THE UNIVERSITY OF SOUTH ALABAMA

SCHOOL OF COMPUTER AND INFORMATION SCIENCES

SYMBOLIC DYNAMICS APPLIED TO COMBINATORIAL GROUP THEORY : A TOOLKIT

BY

Dong-Biao Zheng

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in fulfillment of the requirement of the degree of

Master of Science

in

Computer Science

March 1996

Approved: Date:

___

Chair Of Thesis Committee

___

Co-Chair of Thesis Committee

___

Co-Chair of Thesis Committee

___

Committee Member

___

Committee Member

___

Department Chair

___

Director of Graduate Studies

___

Dean of the Graduate School

SYMBOLIC DYNAMICS APPLIED TO COMBINATORIAL GROUP THEORY : A TOOLKIT

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in fulfillment of the requirement of the degree of

Master of Science

in

Computer Science

By
Dong-Biao Zheng
M.S., Mathematics, University of South Alabama, 1994
B.S., Applied Mathematics, Hua Qiao University, 1990
March 1996

ACKNOWLEDGMENTS

From the bottom of my heart, I would like to express my greatest appreciation to Dr. Marino J. Niccolai who morally and financially supported me through my graduate study in the Department of Computer Science. Without his patience, expertise, firm guidance, and strong support, this project would not have been possible. His compassion and kindness are always remembered.

On the other hand, Dr. Thomas F. Hain, with his expertise in C++, gave me enormous help in the class design of this project. I am indebted to him for guiding me through the C++. He even kindly let me use his office when I had difficulties getting access to UNIX Lab. Without his generous help, this project would not have been done so smoothly.

Needless to say, Drs. Susan Williams and Daniel Silver contributed much to the success of the project. Their careful and strict attitude in doing research impressed me.

It was a pleasure to work with them. I would also like to thank Dr. R. Daigle for his interest in this project.

My biggest debt is to my wife, Lifang. Her support of the project and her understanding of the time I devoted to it are duly noted and deeply appreciated.

TABLE OF CONTENTS

LIST OF TABLES ………………………………………………………….. iv

LIST OF FIGURES …………………………………………………….…. v

ABSTRACT …………………………………………………………… vi

1. INTRODUCTION ……………………………………………….. 1

2. BACKGROUND …………………………………………………… 4

3. THEORETICAL BASIS …………………………………………… 9

4. ALGORITHMS AND IMPLEMENTATION ……….………….. 16

5. EXPERIMENTATION AND ANALYSIS…….………...………… 29

6. CONCLUSION AND RECOMMENDATIONS …………………. 41

REFERENCE LIST …………………………………………………… 43

APPENDIX A: PERFORMANCE CHART FOR p1q2r3 ….…………….. 45

APPENDIX B: PERFORMANCE CHART FOR p2q1r2 …….……….. 50

APPENDIX C: PERFORMANCE CHART FOR ALL 3 SAMPLES … 55

APPENDIX D: SOURCE CODE ……………………………………… 60

VITA …………………………………………………………….……….. 70

LIST OF TABLES

Table 1 - Example for Code Modification …………………………..….. 27

Table 2 - Run Time at Each Step ……………………………………. 34

LIST OF FIGURES

Figure 1 - Graph Γ describing $\Phi_{S3}(g_{k(5, 1, 13)})$ …..…………………… 33

Figure 2 - Step 1 – p3q0r3 ……….…………………………………… 35

Figure 3 - Step 2 – p3q0r3 ……….…………………………………… 36

Figure 4 - Step 3 – p3q0r3 ……….…………………………………… 36

Figure 5 - Step 4 – p3q0r3 …………………………………………. 37

Figure 6 - Step 5 – p3q0r3 …………………………………………… 37

Figure 7 - Step 6 – p3q0r3 …………………………………………. 38

Figure 8 - Total Run Time – p3q0r3 ……………………………….. 38

Figure 9 - Good Nodes – p3q0r3 ……...…………………………… 39

Figure 10 - Run Time at Each Step – R5p3q0r3 ……..……………….. 39

ABSTRACT

Zheng, Dong-Biao. M.S., University of South Alabama, March 1996. Symbolic Dynamics Applied to Combinatorial Group Theory: A Toolkit. Co-Chairs of Committee: Marino J. Niccolai, Thomas F. Hain, Dan Silver.

A recently discovered algorithm for finding all finite permutation representations for a given *finite* ***Z****-dynamically presented* group **K** was implemented using an object-oriented programming model. The program was run for groups **K** that arise naturally in knot theory and was used to compute the first few members of a sequence $\{ h_r \}$ of new knot invariants. It is conjectured that the sequence is a highly effective means to show that different finite **Z-**dynamically presented groups are in fact different. This conjecture was tested on many finite **Z-**dynamically presented groups that arise from knots with 9 or fewer crossings. If two knots have different invariants for some value of r, then the knots themselves must be different. Initial results indicate that a relatively small value of r suffices to distinguish all of the knots in our sample set.

A second contribution of this thesis is to provide the first known example of an alternating knot with the property that the trivial representation of K, the commutator subgroup associated with the group of the knot, can be approximated arbitrarily closely by nontrivial representations of K in the symmetric group S_5.

This research resulted in an object-oriented toolkit that can be used by mathematical researchers in group theory and knot theory to investigate problems that are considered to be of interest but computationally impractical.

1. INTRODUCTION

There are many algorithms available for computing with finite groups, especially finite permutation groups [Si1] [BC] [Bu]. However, there are relatively few methods for working with infinite groups. Recently, Drs. D. Silver and S. Williams (Department of Mathematics and Statistics, University of South Alabama) discovered an algorithm for finding all finite permutation representations for a given *finite **Z**-dynamically presented* group **K** [SW]. This thesis reports on the development of an efficient implementation of this algorithm. The approach relies on the use of object classes in C++ to implement the important mathematical constructions. The object-oriented program was designed to run for a wide variety of examples, and it has been analyzed to determine timing and complexity measures. The program is easily modifiable, allowing one to input a relatively small amount of data that describes **K**, as well as a positive integer r for the permutation group S_r. The output is an incidence matrix for a directed graph Γ, describing a certain *shift of finite type* denoted by Φ_r(**K**). Graphically, elements of the shift are in one-to-one correspondence with bi-infinite walks on Γ. Algebraically, the elements of Φ_r(**K**) are in one-to-one correspondence with r-permutation representations of **K**; i.e., homomorphisms from **K** to S_r. The shift Φ_r(**K**) was shown in [SW] to be an isomorphism invariant of **K**; as a consequence, it can be shown that two finite **Z**-dynamically presented groups are non-isomorphic simply by showing that their

corresponding shifts differ (are not conjugate) for some r. A variety of algorithmic techniques exist to perform this last task[LM].

