POSITION IPOS IN W
          CALL PRODUCT(GEN,MATRIX);

Step4: END MAINLOOP;
```

```
Step5: IF (MATRIX = IDENTITY-MATRIX)
       THEN  PRINT 'TRUE; MATRIX IS IDENTITY.'
       ELSE  PRINT 'FALSE; MATRIX IS NONTRIVIAL.'

Step6: END BURAU;

Step7: /* THE REMAINDER OF THIS PROGRAM IS THE */
       /*    SUBPROCEDURE PRODUCT.             */
       /* THIS PROCEDURE COMPUTES:             */
       /*    MATRIX = REPRESENTATIVE(GEN)*MATRIX*/
       PRODUCT: PROCEDURE(GEN , MATRIX);

Step8: DECLARE AND STORE BURAU REPRESENTATION AS
       SIX MATRICES:  BUR(6).ELEMENT(3,3).

Step9: RESET WORK TO ZERO MATRIX;

Step10: /*DO STANDARD MATRIX PRODUCT TRIPLE LOOP*/
       ILOOP: DO I = 1 TO 3;
        JLOOP: DO J = 1 TO 3;
         KLOOP: DO K = 1 TO 3;

Step11:    /*PICK THE ENTRY OF THE BURAU REP. IN */
           /* POSITION (I,K) TO MULTIPLY WITH.   */
           TENTRY = BUR(GEN).ELEMENT(I,K);
```

```
Step12:   /* MAIN DO-CASE HANDLES (I,K)*(K,J)    */
          BIGCASE:DO CASE;

Step13:    Case1: Caseif ( TENTRY = 0 );
            /*do nothing, really adding 0-vector*/
           End Case1;

Step14:    Case2: Caseif ( TENTRY = 1 );
            /*add two polynomials*/
            /*add bur(gen).ele(i,k)*matrix(k,j)   */
            /*      to  work(i,j).                */
            /* Note: bur(gen).ele(i,k)=1          */
            WORK(I,J) = ADDVEC(MATRIX(K,J),WORK(I,J));
           End Case2;

Step15:    /*before doing any more cases, define  */
           /* procedure ADDVEC     --- a function  */
           /* which adds two vectors and returns  */
           /* their sum as the value of ADDVEC.   */
           ADDVEC: PROC(VECTOR1 , VECTOR2);
              DO  KA = -P TO +P ;
                SUM(KA) = VECTOR1(KA) + VECTOR2(KA);
              END;
```

```
        RETURN (SUM(-P : +P));
    END  ADDVEC;

Step16: Case3: Caseif ( TENTRY = -1 );
        /*subtraction between two polynomials*/
        /*add   bur(gen).ele(i,k)*matrix(k,j)*/
        /*       to work(i,j).                */
        /* Note: bur(gen).ele(i,k)=-1          */
        WORK(I,J)= ADDVEC( -MATRIX(K,J),WORK(I,J));
    End Case3;

Step17: Case4: Caseif ( TENTRY = t );
        /*add two polynomials*/
        /*add bur(gen).ele(i,k)*matrix(k,j)   */
        /*     to  work(i,j).                 */
        /* Note: bur(gen).ele(i,k)=t          */
        /* t*poly(t) = right shift poly(t)    */
        WORK(I,J) = ADDVEC(SHIFT('RIGHT',
                        MATRIX(K,J)),WORK(I,J));
    End Case4;

Step18: /*before doing any more cases, define */
        /* the vector-valued function SHIFT   */
        /* which just shift coefficients of   */
```

```
/* a poly. in an array by 1 position. */
SHIFT:PROCEDURE(DIRECTION , VECTOR);
   IF (DIRECTION = 'LEFT') THEN
     LEFTCASE: DO;
      DO Q = -P+1 TO P;
         VECTOR(Q-1) = VECTOR(Q);
      END;
      VECTOR(P) = 0 ;
     END LEFTCASE;
   ELSE
     RIGHTCASE: DO;
      DO Q = -P TO P-1;
         VECTOR(Q+1) = VECTOR(Q);
      END;
      VECTOR(-P) = 0 ;
     END RIGHTCASE;
   RETURN(VECTOR);
END SHIFT;

Step19:  Case5: Caseif ( TENTRY = -t );
         /*Subtraction of  two polynomials*/
         /*add bur(gen).ele(i,k)*matrix(k,j)  */
         /*     to  work(i,j).                */
```

```
          /* Note: bur(gen).ele(i,k)=-t          */
          /* -t*poly(t) = right shift -poly(t)   */
          WORK(I,J) = ADDVEC(SHIFT('RIGHT',
                        ( -MATRIX(K,J))),WORK(I,J));
          End Case5;

Step20:   Case6: Caseif ( TENTRY = 1/t );
          /*add two polynomials*/
          /*add bur(gen).ele(i,k)*matrix(k,j)  */
          /*     to  work(i,j).                */
          /* Note: bur(gen).ele(i,k)=1/t        */
          /* 1/t*poly(t) = left shift poly(t)   */
          WORK(I,J) = ADDVEC(SHIFT('LEFT',
                         MATRIX(K,J)),WORK(I,J));
          End Case6;

Step21:   Case7: Caseif ( TENTRY = -1/t );
          /*add two polynomials*/
          /*add bur(gen).ele(i,k)*matrix(k,j)  */
          /*     to  work(i,j).                */
          /* Note: bur(gen).ele(i,k)=-1/t       */
          /* -1/t*poly(t) = left shift poly(t)  */
          WORK(I,J) = ADDVEC(SHIFT('LEFT',
```

```
                        -MATRIX(K,J)),WORK(I,J));
          End Case7;