The shift $\Phi_r(\mathbf{K})$ contains very important information about the group **K**, information that until now has been very difficult or impossible to obtain. For example, a description of $\Phi_r(\mathbf{K})$ enables one to find all index-r subgroups of **K**. Moreover, [SW] shows how to obtain a set of generators for any such subgroup. However, what is most striking about the algorithm is that the shift $\Phi_r(\mathbf{K})$ is described by a directed graph that is finite.

The algorithm of [SW] can be evaluated, by hand, only for relatively simple groups **K** and very small r (e.g., $r = 2$ or 3). However, it is important to be able to implement the algorithm on large examples for two reasons. First, understanding nontrivial examples provides intuition about the algorithm and helps form reasonable conjectures. Second, the algorithm's applications are only as useful as the calculations are practical. The program will be run for groups **K** that arise naturally in knot theory, and the results will be tabulated. This study hopes to demonstrate that the new invariant $\Phi_r(\mathbf{K})$ can not only distinguish all of the groups corresponding to the knots in a standard table ([Ro], [BZ]), but that it will do so with relatively small values of r.

At some level every group measures symmetry. The invariant $\Phi_r(\mathbf{K})$, where the group **K** comes from a knot, hopefully describes characteristics of the symmetries of the knot itself (i.e., the ways in which the knot can be positioned in space so that a rotation of some order will carry the knot into itself). However, the connection is far from clear at

this time. Data collected from this study will provide insights into new directions for continuing research in Knot theory and Algebraic Group theory.

A special set of examples involving a large family of interesting knots, pretzel knots K(2p+1, 2q+1, 2r+1), will be used in this research. When p, q and r are non-negative, the knot is alternating. A goal of this study was to find an example of an alternating knot such that the connected component [0] of the trivial representation 0 in $\Phi_r(\mathbf{K})$ contains more than 0 for some r. We expect that such an example will help answer open questions about the groups of alternating knots.

2. BACKGROUND

A group K is a set together with a product $K \times K \to K$, $(g_1, g_2) \to g_1 g_2$ satisfying certain properties (associativity, existence of inverses and identity elements.) The set of permutations of $\{1, ..., r\}$ with product given by ordinary function composition is a finite group called the *symmetric group* S_r. A well known theorem of Cayley says that every finite group is a subgroup of S_r for r sufficiently large [BC]. Group theory is an old area of mathematics with important applications in the sciences.

A homomorphism f from one group K_1 to another K_2 is a function $h : K_1 \to K_2$ that preserves products; i.e., $h(g_1 g_2) = h(g_1)\ h(g_2)$. One way to study an infinite group K is to understand all homomorphisms from K into an arbitrary finite group or equivalently, into an arbitrary symmetric group. Such homomorphisms are called *r-permutation representations* of K [BC].

A presentation of a group K is a pair $< a, b, c, ... \mid r, s, t, ...>$ consisting of generators a, b, c, ... for K and defining relators r, s, t,... More precisely, a, b, c, ... are abstract symbols while r, s, t, ... are words expressed in those symbols or their (formal) inverses such that K is isomorphic to the free group on a, b, c,... modulo the smallest normal subgroup containing r, s, t,...[LS]. If the presentation consists of only finitely many generators and finitely many relators, then the presentation is said to be "finite" and the group K is called "finitely presented."

There exist groups that are not finitely presented. In fact, many groups require infinitely many generators and infinitely many relators in any presentation. Such group are notoriously difficult to understand. However, members of an important class of such groups, so-called "*finite Z-dynamically presented groups*" **,** admit presentations that have such a high degree of structure that algorithms to represent them can be developed. In some sense these groups are "on the boundary" between finitely presented groups and arbitrary non-finitely presented groups.

The term *finite Z-dynamic presentation* was introduced by Hausmann and Kervaire in [HK]. A finite **Z-**dynamic presentation for a group **K** is a presentation $< \mathbf{a}_{i,t} \mid \mathbf{r}_{j,t} >$ consisting of generators $\mathbf{a}_{i,t}$, where i ranges over a finite set, and relators $\mathbf{r}_{j,t}$, where j ranges over a (possibly different) finite set. The index t ranges over all integers. Of course, the relators $\mathbf{r}_{j,t}$ are words in the generators $\mathbf{a}_{i,t}$**,** but we require that $\mathbf{r}_{j,t+k}$ be the word obtained from $\mathbf{r}_{j,t}$ by shifting all of the second indices of generators that occur by k. For example: if $\mathbf{r}_{0,0}$ is the word $\mathbf{a}_{3,1}\mathbf{a}_{2,-1}$, then $\mathbf{r}_{0,1}$ must be equal to $\mathbf{a}_{3,2}\mathbf{a}_{2,0}$.

Finite **Z-**dynamically presented groups arise commonly in geometric topology. There are many unsolved problems involving such groups. Drs. D. Silver and S. Williams discovered an algorithm for finding all finite permutation representations for a given finite **Z-**dynamically presented group **K**. The ideas they used came from Symbolic Dynamical Systems, a field closely related to Information Theory. The permutation representations of **K** into S_r correspond to the bi-infinite walks on a certain finite directed graph that is described in [SW]. The representations form the elements of a rich

dynamical structure called a *shift of finite type* (or *shift*). As a result, any numerical quantity associated to the shift is an invariant of the group **K.**

An overview of symbolic dynamics is presented here, this section is not intended to be a complete presentation of this extensive field of study. More details can be found in [LM]. Let $\mathcal{A}$ be any finite set of symbols. We call $\mathcal{A}$ an *alphabet* and its elements *letters*. The *full $\mathcal{A}$-shift* is the collection of all bi-infinite sequences of symbols from $\mathcal{A}$ and is denoted by $\mathcal{A}^Z$ where $\mathcal{A}^Z = \{ x = (x_i)_{i \in Z} : x_i \in \mathcal{A}$ for all $i \in Z \}$; i.e., $\mathcal{A}^Z$ stands for the set of all function from Z to $\mathcal{A}$. In particular, the full shift over the alphabet $\{ 0,1, \ldots r-1 \}$ is called *full r-shift*. The *shift map* $\sigma : \mathcal{A}^Z \rightarrow \mathcal{A}^Z$ maps a point x to the point $y = \sigma(x)$ whose i^{th} coordinate is $y_i = x_{i+1}$; i.e., taking any $\rho = (\rho_j)$, $\rho_j \in \mathcal{A}$, to $\rho' = (\rho'_j)$, where $\rho'_j = \rho_{j+1}$ [LM] [SW].

A *shift space* is a subset χ of a full shift $\mathcal{A}^Z$ such that $\chi = \chi_{\mathcal{F}}$ for some collection $\mathcal{F}$ of forbidden blocks over $\mathcal{A}$. By a block we mean any finite sequence of letters. The forbidden blocks $\mathcal{F}$ are a collection of blocks over $\mathcal{A}$. For any such $\mathcal{F}$, we define $\chi_{\mathcal{F}}$ to be the subset of sequences in $\mathcal{A}^Z$ which do not contain any block in $\mathcal{F}$. A shift space that can be described by a finite set of forbidden blocks is called a *shift of finite type*; i.e., a shift space χ having the form $\chi_{\mathcal{F}}$ for some finite set $\mathcal{F}$ of blocks [LM].

A shift of finite type is an example of a dynamical system. A *dynamical system* is a pair (X, σ) consisting of a topological space X and a homomorphism (i.e., a continuous mapping with a continuous inverse). A mapping $f : (X, \sigma) \rightarrow (X', \sigma')$ of dynamical systems is a continuous function $f : X \rightarrow X'$ for which $f \circ \sigma = \sigma' \circ f$ (i.e., f commutes

with the shift maps.) Two dynamical systems are *conjugate* if there also exists a mapping $g : (X', \sigma') \rightarrow (X, \sigma)$ such that $f \circ g = g \circ f$ are the identity functions.

Given a group **K** with a suitable presentation (a *finite* ***Z****-dynamic* presentation) and given a positive integer r a finite directed graph Γ can be constructed. The details of the construction are given in [SW]. The vertices of the graph are labeled by n-tuples (fixed n) of permutations; i.e., elements of S_r. These permutations can be thought of as being assigned to a corresponding number of generators in the (infinite) presentation of **K**. A directed edge connects one vertex to another if the second n-tuples of permutations, regarded as being assigned to generators with suitably shifted indices, are compatible with that finite number of relations in the presentation that involve the combined sets of generators. The important point to be made here is that the total number of n-tuples is finite, and the procedure for deciding when two vertices are connected by a directed edge is algorithmic.

From the directed graph Γ a shift of finite type, $\Phi_r(\mathbf{K})$ is obtained. Elements of the shift are in one-to-one correspondence with bi-infinite walks in Γ. The shift map σ is the obvious one - it maps any walk to the same walk but with a different starting place. The topology of the shift is described in [SW] and will not be of concern here.

An examination of the process by which the shift $\Phi_r(\mathbf{K})$ is obtained shows that shift elements are in one-to-one correspondence with r-permutation representations of **K**. In fact, one can give a complete algebraic description of $\Phi_r(\mathbf{K})$. This is done in [SW].

There is a large body of literature describing algorithms for working with finite groups [Si1] [BC] [Bu]. One might, for example, use this implementation for finding

$\Phi_r(\mathbf{K})$ to identify the various finite groups (subgroups of S_r) into which **K** is mapped under the elements of $\Phi_r(\mathbf{K})$. It would be interesting to know when these groups are simple, abelian, etc. A recent paper of Kantor [Ka] gives an algorithm that tests simplicity of any subgroup of S_r specified in terms of a generating set (as are the groups that will be seen); this technique enables us to find all of the composition factors and a composition series for that subgroup.

A natural question is: How large can we allow the value of r to be and still expect our program to run efficiently? A recent paper of Leon gives reason to believe that the value of r can be very large [Le]. Leon develops a technique for computing in permutation groups S_r for large r, using the idea of successive refinement of ordered partitions, introduced by B. McKay [Mc] in connection with the group isomorphism problem.

Finally, a paper by Butler and Cannon [BC] presents an algorithm for computing Sylow subgroups of large permutation groups.

3. THEORETICAL BASIS

Assume that G is a finitely presented group and $\chi : G \rightarrow Z$ is an epimorphism (a homomorphism of Group G into group Z which is onto mapping). The kernel of χ is denoted by K_χ. An augmented group system is a triple (G, χ, x) consisting of a finitely presented group G, an epimorphism $\chi : G \rightarrow Z$, and a distinguished element $x \in G$ such that $\chi(x) = 1$. It can be shown that any augmented group system (G, χ, x) determines a sequence of shifts of finite type (Φ_r, σ_x), where r is any positive integer. The elements of (Φ_r, σ_x) are the representations of K_χ in the symmetric group S_r. By using the tools of combinatorial group theory, Drs. Silver and Williams developed an algorithm for determining the shifts (Φ_r, σ_x) [SW] .

Let r be a positive integer. The representation shift associated with (G, χ, x) is the dynamical system (Φ_r, σ_x) consisting of the space Φ_r of representations $\rho : K_\chi \rightarrow S_r$, and the mapping σ_x described by $\sigma_x(\rho)(g) = \rho(x^{-1} g x)$, for all $g \in K_\chi$.