Step22:   END BIGCASE; /*end the DO CASE  */
        END KLOOP;
       END JLOOP;
      END ILOOP;
     END PRODUCT; /* END OF SUBPROCEDURE */
```

Warning: This program is correct for WP(B(4)) iff the Burau conjecture is true. Evidence of the latter will be presented in a later subchapter.

If the Burau conjecture was not true, then this algorithm will accept (as identity braids) some non-identity braids. Either way, this algorithm is a valid sufficient condition for a 4-braid not being the identity.

IV.B.1 Complexity of the Algorithm

The space bound for this algorithm is:

O(2(9*2m)) = O(m) coefficients.

The nine arises from the number of polynomials. The first two arises from the fact that t*Poly(t) and 1/t*Poly(t) cause a two directional growth for each polynomial. The second two arises from the necessity of a Work array. Other intermediate computations require temporary space, bounded by 5m coefficients.

Each coefficient can grow in value (at worst) as 3**m (this is an upper bound which is clearly not tight.) To see this, note that at each matrix product step, each new polynomial will be the sum of three previous polynomials.

On a base three machine, each new matrix product requires each coefficient to expand by one more digit. On a base-k machine, the result still remains a constant growth at each product step, hence O(m) new bits per step.

So the space complexity is at worst bounded by O(m**2).

The time complexity is at worst bounded by m*9*3*2m additions and shifts. Since the additions occur over linearly growing coefficients, this would seem to give O(m**2*log(m)) but the shifts occur over linearly growing coefficients, so assuming a fixed-word-size machine, each shift will require linear time. So the final bound is cubic time.

Problem: For the average case behavior, the central issue involves how sparse are these matrices in general. During implemented runs, some sparse cases were observed (some with interesting patterns) but occurrence was infrequent.

The next section introduces a superior algorithm in terms of space requirements.

IV.C The Lipton-Zalcstein-Burau Algorithm

Lipton and Zalcstein[1977] proved that if a group is linear, then it's word problem is solvable in logspace.

If Burau's conjecture is true, then B(4) is a linear group. Lipton and Zalcstein[1977] proved part of their theorem by constructing a logspace algorithm for linear groups. Therefore, the Lipton-Zalcstein algorithm can "solve" WP(B(4)) in logspace (ie. "solve" in the sense of the Burau algorithm, except more efficiently.)

IV.D Word Problem for B(3) is Solvable in Logspace

B(3) has a faithful Burau representation; hence, it is a linear group. Therefore, the word problem for B(3) is solvable in logspace (ie. solvable in the absolute sense, without dependence on conjectures holding.) The algorithm is that of Lipton and Zalcstein[1977].

IV.E The Burau Conjecture: New Insights

Reversing the reasoning in the previous section, if a logspace algorithm for WP(B(4)) exists, then it would act as more evidence toward showing B(4) is a linear group. This would not prove the Burau conjecture because this can't prove that the Burau

representation is the right one (ie. the faithful one), as well as not proving even linearity. However, the logspace algorithm would add further evidence that the Burau conjecture is true. So far, there is a long history of papers building up evidence, as described in the next section. The nature of the conjecture is intrinsically complex.

A proof that no logspace algorithm exists for WP(B(4)) would result in the Burau conjecture being false. In a later chapter on Lisa's algorithm, this will appear to be the case. Despite many unsuccessful trials, an even newer "algorithm" is presently being constructed which has the logspace feature but is not completed yet (due to its' length.)

IV.F Previous Results Toward the Burau Conjecture

In general, Burau showed that B(n+1) has the representation:

$$\text{sigma}(i) = \begin{vmatrix} I(i) & 0 & 0 \\ 0 & A & 0 \\ 0 & 0 & I(n-i-1) \end{vmatrix}$$

where: I(k) = kxk identity matrix

$$A = \begin{vmatrix} 1-t & t \\ 1 & 0 \end{vmatrix}$$

Gassner[1961] gave a representation for the unpermuted subgroup of B(n+1).

Magnus and Peluso[1967] showed that the Burau representations of B(2) and B(3) are faithful.

Birman[1974] showed that the Burau representation for B(4) is faithful iff the group GM generated by the following two matrices a and b:

$$a = \begin{vmatrix} -t & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & -1/t \end{vmatrix}, \quad b = \begin{vmatrix} 1-t & -1/t & 1/t \\ 1-t**2 & -1/t & 0 \\ 1 & -1/t & 0 \end{vmatrix}$$

is free.

Birman[1974] also provided a complete review of the previous results but in a more general setting, as well as very powerful new results associating the categories of Magnus representations, the Alexander polynomial, and some decision procedures.

Magnus and Tretkoff[1980] proved that the linearity of Aut(F(2)) implies the linearity of Aut(F(n)).

Siegfried Moran[1980] showed that {a,c} generate a free group (where a and b were the matrices of GM and c=ab'.) Furthermore, he demonstrated that {a',cac} generate a free group, as well as other similar sets.

Dyer, Formanek, and Grossman[1981] showed that linearity of Aut(F(2)) is true iff B(4) is linear.

Craig Squier[1984] showed that a variant of the Burau representation is unitary and reduced the Burau conjecture to two others. Unfortunately, neither conjecture has been proven. They involve the relationship between kernels in a mappings involving the addition of relators of the form s(i)**k for all i (where k is fixed.)

Among other important recent papers are those of Lipschutz[1961], Gorin and Lin[1969], Tits[1972], and Dixon[1972].

IV.G Partial-Computation for Faithfulness Testing

By treating program Burau as a subroutine in a larger program for enumerating braids and also inserting the Artin algorithm as a subroutine, we have an procedure for recursively enumerating braids which act as counterexamples to the Burau conjecture.

```
/*Partial-Test for Faithfulness*/
--------------------------------
Step0: W=1;

Step1: LOOP FOREVER;

Step2:    CALL ARTIN(W);

Step3:    IF (ARTIN RETURNS FALSE) THEN
            /* nontrivial braid*/
             CALL BURAU(W);
             IF (BURAU RETURNS TRUE) THEN
               PRINT, W 'DISPROVES BURAU CONJECTURE';
               STOP;
```

```
            ENDIF;
         ENDIF;
Step4:   L = LENGTH(W);

Step5:   CALL NEXTWORD(W);
         /* this subroutine generates the next  */
         /* braid word W in a breadth first     */
         /* way, returning it in W.             */

Step6:   K = LENGTH(W);
         IF (L < K) THEN
            PRINT 'BURAU CONJECTURE IS TRUE FOR' ;
            PRINT 'ALL WORDS OF LENGTH = ' L ;
         ENDIF;
Step7: END LOOP /*forever*/;
       END-OF-PROGRAM
```

This program was run for all braid words of length upto and including nine but with no resulting counterexamples. Hence Burau's algorithm and the others based on the Burau conjecture work for braids of length nine. At the very least, it adds further evidence that the Burau conjecture may be true.

A proof that the Burau conjecture holds for all m if it holds for all m less than a certain k seems like a possibility.

IV.H Lie Ring Representation Algorithm for B(4)

S. Lipschutz[1961] characterized important subgroups of B(4) in terms of automorphisms of free rings. This alternate approach warrants further analysis.

V.0 Combing Algorithm for B(n+1)

V.A Braid Word Formalism

L. O. James[1971] devised an alternate notation for braids. In the L. O. James notation, a braid is represented as a finite sequence of integer pairs, the first number in the pair unsigned, the other signed. Each generator in the standard notation becomes one pair in the James notation. In the L. O. James notation, each strand's original input number is used throughout the notation.

So, (a -b) means strand a goes under strand b.

V.A.1 An Example of the James Notation

Starting with braid 1'2'1'212, we construct a vector and apply those transformations:

Vector	Transformation	Notation
1 , 2 , 3 , 4		
----------	1'	(1 -2)
2 , 1 , 3 , 4		
----------	2'	(1 -3)

```
 2 , 3 , 1 , 4
 ------------------  1'          (2  -3)
 3 , 2 , 1 , 4
 ------------------  2           (2   1)
 3 , 1 , 2 , 4
 ------------------  1           (3   1)
 1 , 3 , 2 , 4
 ------------------  2           (3   2)
 1 , 2 , 3 , 4
```

Note: The vector records the effects and the strands affected. The James notation then records the affected strands.

V.B More Definitions

Def: A braid is unpermuted iff the order of the output strands matchs the order of the input strand. In other words, the permutation resulting from the braid is the identity.

Example: . 123 is a permuted braid while 1111 and 1212'1'2' are not.

Def: A braid C is in combed iff it is unpermuted and it has the structure c1c2c3c4c5...cn where ck is a subbraid in which only the kth strand is permitted to move (or cross) over or under any other strand provided that the other strand has a higher strand number (ie. k+1, k+2, ..., n.)

Example: 1123322332 is a combed braid where c1=11 and c2=23322332.

Artin[1950] showed that, after free-reduction, the combed form of a braid is unique.

V.C The Combing Algorithm

V.C.1 Notes

The combing algorithm used takes a braid in standard notation and converts it into James notation. Unlike combing, I first test for permutedness (in fact this is done for free because the notation conversion involves a vector which already will compute the permutation.

Note: A necessary condition for a braid to be the identity is that the braid be unpermuted. This reduces time by a factor of Factorial(n).

Note: The core of the combing algorithm (described below) is really a two stack algorithm, where the finite control looks at the top two pairs on the stacks and applies productions accordingly. The first stack (W) starts with the initial braid and the algorithm halts when the first stack is empty. At the termination, the second stack (called C) has the final combed braid. This core algorithm was developed over a series of papers, primarily Artin[1950], L. O. James[1971], Thomas[1971], and Thomas[1972]. It was previously implemented by Luginbuhl[1973] in LISP. The present PL/I implementation incorporates a variant of the Luginbuhl algorithm, solving WP(B(n+1)) instead of combing braids.

V.C.2 The Combing Algorithm

Procedure to Solve WP(B(n+1)) by Combing

```
comber: proc;
```

Combing Algorithm for B(n+1)

```
Step1: Comment: Initialize vector of strands.