Of concern in this research is the case $K = K_\chi$, where χ is the epimorphism of some augmented group system (G, χ, x). Given any finite presentation $< x_1, \ldots, x_n \mid r_1, \ldots, r_m >$ for G a well-known algorithm from combinatorial group theory, the Reidemeister-Schreier method, enables one to find a presentation (possibly infinite)

for K_χ. Since this procedure is not developed further in this thesis, only a brief description is given below[SW]:

The distinguished element x corresponds to some word w in the generators x_i. A new symbol x and relator x = w is added to this presentation for G. Next each generator x_i is replaced by $a_i = x_i x^{-\chi(xi)}$, finally the old symbols x_i and the relators $a_i = x_i x^{-\chi(xi)}$ are eliminated. For each i = 1, …, n and $j \in Z$, The element $x^{-j} a_i x^j$ is denoted by the symbol $a_{i,j}$. Clearly, each $a_{i,j}$ is an element of K_χ. In fact, it is not difficult to show that these elements generate K_χ. A set of relators for K_χ is obtained by rewriting each of $x^{-j} r_1 x^j$, …, $x^{-j} r_m x^j$ as a word in the
$a_{i,j}$, a rewriting that is possible since the exponent sum of x is zero.

The theoretical basis of this algorithm is based on the following theorem[SW] and since the proof describes constructions required in the C++ implementation, the proof will be included in its entirety.

THEOREM 1. Assume that (G, χ, x) is an augmented group system. For any positive integer r, the associated representation shift (Φ_r, σ_x) is conjugate to a shift of finite type.

PROOF. Recall that G has a presentation of the form $< x, a_1, \ldots, a_n \mid r_1, \ldots, r_m >$ such that $\chi(x) = 1$ and $\chi(a_1) = \ldots = \chi(a_n) = 0$. Also, K_χ has a presentation

$$< a_{i,j} \mid R_j, 1 \le i \le n, j \in Z >, \qquad (I)$$

where the symbols $a_{i,j}$ denote generators $x^{-j} a_i x^j$, and R_j is $\{ x^{-j} r_1 x^j, \ldots, x^{-j} r_m x^j \}$ written as words in the generators. Note that R_{q+t} is obtained from R_q by adding t to the second subscript of every symbol in R_q. Assume that the words in R_0 (and hence in each R_q) are

reduced and cyclically reduced; i.e., no generator appears next to its inverse, and no word in R_0 ends with the inverse of the generator with which it begins. Replacing the original relators r_j by suitable conjugates $x^{-tj} r_j x^{tj}$, one can assume that if R_0 contains $a_{i,j}$ for some j, then R_0 contains no $a_{i,j}$ with $j < 0$. If $a_{i,0}$ occurs in R_0, then let M_i be the largest value of j such that $a_{i,j}$ occurs. If $a_{i,0}$ doesn't occur in R_0, then let M_i be zero.

From the representation (I) of K_χ, a presentation of some group $H_0 = < a_{1,0}, a_{1,1}, \ldots, a_{1,M1}, a_{2,0}, \ldots, a_{n,Mn} \mid R_0 >$ can be obtained. Since in K_χ the generators of H_0 might satisfy relators other than those that are consequences of R_0, the group H_0 is in general not a subgroup of K_χ. Nevertheless, H_0 is valuable for studying the permutation representations of K_χ. Abbreviate the set of generators $\{ a_{1,0}, a_{1,1}, \ldots, a_{1,M1}, a_{2,0}, \ldots, a_{n,Mn} \}$ by the symbol A_0, and let $A_t = \{ a_{1,t}, a_{1,1+t}, \ldots, a_{1,M1+t}, a_{2,t}, \ldots, a_{n,Mn+t} \}$. Combining the representations $< A_t \mid R_t >$ as t ranges over Z produces the presentation (I) of K_χ.

Deleting $a_{1,M1}, \ldots, a_{n,Mn}$ from A_0 produces a subset that is denoted by $A_{0.1}$. Similarly $A_{0.2}$ is the result of deleting $a_{1,0}, \ldots, a_{n,0}$ from A_0.

Let $\mathbb{A}$ denote the set of all representations of H_0 in S_r. Such representations are precisely those function $\rho_0 : A_0 \rightarrow S_r$ such that the m equations $\rho_0(r_i) = \text{id}$, $r_i \in R_0$, hold in S_r. In particular, $\mathbb{A}$ is a finite and computable set. Construct a directed graph Γ with vertex set $\mathbb{A}$. Draw a directed edge from vertex ρ_0 to vertex ρ_0' if and only if $\rho_0(a_{i,j+1}) = \rho_0'(a_{i,j})$ for each $a_{i,j} \in A_{0,1}$. The graph Γ determines a shift of finite type X with alphabet $\mathbb{A}$.

Any element $\rho = (\rho_j)$ of X determines a well-defined function $\cup_{t \in Z} A_t \rightarrow S_r$ by

$a_{i,j} \to \rho_t(a_{i,j-t})$ if $a_{i,j} \in A_t$. This function maps each relator $x^{-t} r_j x^t$ in R_t to the element $\rho_t(r_j)$ which is the identity (since ρ_t is a homomorphism), and hence it induces a homomorphism from $K = \langle A_t \mid R_t, t \in Z \rangle$ to S_r. It is easy to check that this determines a continuous shift-commuting function f from X to (Φ_r, σ_x).

Conversely, any presentation $\rho : K \to S_r$ determines a function $A_0 \to S_r$ for each t by $a_{i,j} \to \rho_t(a_{i,j+t})$. The function maps each relator r_j in R_0 to $\rho(x^{-t} r_j x^t)$, the identity element of S_r, and hence it induces a homomorphism ρ_t from H_0 to S_r. Clearly, $\rho = (\rho_t)$ is an element of the shift X, and a continuous shift-commuting function $g : (\Phi_r, \sigma_x) \to X$ is obtained. Since f and g are inverses, the shifts (Φ_r, σ_x) and X are conjugate. ■

Shifts of finite type X and X′ are finitely equivalent if there exists a shift of finite type that maps onto each by mappings that are finite-to-one. The entropy of a shift can be defined as Lim Sup $\{1/N \log |\beta_N|\}$, where $|\beta_N|$ is the number of allowable N-blocks of the shift. When the shift is described by a directed graph Γ, its entropy is log λ, where λ is the Perron eigenvalue of the adjacency matrix of Γ [LM]. Conjugate shifts have the same entropy. In fact, finitely equivalent shifts also have the same entropy.

THEOREM 2. Assume that (G, χ, x) and (G, χ, y) are augmented group systems that differ only by the choice of distinguished elements x and y. Then for each r > 0, the associated shifts (Φ_r, σ_x) and (Φ_r, σ_y) are finitely equivalent [SW].

The proof of this theorem is straightforward and the details can be found in [SW]. A final definition is required before a conclusion is made. A group system is a pair (G, χ) consisting of a finitely presented group G and an epimorphism $\chi : G \to Z$. Two group system (G, χ) and (G′, χ′) are isomorphic if there exists a group isomorphism $h : G \to G'$

such that $\chi = \chi' \circ h$. The entropy of the shifts (Φ_r , σ_x) is used to define a sequence of numerical invariants for any pair (G, χ) because of the following corollary:

COROLLARY. Let (G, χ, x) be an augmented group system, and let r be a positive integer. The entropy $h(\Phi_r)$ of the associated representation shift (Φ_r , σ_x) is an invariant of the group system (G, χ); i.e., the entropy depends only on the isomorphism class of the group system [SW].

Very little software exists for working with infinite non-abelian groups. The interactive program that was developed will benefit researchers in the areas of both group theory and topology.

An interactive program based on [SW] for constructing " representation shifts" was designed by implementing the algorithm of Drs. Silver and Williams. The basic strategy is to convert a "finite **Z**-dynamic" group presentation into a directed graph for each positive integer *r*. The bi-infinite walks on the graph are in a natural one-to-one correspondence with the elements of a *shift of finite type*. Information about the shift (e.g. entropy) provides us information about the original group.

One of the systems outputs will be an incidence matrix that can determine the desired shift $\Phi_r(\mathbf{K})$. A variety of invariants for **K** can then be found using this matrix. For example, the logarithm of the spectral radius of the matrix is the so-called entropy of the shift; this number is an invariant of the group **K** in the sense that any two finite **Z**-dynamically presented groups with different entropies must themselves be different. Groups with different entropies for some *r* are necessarily different (non-isomorphic). Based on our research, this is the first invariant of a large class of groups defined using

ideas from symbolic dynamical systems, and consequently, this algorithm should attract the attention of mathematicians working in that field.

The program has been used to compute a sequence of new knot invariants. Varying r produces an infinite sequence of numerical invariants for **K.** Such a sequence is mathematically new, and it is conjectured that the sequence will be a highly effective means to show that different finite **Z-**dynamically presented groups are in fact different. Any knot k determines a finite **Z-**dynamically presented group **K,** the commutator subgroup of the group of k, and so for any positive integer *r* an entropy can be found; the entropy of the shift $\Phi_r(\mathbf{K})$. The research suggests that this sequence of entropy invariants is very sensitive. The tool developed in this thesis has been used to compute an initial segment of the sequence for each of the knots tabulated in [Ro]. If, as is expected, a relatively short segment is capable of distinguishing all of these "9-crossing knots," then the entropy invariant will be an important new tool for knot theorists.

Much of the existing software used for solving mathematical problems is either too difficult to use or too general to cover specific mathematical areas, such as knot theory and group theory. Under certain circumstances, a tool for solving relatively narrow problems in a specific mathematical field would be very useful. The tool once implemented would be used to solve an unsolved problem or conjecture. Finally, the output from this tool can be used to make reasonable conjectures about infinite non-abelian groups.

4. ALGORITHMS AND IMPLEMENTATION

The research that was undertaken in this project was to investigate their apparent connections between the mathematics described in Chapter 2 and 3 and the object-oriented programming paradigm described by Classes and Class library functions. Thus, the algorithm was developed and implemented using a BORLAND C++ compiler running on an IBM 486PC with 8 Mbytes of memory. When completed, the complexity and size of the algorithm required a system with a larger amount of memory, so the code was ported to the UNIX platform and compiled using the GNU C++ compiler (g++) and debugged with another GNU tool called gdb. The portability of the code across platform was demonstrated.

Object Classes

C++ was chosen to implement this algorithm because of its support for: encapsulation, multiple instances, overloading operators, inheritance, polymorphism, and dynamic binding. The C++ programming language was designed and implemented by Bjarne Stroustrup of AT&T Bell Lab as a successor to C. While borrowing several key ideas from SIMULA 67 and ALGOL 68, C++ retains compatibility with existing C programs and possesses the efficiency of C. However, C++ adds many powerful new features, making it suitable for use in a wide range of applications from games to operating systems. C++ not only corrects many C deficiencies, it also introduces many

completely new features that were designed into the language to provide support for data abstraction and object-oriented programming. Following are some of the more prominent new features of C++:

- **classes, the basic language construct for creating programmer defined data types that are called abstract data types;**
- **member variables, describe the data in abstract data types, and member functions, define the permissible operations on the data type;**
- **operator overloading, allows the programmer to give additional meaning to most operators so that they can be used with one's own data types, thereby making those data types easier to use; while function name overloading, similar to operator overloading, lets one reduce the need for unusual function names, making code easier to read;**
- **programmer-controlled automatic type conversion, allows one to blend their own types with the fundamental data types provided by C++ languages;**
- **derived classes, inherit member variables and member functions from their base classes, and can be differentiated from their base classes by adding other member variables and member functions;**

virtual functions, let a derived class redefine member functions inherited from a base class. One can then write very general programs that are obvious to the specific classes of the objects they manipulate; through dynamic binding, the run time system will choose the function appropriate to a particular class. [KSP]

A class defines a data type. In a computer science sense, a type consists of both a set of states and a set of operations that transition between those states. For example,

“int” is a ‘type’ because it has both a set of states and it has operations like “adds two ints” or “int*int”. In exactly the same way, a class provides a set of (usually public) operations, and a set of (usually non-public) data bits representing the abstract values that instances of the type can have. This class concept surprisingly fits well with the class concept in group theory. One can consider a symmetric group to be a class that arises from the set, S_n, of all one-to-one mappings of {1,2, …, n} onto { 1,2, …, n} with operations like inverse and composition.