   /* must initialize vector as <1,2,3,4...> */
   dcl vector(40) fixed bin;
   init40: do iv=1 to 40;
           vector(iv) = iv;
   end init40 ;

Step2: Read input braid word (BWORD) in
                        standard notation;
   Set stack W to nil.
   Set stack C to nil.
   Stacks W and C are stacks of pairs (a,b).
   /*W is a Work-stack and C is a stack containing */
   /* the part of the braid that is combed so far. */

Step3:/*Translate braid word to L.O. James notation.*/
   transloop: do itrans = 1 to length(bword.gen);
       /* Translate generator to numeric.*/
       get string (substr(bword.gen,itrans,1))
                              edit (ivec) (f(1));
       /* Get the strand number which ivec acts on.*/
       istrand = vector(ivec);
       /* Compute second element of the James pair.*/
```

```
      if (substr(bword.expon,itrans,1)=' ') then
            posgen: istrand2 = vector(ivec+1);
      else
            neggen: istrand2 = -vector(ivec+1);
      /* Now push on the stack. */
      call PUSH( (istrand,istrand2)  on to W);
      /* apply action of generator on the stack.*/
      itemp = vector(ivec);
      vector(ivec) = vector(ivec+1) ;
      vector(ivec+1) = itemp;
   end transloop;
   /* Now, stack W contains the initial */
   /* braid in L.O.James notation.      */

Step4: /* Test if braid is permuted. Delete if so. */

   /*Note that the L.O.James algorithm will work */
   /*   for unpermuted braids; however, that     */
   /*   subgroup contains the identity, so we    */
   /* need first test if this braid is unpermuted.*/
   unperm: do iperm = 1 to 40;
        if (vector(iperm) = iperm) then;
        else
```

```
            permed: do ;
              put skip list ('permuted braid has',
                       ' quotient perm:'   ,vector);
              put skip list ('not the identity');
              stop;
            end permed;
   end unperm;

Step5: Comment: Call the main combing algorithm.
   Call COMB3;

Step6: Comment: Combing routine (begin main loop.)
   COMB3: do ;
      /*initial work stack has input braid*/
      /*initial combed stack is          */
      /*    empty (ie. has identity braid)*/

      /*loop until work stack is nearly   */
      /*    empty (ie. all translated into*/
      /*    combed form in combedstack)   */
      looptillnearempty: do while (topW > 2);

Step7: /*set up (ie.pop out) context */
```

```
        /*  to apply a production now */
        call pop( (c,d)  off stack W);
        call pop( (a,b)  off stack W);
        resetlpair = false;

Step8: /*Main Do-Case will split into types*/
        /*     of productions.                */
        MAINDO: DO-CASE;

Step9: /*see if production is a cancellation*/
        /*  (ie.  (c,d)(d,-c) type pair of pairs.)*/
        Cancelcase:
        Caseif ( (c=|b|) & (a=|d| & sign(b)=-sign(d))) ;
            /*cancellation pair annihilated*/
            /*reset context by one pair to */
            /*        guarantee correctness*/
            resetlpair= true;
        End Cancelcase;

Step10:/*This is the second case.*/
        /*Case of shift pair on to stack C because */
        /*   combing partial order is satisfied*/
        Shiftpaircase:
        Caseif (min(c,|d|) >= min(a,|b|)) ;
```

```
            /*Shift one pair over to combed, since */
            /* interpair order is okay. Keep other */
            /* on the work stack for next loop.    */
            call PUSH( (c,d) on to stack C );
            call PUSH( (a,b) on to stack W );
        End Shiftpaircase;
        /*The remaining six cases need order change.*/

Step11:/*Boundary Case (need only switch order) */
        Milddisordercase:
        Caseif (topC=0) ;
            /*empty stack case*/
            /*simple switch and shift*/
            call PUSH( (a,b) on to stack C );
            call PUSH( (c,d) on to stack W );
        End Milddisordercase;


Step12:/*All remaining five cases need more context */
        /*  and require more complex productions.    */
        COMPLEXPRODUCTION:
        DO-CASE ;
          /*need more context for these productions*/
          call POP( (e,f)  off stack C);
```

```
        /* to preserve correctness, must reset to */
        /* allow next loop to compare these to    */
        /* previous pairs                         */
        reset1pair=true;

Step13:/*First of the complex productions*/
       Prod1case:
       Caseif (a=c & c=|f| & e=|d|) ;
         call PUSH((|b|, -sign(b)*|d|) on to stack W);
         call PUSH((a   , d          ) on to stack W);
         call PUSH((|d|, sign(d)*a   ) on to stack W);
         call PUSH((|d|, b           ) on to stack W);
         call PUSH((a   , b          ) on to stack W);
       End Prod1case;

Step14:/*Second of the complex productions*/
       Prod2case:
       Caseif (a=c & |d|=|f| & e=|b|);
          call PUSH((|b|,-sign(b)*|d|) on to stack W);
          call PUSH((a  , sign(d)*|d|) on to stack W);
          call PUSH((a  , b          ) on to stack W);
          call PUSH((|d|, b          ) on to stack W);
          call PUSH((|b|, sign(f)*|d|) on to stack W);
       End Prod2case;
```

```
Step15:/*Third of the complex productions*/
       Prod3case:
       Caseif (a=|f| & |b|=|d| & e=c) ;
          call PUSH( (c, -sign(b)*a ) on to stack W );
          call PUSH( (c, sign(d)*|b|) on to stack W );
          call PUSH( (a, b          ) on to stack W );
          call PUSH( (a, sign(b)*c  ) on to stack W );
          call PUSH( (c, sign(f)*a  ) on to stack W );
       End Prod3case;


Step16:/*Fourth of the complex productions*/
      Prod4case:
      Caseif (|b|=|d|& e=|d|& c=|f|& sign(d)=sign(f));
         call PUSH( (c  ,-sign(b)*a  ) on to stack W );
         call PUSH( (c  , sign(d)*|b|) on to stack W );
         call PUSH( (|b|, sign(d)*c  ) on to stack W );
         call PUSH( (a  , sign(b)*c  ) on to stack W );
         call PUSH( (a  , b          ) on to stack W );
      End Prod4case;


Step17:/*Fifth of the complex productions*/
       Prod5case:
```

```
        /* Otherwise Case*/
        Caseif (TRUE) ;
           /*need to reorder only */
           call PUSH( ( c, d) on to stack W );
           call PUSH( ( a, b) on to stack W );
           call PUSH( ( e, f) on to stack W );
        End Prod5case;

Step17.5: /*Close all DO-CASEs*/
        End COMPLEXPRODUCTION;
      End MAINDO;


Step18:/* Handle cases here where a reset is needed  */
        /* before next loop to maintain consistency   */
        /* (ie. leave partial order consistency work  */
        /* for the boundary between stacks for later.)*/
        if (reset1pair & (topC>0)) then
             do; /*must move context back to stack W*/
               call POP((c,d) from stack C );
               call PUSH( (c,d) on to stack W );
               /*Next loop will "comb-up" the  */
               /*          context just added. */
             end;
```

```
Step19: /* Close Main Loop and Print Output*/
      End looptillnearempty;
      outputstage: do;
      if (topC=0)
      then put skip list('braid is the identity');
      else put skip list ('braid is not trivial.');
     end outputstage;
    end COMB3;
   end  comber;
```

End of Combing Algorithm for WP(B(n+1))

V.D Combinatorial Analysis of Combing Algorithm

Analysis still in progress. Problem: proof falls into twenty-three cases, each requiring a closure proof (for no more cases.) Even then, each case will have a difference equation inter-related with the others; even if they prove decomposable, the distributions of initial words, over which they operate has to be determined. Even asymptotically, this is not uniform and requires more time.

The Monte Carlo analyses will demonstrate the irregular behavior, especially the enumerations over long words.

V.E Monte-Carlo Analysis of Combing Algorithm

Measures Used:

AVCOMBEND = average size of topC at end-of-run (when exit occurs from the main loop.)

AVCOMBALL = average size of all topC (during all stages in progress.) Actually, this is the average for all times through the main loop.

AVWORKALL = average size of all topW.

Note: AVWORKEND=0 since stack W is empty (as a loop exit condition.)

MXCOMBEND = Maximum topC at end-of-run.

MXSUMALL = Maximum of all stack sums (ie. topC+topW) for all times through the loop (as well as end-of-runs.)

Note: MXSUMEND=MXCOMBEND since topW=0 at end-of-runs.

Note: All these measures are in number of integers used. So the above values are two times the number of pairs stored in the respective stacks used.

Combing Statistics for: n=2

n=number of generators , l=length of braid word

	l=2	l=4	l=6	l=8
AVCOMBEND	2.00	4.462	7.381	10.065
AVCOMBALL	1.00	3.143	4.845	6.152
AVWORKALL	1.00	6.750	11.609	15.370
MXSUMEND	4	20	40	60
MXSUMALL	4	28	64	100

Combing Algorithm for B(n+1)

For n=2, MXSUMEND=MXSUMALL for lengths m = 2, 4, 6, and 8. Their corresponding worst case braid-words were:

m=2	11
m=4	2211
m=6	222211
m=8	22222211

Combing Statistics for: n=3

n=number of generators , l=length of braid word

	l=2	l=4	l=6	l=8
AVCOMBEND	2.00	4.423	7.511	
AVCOMBALL	1.00	3.709	4.709	
AVWORKALL	1.00	6.234	10.649	
MXSUMEND	4	20	60	partial answer 148
MXSUMALL	4	28	76	partial answer 184

For n=3, MXSUMEND=MXSUMALL for lengths m = 2, 4, 6, and 8. Their corresponding worst case braid-words were:

m=2	11
m=4	2211
m=6	332211
m=8	3'23'23'211 =partial answer

Combing Statistics for: n=4

n=number of generators , l=length of braid word

	l=2	l=4	l=6
AVCOMBEND	2.00	4.343	7.296
AVCOMBALL	1.00	2.464	4.188
AVWORKALL	1.00	5.913	9.7827
MXSUMEND	4	20	60
MXSUMALL	4	28	76

For n=4, MXSUMEND=MXSUMALL for lengths m = 2, 4, and 6. Their corresponding worst case braid-words were:

m=2	11
m=4	2211
m=6	332211

Combing Statistics for: n=7

n=number of generators , l=length of braid word

	l=2	l=4
AVCOMBEND	2.00	4.207
AVCOMBALL	1.00	2.139
AVWORKALL	1.00	5.476
MXSUMEND	4	20
MXSUMALL	4	28

For n=7, MXSUMEND=MXSUMALL for lengths m = 2 and 4. Their corresponding worst case braid-words were:

m=2	11
m=4	2211

Combing Algorithm for B(n+1)

One further enumeration was carried out to note the worst case strings of length eight and three generators.

m=8 , n=3

String	MXSUMEND	MXSUMALL
2222222211	80	136
3322222211	148	188
2'332222211	168	196
23'3'2222211	184	200
2'2'33222211	204	264
3'223'222211	228	296
223'3'222211	236	308
332'3322211	240	340
23'223'22211	296	---
3'23'23'22211	316	384
223'223'2211	---	420
23'23'23'2211	372	---

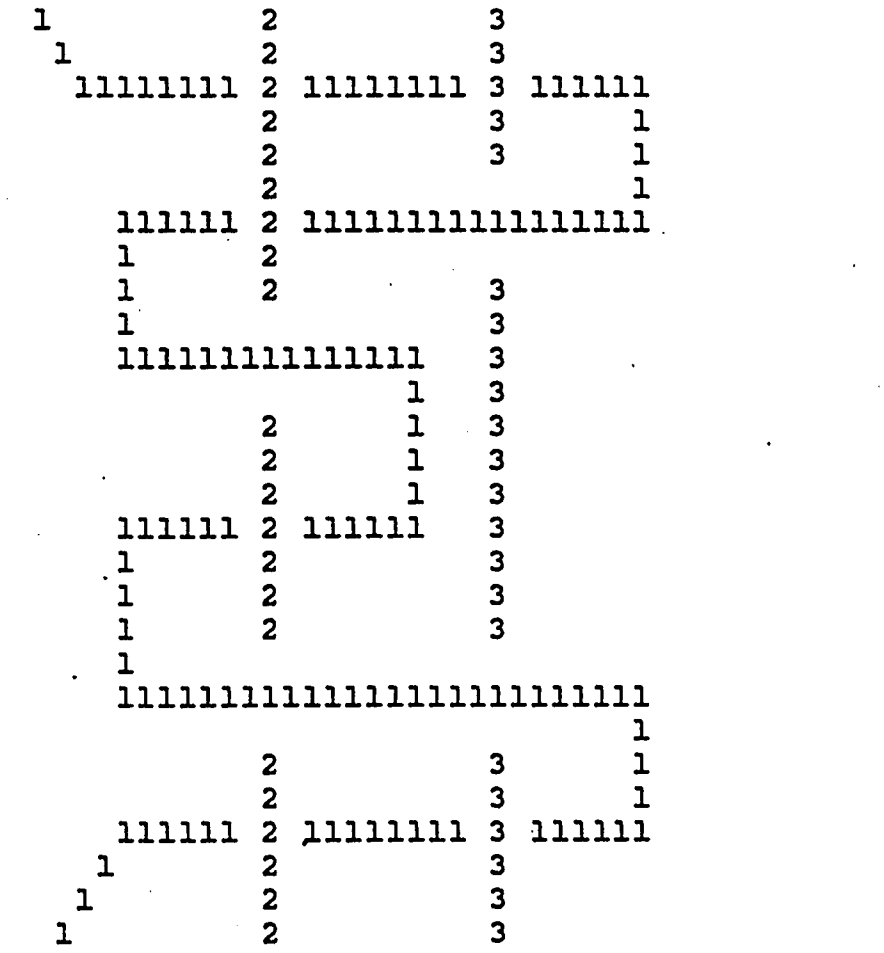
VI.0 Lisa's Algorithm

VI.A Lisa Braids

Def: A Lisa braid is an unpermuted braid, all of whose 2-subbraids are reducible to the identity.

Example: 131'3' is a Lisa braid.

Example: 1'2'2'1111221 is not a Lisa braid.



The above is not a Lisa braid because the 1-2-subbraid (ie. the subbraid composed of strands 1 and 2) is nontrivial, specifically being 1'1' or equivalently, [1 2][2 1]. The 1-3-subbraid and the 2-3-subbraid are trivial.

Note: Testing a 2-braid to determine if it is trivial requires logspace and linear time on a Turing Machine. Essentially, one sweep is made of the input tape, counting occurrences of 1' as -1 and 1 as +1. If the final sum is zero, the braid is the identity.

Note: To determine which braids are Lisa braids, the standard braid notation cannot be used directly because the specific strand numbers are not specified along each step (or generator) in the braid word. So this notation loses track of the strands and the subbraids are soon no longer apparent.

The L. O. James notation follows the strands exactly. In fact, the James notation specifies the very pairs that are used. Any two strand subbraid (for instance a and b) can be constructed by picking off all pairs [a b], [b a], [a -b], and [b -a] sequentially, in the exact order that they occur.

Finally, to determine if a two strand subbraid is the identity, we need only count [a b] or [b a] as +1 and count [a -b] or [b -a] as -1. If the final sum is zero, then the subbraid is trivial.

If we simultaneously have (n)(n-1)/2 counters, then if all of them are finally zero, the braid is a Lisa braid.

VI.B Lisa's Algorithm

Sketchy Version

Step1: Read braid word W;

Step2: Convert to L. O. James Notation;

Step3: Using (n)(n-1)/2 counters, sequentially count off each pair and accordingly add +1 or -1 into the proper counter.

Step4: If all (counters=0)
then print 'This is a Lisa braid.'

Note: Since n is fixed beforehand, the n(n-1)/2 is not of real significance.

Note: The James notation requires linear space. To alleviate this gross waste of space, we can develope one James notation pair at a time, use it in the computation, and then save space by not storing it. This gives the final version of the program below:

Lisa's Algorithm