The Strategy

The basic strategy to convert a finite Z-dynamically presented group into a directed graph for each positive integer r is as follows:

- The vertices of the graph are labeled by n-tuples (fixed n) of elements of S_r, subject to certain constraints imposed by relators in the group presentation.
- There will be a directed edge from vertex (π , ω) to vertex (γ , ψ) if and only if $\omega = \gamma$, where $\pi , \omega, \gamma, \psi \in S_r$.
- Bi-infinite walks on the graph are in a natural one-to-one correspondence with the elements of a shift of finite type $\Phi_r(K)$.
- The walks are also in one-to-one correspondence with the homomorphisms of K into S_r (i.e., r-permutation representations of K) This provides another way to think about the shift.
- Isomorphic groups will determine equivalent shifts.

- Given any knot k, its associated finite Z-dynamically presented group K now determines a shift $\Phi_r(K)$ for each positive integer r = 2, 3, … . We obtain a sequence $\Phi_2(K)$, $\Phi_3(K)$, ... of new knot invariants.

There are four main C++ classes that constitute the basis of this implementation, each class contains a rich set of operations. Some class features, such as operator overloading, programmer-controlled automatic type conversion, and virtual functions are widely used in this implementation. Following is a detailed descriptions of these main classes:

- *Symmetric group class* is the basis class for handling the permutation calculations in a given symmetric group for various positive r. A new bracket notation is used to replace the traditional cycle notation, for example, [1 2 3] denotes the identity and [3 2 1] represents (2)(1 3). Typical operations in this class are:

    1. operator " * " : Returns the composition of two permutations.
    2. operator " = " : Assigns one permutation to another.
    3. power(n) : Takes the power n of a permutation ($n \in Z$) without changing the original one. e.g. power(-1) is the operation for taking inverses.
    4. operator " ++ " : Pre-increment of a permutation in which the current value of a permutation is returned and then set to the next permutation in increasing order.

5. operator “ ++ “ : Post-increment of a permutation in which the permutation is set to the next permutation in increasing order and returns that value.
6. isIdentity() : Returns TRUE if a permutation is the identity.
7. operator “ << “ : Prints the permutation in a new form.
8. operator “ == “ : Returns TRUE if two permutations are equivalent.
9. operator “ >, >=, <, <=” : Compares two permutations.

- The second class is called the *general symmetric group class* and is built upon the symmetric group class. It combines single permutations into a collection of permutations that will become vertices in the desired output graph. i.e. ([1 2 3], [2 3 1]) would be a valid node (vertex) formed by grouping the permutation [1 2 3] and [2 3 1] together. By operator overloading and inheritance, one can easily modify needed operations of symmetric group classes for this new general symmetric group class without massive amounts of code being rewritten. Operators like “ == ”, “ > “, “ >= “, “ < “, and “ <= “ are written for general symmetric group classes by slightly modifying the existing code.
- The third important class is the *permutations list class* which enables one to find the “good nodes” out of hundreds of possible nodes. By “good nodes”, we mean all those n-tuples of elements of S_r that satisfied the constrains imposed by relators in the group representation. This class is different from an ordinary linked list class in the way it manipulates data. The data are a collection of single permutations. Each node in the list consists of two fields:

Data and Pointer. The data are elements of a general symmetric group class and the pointer identifies the next node in the list. Special attention is required to make the program run correctly and quickly since it must access this general symmetric group class from almost every function in the permutations list class. There are four critical functions in this class:

1. push_end_new() : takes an element of the general symmetric group class and puts it into a list if the element is not already in the list. Note that the elements in the list are not all "good nodes" since there might be some dead-end-nodes that will be eliminated later. This function ensures that there are not any duplicate nodes in the list.
2. rmDead1() & rmDead2() : As the name suggest, these two functions remove all of the "dead-end-nodes". There are two types of dead-end-nodes in our case: one with only outgoing edges and no incoming edges; and one with only incoming edges and no outgoing edges.
3. fillArray() **:** After all the good nodes are found, they are put into an array to build the adjacent matrix that describes the final output graph.

- Finally, in order to find the strongly connected component from the adjacent matrix the union-find algorithm was implemented. This operation results in the construction of the fourth C++ classes; the *disjoin sets class*. The union-find algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. More precisely, the problem is to carry out three types of operations on disjoint sets: makeset (creates a new set); find (locates the set containing a given element); and link (combines two sets into one)

[Ta]. In our case, these operations are applied to the set { 1, …, N }, where N is the number of rows (or columns) in the adjacent matrix. In order to identify the sets, assume that the algorithm maintains, within each set, an arbitrary but unique representative called the canonical element of the set. In this implementation, the smallest member in the set is the canonical element. The operation find(x) returns the canonical element of the set containing element x. The operation link(x, y) forms a new set that is the union of the two sets whose canonical elements are x and y while destroying the two old sets, and then it selects and returns a canonical element for the new set. The general algorithm is implemented in TGART (Temporal Graph Algorithm Research Tool) written by Dr. Thomas Hain from the school of CIS of the University of South Alabama [Ha]. A simplified algorithm of union-find is implemented here for our special need.

The Algorithm

After building the above classes, it was, as anticipated as, a relatively easy task to put them together to complete the implementation. The complete procedure to implement Drs. Silver and Williams algorithm [SW] is as follows:

1. Given a presentation with n generators**,** test each generator to see if any node, an n-2 tuple, satisfies the relator. Next link those tuples that satisfy the relator to a list and discard the others. Note that these nodes are only potential good nodes, so step 2 below is used to eliminate other dead-end-nodes. There are no duplicate elements in the list since only nodes that are different from ones already in the list are added.

Also the list is sorted by considering each node to be represented by a unique positive integer. For example, consider the node [1 2 3] to be represented by 123 (an integer) and [1 3 2] is 132. Thus, the node ([1 2 3], [1 3 2]) represented by 123132 is less than the node ([1 2 3], [3 2 1]).