```
Step1: Read braid word W=a1a2...am;

Step2: Set VECTOR=<1,2,...,n+1>;
       /* This is the strand recording vector.*/

Step3: Set COUNTERS( 1:n , 1:n ) = 0;

Step4: LOOP: DO  I = 1 TO m ;
         /* take i'th generator of W */
         GEN = ai ;
         STRAND1 = VECTOR(GEN);
         STRAND2 = VECTOR(GEN + 1);
         IF ( GEN is a positive generator) THEN
            COUNTERS(STRAND1,STRAND2) =
                COUNTERS(STRAND1,STRAND2) + 1 ;
         ELSE
```

```
            COUNTERS(STRAND1,STRAND2) =
                COUNTERS(STRAND1,STRAND2) - 1 ;
         ENDIF;
         /* now apply action of generator on */
         /* strand order vector                */
         ITEMP = VECTOR(GEN) ;
         VECTOR(GEN) = VECTOR(GEN + 1) ;
         VECTOR(GEN + 1) = ITEMP ;
      END LOOP;
      /*NOTE: In the above code, [i j] and [j i]*/
      /* cases are treated as distinct in the   */
      /* counter sums. This saves time. In the  */
      /* final step, counters (j,k) and (k,j)   */
      /* are added together, giving the correct */
      /*values.                                 */

Step5: /* final count test */
      KLOOP: DO K = 1 TO N;
         JLOOP: DO J = K TO N;

            IF (COUNTERS(J,K) + COUNTERS(K,J)
                = 0 ) THEN ; /* OK CASE */
            ELSE
```

```
            DO;
             PRINT, W 'IS NOT A LISA BRAID';
             STOP;
            END;
          ENDIF;
       END JLOOP;
    END KLOOP;
    PRINT, W 'IS A LISA BRAID';
    STOP;
```

Note: We need not test if a braid is permuted. All Lisa braids are unpermuted (because even a single transposition would result in two strands not being an identity 2-braid.)

VI.C Analysis of Lisa's Algorithm

VI.C.1 Worst Case Analysis

Space: VECTOR requires (n+1)*log(n+1) bits.
COUNTERS requires ((n)(n-1)/2)(log m) bits.

Since n is assumed fixed, this gives a logspace worst case behavior. The exact worst case is bound is far below this figure but this will suffice.

Clearly, this algorithm requires logspace.

Furthermore, if n is very large, we can represent the counters as a linked list which can be bounded in storage requirements by min[(n)(n-1)/2*logm , 5mlogm.] The 5mlogm is an overestimate of the upper bound for a binary tree having five fields: left and right links, both strand numbers, and the actual counter itself. This model operates on the USE principle; each time a new i-j-pair is encountered, a new leaf is inserted with a counter for that pair. In fact, by using a clever hashing function, we can reduce this further (as well as reduce time.)

Time: n steps to set VECTOR to 0.

O(n**2) steps to clear the counters.

O(m) steps in the main loop.

O(n**2) steps to test the counters.

So the algorithm runs in linear time.

Generally, n is not of concern; however, if n is very large relative to m, then using binary trees (with a partial ordering of nodes) runs into O(mlog(n**2)) = O(mlog(n)) time.

Note: An average case analysis is unnecessary because even the worst case analysis is better than can be believed.

Note: This general algorithm is much faster than the Lipton-Zalcstein-Burau algorithm for B(4).

Unfortunately, this algorithm is not a sufficient condition test for identity braid .

VI.D Proof: Necessary but Almost Sufficient

Thm: Every identity braid is a Lisa-braid.

Proof1: The null braid is a Lisa-braid (trivial, since all counters are zero.)

Lisa's Algorithm

All free relators are Lisa braid.

All commutative braid relators (eg. 131'3') are Lisa braids.

All length-six braid relators (eg. 1212'1'2') are Lisa braids.

If a braid is a Lisa braid, then the insertion or deletion of a braid relator will result in a Lisa braid. The argument here is based on counters (ie. 0+0=0 and 0-0=0.) Since this construction generates all identity braids, all identity braids are Lisa braids.

QED1

As an alternate proof (using contradiction):

Proof2: Assume a braid is not a Lisa braid. Then two strands exist which cannot be unraveled. By adding on the remaining strands, those two strands will still not be ravelable. If a braid cannot be unraveled, then it is not an identity braid.

QED2

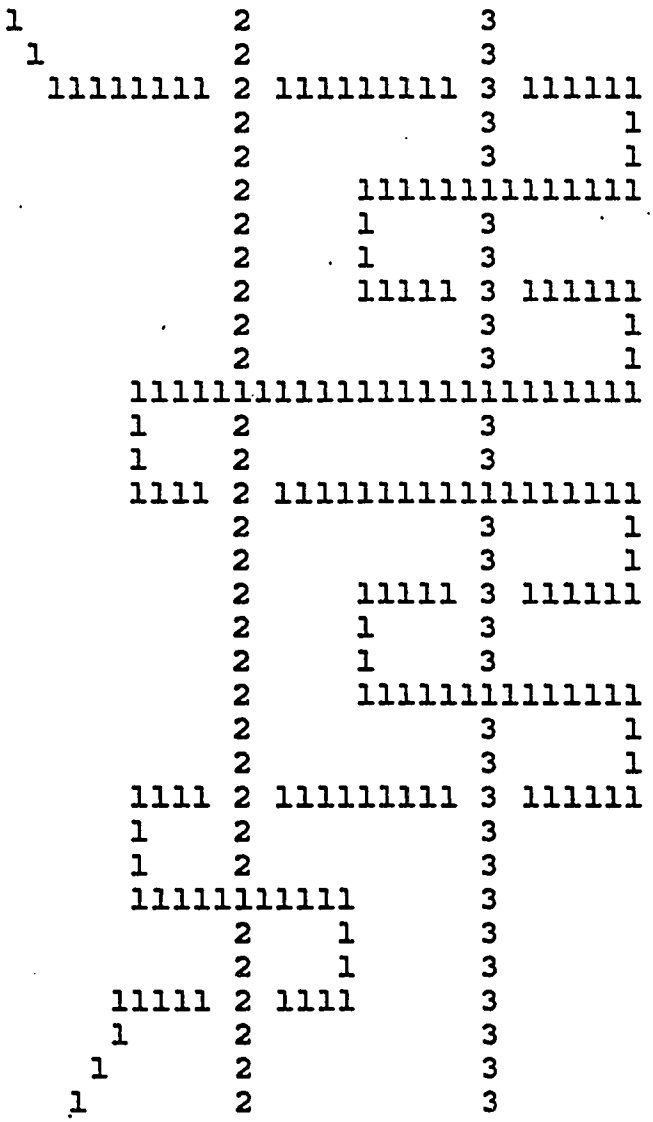
Unfortunately, the converse is not true.

Thm: There exist Lisa braids which are not reducible to the identity braid.

Proof: (By counterexample:)

The braid 1'2'2'2'1'1'222111 is a Lisa braid but not an identity braid. This is also the smallest known case of one.

Picture of 1'2'2'2'1'1'222111



One can prove this is a Lisa braid by either pulling out one strand (3 cases) or by using the algorithm. The braid is clearly non-trivial (can be

proven by heuristics, algorithms, invariant, etc.)

As a alternate example of a non-identity Lisa braid consider: 111112222221'1'2'2'2'2'2'2'1'1'1'. This was the first one discovered. (QED)

VI.E Final Notes

Lisa's algorithm could possibly be fixed (to satisfy sufficiency) by analyzing the categories of counter-example cases. Unfortunately, the ones described are among the shortest and enumeration would be impossible with the present generation of computers. Hours of human labor were required to get a number of categories. Winding numbers are not sufficient; there are other categories with bundles of strands which do not fit the nice model of the counterexamples shown. Unfortunately, no proof of completeness has been worked out yet. Despite the large number of classes, these counterexamples are proportionally extremely rare. If categorized in easily detected classes, this may be the first logspace algorithm for WP(B(n)) for all n.

Lisa's Algorithm

As a last attack on the logspace issue, a new algorithm is still being designed which is strictly logspace (despite much wasted time (presently quadratic time.)) Unfortunately, this algorithm has many cases which have not been worked out yet. It is based on a number of theorems from the theory of functions.

VII.0 Other Notations and Presentations

VII.A < a , b ; a**3 = b**2 > Algorithm

Using the < a , b ; a**3 = b**2 > presentation, we have the following algorithm for WP(B(3)):

The a**2=b**3 Algorithm

```
PROCEDURE A3B2BRAID;
Assume input word is
     in form W=(a**k1)(b**k2)(a**k3)...(b**kn);
Set CounterA =0;  Set CounterB=0;
Start Turing Machine head (TMH) on
                leftmost generator;
m345: loop until (TMH moves off word on right);
    Do Case
       Case1:IF ( exponent k>0mod3 and TMH=a) then
                do;
                CounterA = k - kmod3 +CounterA;
                Leave a**(kmod3) in place of a**k;
                end;
       Case2:IF ( exponent k=0mod3 and TMH=a) then
                do;
                CounterA = k +CounterA;
                Remove a**k off tape;
```

```
                Do all cancellations between
                   the two boundaries of deleted
                   term;
                Set TMH in position;
                end;
        Case3:IF ( exponent k>0mod2 and TMH=b) then
                do;
                CounterB = k - kmod2 +CounterB;;
                Leave b**(kmod2) in place of b**k;
                end;
        Case4:IF ( exponent k=0mod2 and TMH=b) then
                do;
                CounterB = k +CounterB;
                Remove b**k off tape;
                Do all cancellations between
                   the two boundaries of deleted
                   term;
                Set TMH in position;
                end;