2. Find the "dead-end-nodes" and remove them from the list. By the "dead-end-nodes", we mean those nodes that have only outgoing edges or only incoming edges.
3. Assign the remaining nodes that were not removed in step 2 to an array for building the adjacent matrix. It is much easier and faster to access an array than a list to find the remaining information.
4. Build the adjacent matrix from the array that was obtained in step 3, note that the relators must be tested again. The matrix is a square N by N array that is also sparse, where N is the total number of good nodes found in step 3.
5. Find all possible paths from one node to another by taking loops $K = \log_2 (N-1) + 1$ times. For each loop, the matrix is squared.
6. Construct the strongly connected components from the matrix in step 5 by using the simplified union-find algorithm.

Analysis

In step 1, there are a total of $(r!)^n$ permutations that must be inspected, where n is the number of generators and r is the degree of the given symmetric group. Obviously,

execution cost increase tremendously as n or r increases. We can eliminate up to 80% of the useless nodes in this step. Normally, the list can be constructed in a reasonable period of time despite the large amount of calculations that are required.

The cost of step 2 is determined by the total number of nodes in the list obtained from step 1. The entire list is traversed while the dead-end-nodes are removed. The worst case occurs when most of the dead-end-nodes are located at the end of the list. In that case the algorithm traces down to the end of the list to remove a dead-end-node. Fortunately, the worst case does not happen often. Actually, a two-way scanning is performed to ensure all the dead-end-nodes are found and removed. First, the algorithm tries to find if any given node "connects" to any other node. If so, it is kept. Otherwise, it is removed from the list. Next the entire list is scanned again to see if any other node "connects" to a given node, and the process is repeated. By this two-way scanning, the implementation guarantees there are no dead-end-nodes remaining. The total cost of this two-way scanning is approximately O(N) since one-way sequential searches uses on average N/2 comparisons for both successful and unsuccessful searches [Se].

Step 3 is trivial and fast. As one might expect, the bottlenecks are in steps 4 and 5 where we attempt to build an adjacent matrix and raise it to a power. At first, a sparse matrix class was used to store only the information related to non-zero elements. A large amount of memory can be saved by using this class. Unfortunately, the implementation runs slowly especially when the degree of the given symmetric group is larger than 5. The time to complete a single run is on the order of several days. This approach seemed impractical because of its inefficiency, even though it worked. Later, a second approach was implemented that used an ordinary matrix instead of a sparse matrix. That is a

tradeoff between memory space and speed was made. The new approach did run much faster and the performance was acceptable even if it had to run more than 10 hours for the case r = 5. During the experimentation phase, difficulties to complete some cases were encountered because of insufficient memory in the Sun environment. The problem occurred when the total number of good nodes was large, e.g. when N was close to 10,000. This situation was expected: but what about when N is close to 100,000? It is not impossible to have numbers of this magnitude since the size of the adjacent matrix grows rapidly as r gets larger. Although the execution speed is greatly improved by using this new approach, much CPU time is still spent in steps 4 and 5. An alternative method, where we are primarily concerned if [0] contains anything different than node 0, was found to bypass the matrix calculations. Details of this approach are discussed in the next chapter.

The union-find algorithm is very efficient as expected. Step 6 requires O(N) time per find in the worse case, N is the total number of good nodes [Ta]. The proportion of CPU time spent at this step is very small even when N is large. In the following chapter, experimental results and performance characteristics under different parameter settings are discussed.

The Code

The implemented algorithm, see APPENDIX D, is a general program and with only a few changes can be used to run any specified Knot representation. The source code is highly reusable and most of it can be used without any changes. A brief summary of each file is as follows:

1. main.cc : Main program uses all the individual module.

2. symgrp.h : Header file for symgrp.cc.

3. symgrp.cc : Source code for Symmetric Group Class.

4. gensymgrp.h : Header file for gensymgrp.cc

5. gensymgrp.cc : Source code for General Symmetric Group Class.

6. plist.h : Header file for plist.cc

7. plist.cc : Source code for Permutations List Class.

8. uf.h : Header file for uf.cc.

9. uf.cc : Source code for Union-Find Class.

10. timer.h : Run-time recording header file.

An attempt has been made to create a straight-forward user interface. However, due to the unlimited member of representations, it was not possible, except in a few special cases to develop a complete user interface solution. These cases are discussed in Chapter 5 where the concept of Pretzel knots is analyzed. At this time, the program will work for any knot with the following changes:

1. Line 52 and Line 155 of main.cc must be changed according to a specific knot representation.

2. There are two "remove dead-ends" functions in plist.cc, rmDead1 (Line 158) and rmDead2 (Line 185), that must be changed for a specific knot representation.

One example of code modification is found in Table 1.

Representation	$< a_0, a_1, b_0, b_1 \mid b_0 a_0^{-1} a_0 a_1 b_1^{-1} >$	$< a_0, a_1, b_0, b_1 \mid b_0 a_0^{-1} b_1^{-1} a_1 b_1 >$
Line 52 in main.cc	b0*a0.power(-1)*a0*a1*b1.power(-1)	b0*a0.power(-1)*b1.power(-1)*a1*b1
Line 155 in main.cc	n_array[i][1] *(n_array[i][0]).power(-1) *n_array[i][0] *n_array[j][0] *(n_array[j][1]).power(-1)	n_array[i][1] *(n_array[i][0]).power(-1) *(n_array[j][1]).power(-1) *n_array[j][0] *n_array[j][1]

Line 158 in plist.cc	(pnp->next->p)[1] *(pnp->next->p)[0].power(-1) *(pnp->next->p)[0] *(np->p)[0] *(np->p)[1].power(-1)	(pnp->next->p)[1] *(pnp->next->p)[0].power(-1) *(np->p)[1].power(-1) *(np->p)[0] *(np->p)[1]
Line 185 in plist.cc	(np->p)[1] *(np->p)[0].power(-1) *(np->p)[0] *(pnp->next->p)[0] *(pnp->next->p)[1].power(-1)	(np->p)[1] *(np->p)[0].power(-1) *(pnp->next->p)[1].power(-1) *(pnp->next->p)[0] *(pnp->next->p)[1]

Table 1
Example for Code Modification

There are a total of four changes that must be made to run any specific knot representation. The changes are quite straight-forward and can be easily developed for many different problems. Specifically:

1. Additional generators can be added without affecting other parts of the code.
2. The notation used, pnp->next->p, means a pointer to the next node in the 'Good Node' list while np->p points to the current node that is being considered. For [a0, b0] -> [a1, b1] in this example, see Table 1:

   a0 corresponds to (pnp->next->p)[0].
   a1 correspond to (pnp->next>p)[1].
   b0 corresponds to (np->p)[0].
   b1 corresponds to (np->p)[1].
3. The symmetry of Line 158 and 185 means that the changes involves only swapping pnp->next->p and np->p.