    End Case;
    Move TMH right;
end loop m345;
Comment: Final test.
```

```
    IF ((CounterA=0 and CounterB=0 and Tape is empty) or
        ( (CounterA/3) + (CounterB/2) = 0) )
      THEN PRINT 'IDENTITY';
      ELSE PRINT 'FALSE';
  END  A3B2BRAID;
```

To see the process involved, note that counters A and B record only the a**3 and b**2 chunks because they can commute over the opposite generators, respectively.

Note that the modulo-residues will remain until a nearby cancellation removes terms and starts a possible chain reaction of cancellations.

Theorem: This algorithm operates in linear space and time.

Proof: Probability(ab or ba at a given position in the input string) = 1/2. So long strings of a's or b's are "very" rare (exponentially decreasing relative to length.) So the a**k's and b**k's occur linearly often (relative to the average initial input length.)

Probability of an a-string leaving a residue is 2/3. For b, it is 1/2. So, the TMH (as it moves right) leaves a linearly long trail of residues. The probability of cancellation at one point is 1/2 and 1/3 respectively (so that still keeps the trail linear.) Finally, the probability of chain-cancellations decreases exponentially with respect to the number in the chain (in fact, the alternating a's and b's gives the chain exponential a form like (1/2)**k*(2/3)**(k+constant) where k is the number of ab alternations in the chain.) So, the trail is linear space. The algorithm is linear time because cancellation occurs only once for any symbol and cancellation is refused linearly many time.

(QED)

VIII.0 Birman-Hilden Algorithm

A new version of the Artin algorithm was under developement, using a reduction based upon the Birman-Hilden theorem (ie. adding a contracting rule a(i)**2 to the TUPLE's.) As noted in the chapter on the Artin algorithm, this almost never occurs (ie. TUPLE rarely has as square term.) This could probably be shown not to ever occur; this would offer an insight into an alternate proof for the Birman-Hilden isomorphism result. Unfortunately, this cancels any benefit from using this reduction in B(n+1) for n greater than two. Therefore, the program was not implemented.

Note: For B(3) significant savings do occur.

References

Anisimov, A. V. [1973], "Group Languages", Cybernetics, vol. 4, p594-601

Artin, Emil [1925], "Theorie der Zopfe", Abhandl. Math. Sem. Univ. Hamburg, vol. 4, p47-72, (presentation of B(n) and solution of its' WP)

Artin, E. [1947a], "Theory of Braids", Annals of Math., vol. 48, p101-126

Artin, E. [1947c], "Braids and Permutations", Annals of Math., vol. 48, p643-649

Artin, E. [1950], "Theory of Braids", American Scientist, vol. 38, p112-119

Birman, Joan S. [1968], Braid Groups and Their Relationship to Mapping Class Groups, Ph.D. Thesis, New York Univ., 93pp., (diss. abstracts 29/03-B p.1085, order no. 86-13107)

Birman, J. S. [1968], "Automorphisms of the fundamental group of a closed orientable 2-manifold", Proc. Amer. Math. Soc., vol. 21, p351-354

Birman, J. S. [1969a], "On Braid Groups", Comm. Pure and Applied Math., vol. 22, p41-72

Birman, J. S. [1969b], "Mapping Class Groups and Their Relationship to Braid Groups", Comm. Pure and Applied Math., vol. 22, p213-238

Birman, J. S. [1969c], "Non-Conjugate Braids can Define Isotopic Knots", Comm. Pure and Applied Math., vol. 22, p239-242

References

Birman, J. [1975], Braids, Links, and Mapping Class Groups, (Math. Studies Series 82), Princeton University Press, Princeton, New Jersey,

Birman, J. [1981], Braid Groups, (manuscript (dated November 8) and and conference held at the C.U.N.Y. Graduate Center Computer Facility)

Birman, J. S. and Hilden, H. M. [1971], "On the mapping class group of closed surfaces as covering spaces", in Annals of Math. Studies Series 66, Princeton Univ. Press, p81-115

*Burau, W. [1932], "Uber Zopfinvarianten", Abh. Math. Sem. Hamburg Univ., vol. 9, p117-124

*Burau, W. [1934], "Kennzeichnung der Schlauchverkettungen", Abh. Math. Sem. Hamburg Univ., vol. 10, p285-297

*Burau, W. [1935], "Uber Verkettungsgruppen", Abh. Math. Sem. Hamburg Univ., vol. 11, p171-178, (important)

*Burau, W. [1936], "Uber zopfgruppen und gleicksinnig verdrillte Verkettungen", Abh. Math. Semin. Hamburg (or Hanisischen)Univ., vol. 48, p179-185, (or vol. 11, p171-178) (important paper on braid-representation)

*Dehn, M. [1910], "Uber die Topologie des dreidimensionalen Raumes", Math. Ann., vol. 69, p137-168

*Dehn, M. [1911], "Uber unendliche diskontinuierliche Gruppen" Math. Ann., vol. 71, p116-144, (the origin of WP, CP, and ISOP)

References

*Dehn, M. [1912], "Transformation der Kurven auf zweiseitigen Flaechen", Mathematische Annalen, vol. 72, p413-421

*Dehn, M. [1914], "Die beiden Kleeblattschlingen", Math. Ann., vol. 75, p1-12, (also check: p402-413)

*Dehn, M. [1928], Uber die geistige Eigenart des Mathematikers, Frankfurter Universitatsreden no. 27, Universitatsdruckerei Werner und Winter, Frankfurt am Main, 25pp.

*Dehn, M. [1931], "Uber einige neuere Forschungen in den Grundlagen der Geometrie", Matematisk Tideskrift B, No. 3-4

*Dehn, M. [1938], "Die gruppe der abbildungsklassen", Acta Math., vol. 69, p135-206

*Dehn, M. [1939], "Die Gruppe der Abbildungsklassen", Acta Math., vol. 69, p. 135-206

*Dehn, Max and Heegaard, P. [1907], "Analysis Situs", in Enzyklopadie der Mathematischen Wissenschaften, vol. 3 (III AB3), p153-220, Teubner, Leipzig-Berlin

Dixon, John Douglas [1973], "Free Subgroups of Linear Groups", p45-56 in Conf. on Group Theory (Univ. of Wisconsin Parkside, Kenoska, Wisconsin)), (Lecture Notes in Math. 319), Springer Verlag, Berlin - N.Y.

Domanski, B. [1979], The Complexity of Decision Problems in Group Theory, Ph.D. Thesis, Graduate Center (C.U.N.Y), New York

References

Domanski, B. [1982], "The Complexity of Two Decision Problems for Free Groups", Houston Journal of Math., vol. 8, no. 1, p29-38

*Dyck, Walther von [1882], "Gruppentheoretische Studien (3 illustrations at end of vol.)", Math. Ann., vol. 20, p1-44, (unified all of group theory under the auspices of abstract group theory; also earliest records of free groups, relations, and presentations)

*Dyck, Walther von [1883], "Gruppentheoretische Studien II", Math. Ann., vol. 22, p70-118

Dyer, Joan L. [1979a], "The Algebraic Braid Groups are Torsion-Free: An Algebraic Proof", IBM T. J. Watson Research Center (Research Report 7736 (33423) 6/15/79 Math.), 7pp.

Dyer, Joan L. [1979b], "Separating Conjugates in Amalgamated Free and HNN Extensions", IBM T. J. Watson Research Center (Research Report 7762 (33626) 7/17/79 Math.), 21pp.

Dyer, Joan L. and Grossman, Edna K. [1981], "The Automorphisms of Braid Groups", Amer. Journal of Math., vol. 103, no. 6, p1161-1169

Dyer, Joan L., Grossman, Edna K., and Formanek, E. [1981], "Automorphism Groups of Free Groups", IBM T. J. Watson Research Center (Research Report 8701 (38025) 2/17/81 Math.), 10pp.

Feller, W. [1968], An Introduction to Probability Theory and its Applications, (2 vol.), Wiley, New York

References

Garside, F. A. [1965], The Theory of Knots and Associated Problems, D. Phil. Thesis (Nov. 1965), Oxford University (Corpus Christi College), (ASLIB Index to Theses 1965/66 p.67 1333), 97pp., (Solves Conjugacy Problem for braid groups)

Garside, F. A. [1969], "The braid group and other groups", Quart. J. Math. Oxford, vol. 20, p235-254, (compact version of thesis)

Gassner, Betty Jane [1957], On Braid Groups, Ph.D. Thesis, New York Univ., 42pp., (diss. abstracts 33/04-B p.1664, order no. 72-24487)

Gassner, Betty Jane [1961], "On Braid Groups", Abh. Math. Sem. Hamburg Univ., vol. 25, p10-22, (condensation of her thesis)

Gorin, E. A. and Lin, V. Ja [1969], "Algebraic Equations with Continuous Coefficients and Some Problems in the Algebraic Theory of Braids", (English Translation), Math. Sobornik, vol. 7, no. 4, p569-596

Harrison, M. A. [1978], Introduction to Formal Language Theory, Addison-Wesley, Reading, Mass., 608pp.

Harrison, N. [1972], "Real length functions in groups", Trans. Amer. Math. Soc., vol. 174, p77-106

Hopcroft, J. and Ullman, J. [1969], Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Mass., 242pp.

References

James, L. O., Luginbuhl, E., and Thomas, R. S. D. [1971], "On notations for non-permuting braids", in Proc. 25th Summer Meeting Canadian Math. Congress at Thunder Bay, Thunder Bay, Ontario, pages 457-470

Kesten, H. [1959], "Symmetric random walks on groups", Trans. Amer. Math. Soc., vol. 92, p336-354

Kesten, H. and Spitzer, F. [1965], "Random walks on countably infinite Abelian groups", Acta Math., vol. 114, p237-259

Lipschutz, Seymour S. [1960], On the Braid Group, Ph.D. Thesis, New York Univ., 28pp., (diss. abstracts 21/01 p.202, order no. 60-02296)

Lipschutz, Seymour [1961], "On a Finite Representation of the Braid Group", Arch. Math., vol. 12, p7-12

Lipschutz, Seymour [1963], "Note on a paper by Shepperd on the braid group", Proc. Amer. Math. Soc., vol. 14, p225-227

Lipschutz, Seymour [1964], "An Extension of Greendlinger's Results on the Word Problem", Proc. Amer. Math. Soc., vol. 15, p37-43

*Lipton, R. [1975], "Probabilistic Algorithms for Group-Theoretic Problems", SIGSAM Bulletin, (A.C.M. Publication), Issue 46, p8

Lipton, Richard J. and Zalcstein, Yechezkel [1977], "Word Problems Solvable in Log-Space", Journal A.C.M., vol. 24, no. 3, p522-526, (WP(free groups) solvable in logspace and ref. to Cannonito' s Grzegorzck work)

References

Lipton, Richard J. and Zalcstein, Yechezkel [1978], "Probabilistic algorithms for group-theoretic problems", A.C.M. SIGSAM Bull., vol. 12, no. 2, p8-9

Liu, C. L. [1968], Introduction to Combinatorial Mathematics, McGraw-Hill, New York

Luginbuhl, E. [1973], The Computer Implementation of Braid Algorithms, M.Sc. Thesis, Univ. of Manitoba, 129pp.

Magnus, W. [1954], "On the exponential solution of differential equations for a linear operator", Comm. Pure and Appl. Math., vol. 7, p649-673

Magnus, W. [1972], "Braids and Riemann surfaces", Comm. Pure Appl. Math., vol. 25, p151-161

Magnus, W. [1974a], "Braid groups: A survey", Proc. 2nd Internat. Conf. on the Theory of Groups, (Lect. Notes in Math. 372), Springer-Verlag, New York, pages 463-487

Magnus, W. [1978], "Max Dehn", Math. Intelligencer, vol. 1, p132-142, (Dehn's 100th birthday rememberence)

Magnus, W., Karrass, A., and Solitar, D. [1966;reprinted 1976], Combinatorial Group Theory: Presentations of Groups in Terms of Generators and Relations, Dover, New York, 444pp.

*Magnus, W. and Moufang, R. [1954], "Max Dehn zum Gedachtnis", Math. Ann., vol. 127, 215-227, (Obituary of Max Dehn, includes a list of complete works)

References

Magnus, W. and Peluso, A. [1967], "On Knot Groups", Comm. Pure and Appl. Math., vol. 20, p749-770

Magnus, Wilhelm and Tretkoff, Carol [1980], "Representations of automorphism groups of free groups", in Adjan, Boone, and Higman (ed.) [1980], Word Problems II, pages 255-259

Makanin, G. S. [1968], "The conjugacy problem in the braid group", Doklady Akad. Nauk. SSSR, vol. 182, p495-496

*Markov, A. A. [1936], "Uber fie freie Aquivalenz der geschlossen Zopfe", Rec. Math. Moscow, vol. 1, p73-78, (Markov operations on braids)

*Markov, A. [1945], Foundations of the Algebraic Theory of Braids, (Trudy Math. Inst. Steklov vol. 16), 55pp., (Russian paper with English summary), (Braids were called tresses then.), (MR 8 p131)

Moran, S. [1980], "Matrix Representations for the Braid Group B(4)", Arch. Math., vol. 34, p496-501

Moran, Siegfried [1983], The Mathematical Theory of Knots and Braids, (North-Holland Math. Studies vol. 82), Elsevier Sci. Pub. Co., New York, 296pp., (ISBN 0-444-86714-7)

Muller, David E. and Schupp, P. E. [1983], "Groups, the theory of ends, and context free languages", Jour. Comput. System Sci., vol. 26, no. 3, p295-310, (MR 84k:20016)

References

Renyi, A. [1970], Foundations of Probability, Holden-Day, San Francisco, Calif.

Spitzer, F. [1976], Principles of Random Walks, (Grad. Texts in Math. 34), Springer-Verlag, New York

Squier, Craig C. [1984], "The Burau Representation is Unitary", Proc. Amer. Math. Soc., vol. 90, no. 2, p199-202

Stillwell, J. [1980], Classical Topology and Combinatorial Group Theory, (Graduate Texts in Mathematics: Volume 72), Springer-Verlag, New York - Heidelberg - Berlin, 301 pp.

Thomas, R. S. D. [1971], "Computed Topological Equivalence of Partially Closed Braids", in Proc. of the Twenty-fifth Summer Meeting of the Canadian Math. Congress (Lakehead Univ., Thunder Bay, Ontario, June 1971), Lakehead Univ., Thunder Bay, Ontario, p564-584, (MR 49 3887)

Thomas, R. S. D. [1971], "An Algorithm for Combing Braids", in Proc. Second Louisiana Conference on Combinatorics and Graph Theory, Louisiana State Univ., Baton Rouge, La., p517-532, (MR 50 5780)

Thomas, R. S. D. [1972], Note on Isotopy of Closed Braids, Science Report no. 56, Univ. of Manitoba

Thomas, R. S. D. [1972], "Closed and Partially Closed Braids", in Proc. Third Southeastern Conference on Combinatorics, Graph Theory, and Computing (Florida Atlantic Univ., Roca Baton, Fla.), Florida Atlantic Univ., Roca Baton, Fla., p447-450, (MR 49 11496)

References

Thomas, R. S. D. [1974], "Partially Closed Braids", Canadian Bull. Math., vol. 17, p99-107

Thomas, R. S. D. [1974], "The Structure of Fundamental Braids", Quarterly Journal of Math. Oxford, Series 2, vol. 26, no. 103 p283-288

Thomas, R. S. D., James, L. O., and Luginbuhl, E. [1971], "On Notations for Non-Permuting Braids", in Proc. of the Twenty-fifth Summer Meeting of the Canadian Math. Congress (Lakehead Univ., Thunder Bay, Ontario, 1971), p457-470, Lakehead Univ., Thunder Bay, Ontario, (MR 50 9044)

Thomas, R. S. D. and Luginbuhl, E. [1973], "A Table of Knots as Closed Braids", in Proc. Southeastern Conference on Combinatorics, Graph Theory, and Computing (Florida Atlantic Univ., Roca Baton, Fla.), Florida Atlantic Univ., Roca Baton, Fla., p423-444, (MR 50 5776), reprinted in Utilas Math., (Winnipeg, Manitoba) vol. 13,

Thomas, R. S. D. and Paley, B. T. [1974], "Garside's Braid-Conjugacy Solution Implemented", Utilas Mathematica, vol. 6, p321-335, (Published by: Utilas Math. Pub. Inc., P. O. Box 7, University Centre, Univ. of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2), (ISSN: 0315-3681), (MR 50 13208), (Algorithm is in Paley [1974], his thesis)

Tits, J. [1972], "Free subgroups in linear groups", Journal of Algebra, vol. 20, p250-270