5. EXPERIMENTATION AND ANALYSIS

A famous family of knots, the so called pretzel knots, was carefully analyzed in this study and used to establish experimental results. Pretzel knots are a very rich family of knots with many interesting aspects. The commutator subgroup K of the group of a general pretzel knot k(2p+1, 2q+1, 2r+1) has presentation:

$$< a_0, a_1, b_0, b_1 \mid (b_0 a_0^{-1})^{-q} a_0^{p+1} a_1^{-p} (b_1^{-1} a_1)^{-q-1}, b_0^{r} (b_0 a_0^{-1})^{q+1} (b_1^{-1} a_1)^{q} b_1^{-r-1} >$$

As usual, a_0, a_1, b_0 and b_1 are generators while $(b_0 a_0^{-1})^{-q} a_0^{p+1} a_1^{-p} (b_1^{-1} a_1)^{-q-1}$ and $b_0^{r} (b_0 a_0^{-1})^{q+1} (b_1^{-1} a_1)^{q} b_1^{-r-1}$ are relators, p, q and r are integers. This is a large family, with every choice of p, q, and r corresponding to some knot. Of special interest are the choices of p, q, r that are all non-negative, producing special knots called “alternating”. In these cases, the crossings are alternately “over” and “under” each other. While there were known examples of non-alternating knots for which [0] contains something more than node 0, it was unknown whether any alternating knot had this property. Using this toolkit, we were able to produce the first such example.

Some notational conventions used in this research are now defined. In the following output, R stands for the degree of the symmetric group; p, q and r are any integers as shown in the representation of pretzel knots. [X] represents a “class” of a strongly connected component where X is the “smallest node” in that component, e.g. [0] is a strongly connected component “class” including node 0 and other nodes, if any. We also number each good node by an integer; this scheme is a one-to-one correspondence and in increasing order beginning from 0. The algorithm described in the

chapter 4 can determine if [0] contains anything besides node 0. A complete result produced by the implementation is shown as follow:

```
Script started on Sat Oct 28 02:01:29 1995

Input the necessary parameters you'd like to run:

Please enter the integer R:
3
Please enter the integer p:
2
Please enter the integer q:
0
Please enter the integer r:
6

Linking up the potential good nodes to a list:
Please wait.....
DONE.

The # of potential good nodes in permutations list is: 24

The time used to get the permutations list is (Step 1): 0.05

Scanning the list and removing the dead-end-nodes:
Please wait.....
DONE.

There are 16 nodes left after the first scan.
That was only half-way done, scan the list again:

Scanning the list again and removing the dead-end-nodes:
Please wait.....
DONE.

There are 16 GOOD NODES left after the second scan.
The two-way scanning completed.

The time used to remove all the dead-end-nodes is (Step 2):0.02

Put all the Good Nodes to an array in increasing order:
Please wait....
0( [1 2 3], [1 2 3] )
1( [1 2 3], [1 3 2] )
2( [1 2 3], [2 1 3] )
3( [1 2 3], [3 2 1] )
4( [1 3 2], [1 2 3] )
5( [1 3 2], [1 3 2] )
6( [2 1 3], [1 2 3] )
7( [2 1 3], [2 1 3] )
8( [2 3 1], [2 1 3] )
9( [2 3 1], [3 2 1] )
10( [2 3 1], [1 3 2] )
11( [3 1 2], [3 2 1] )
12( [3 1 2], [1 3 2] )
13( [3 1 2], [2 1 3] )
14( [3 2 1], [1 2 3] )
15( [3 2 1], [3 2 1] )
DONE.
```

```
The time used to put all the good nodes to an array is (Step 3): 0.01

Building the adjacent matrix:
Please wait.....

The adjacent matrix is too large to print out, skip printing......

But we can describe the adjacent matrix:
Describe the graph represented by this matrix:

There is an edge from 0 to 0
There is an edge from 1 to 5
There is an edge from 2 to 7
There is an edge from 3 to 15
There is an edge from 4 to 1
There is an edge from 4 to 10
There is an edge from 4 to 12
There is an edge from 5 to 4
There is an edge from 6 to 2
There is an edge from 6 to 8
There is an edge from 6 to 13
There is an edge from 7 to 6
There is an edge from 8 to 5
There is an edge from 9 to 7
There is an edge from 10 to 15
There is an edge from 11 to 5
There is an edge from 12 to 7
There is an edge from 13 to 15
There is an edge from 14 to 3
There is an edge from 14 to 9
There is an edge from 14 to 11
There is an edge from 15 to 14
DONE.

The time used to get the adjacent matrix is (Step 4): 0.03

The power K is: 4
Now Taking the adjacent matrix to the power K = 4:
Please wait......
DONE.

The time used to take the adjacent matrix to power K is (Step 5): 0.01

Finding the strongly connected components:
Please wait.....

The strongly connected components class are:

The class [0] contains: 0
The class [1] contains: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
DONE.

The time used to find the strongly connected components is (Step 6): 0

WE ARE DONE!!

script done on Sat Oct 28 02:01:39 1995
```

This example output is the result when R = 3, p = 2, q = 0 and r = 6, and each result is named by R, p, q and r (R3p2q0r6 in this case). One can chose any R, p, q and r they wish, but the execution time may be extremely large and it may not be possible to finish the computation because of the limitations discussed in chapter 4. The shift for R = 3 is uncountable, i.e. the shift contains uncountable many bi-infinite paths. Observe that an adjacent matrix is obtained in Step 4 that describes a directed graph. From the output of Step 6, we know that there are two strongly connected components, [0] and [1], where [0] has only node 0 while [1] contains all the remaining good nodes. This shift could be computed by hand since the symmetric group S_3 is relatively small, it turned out that this algorithm obtained the best possible result with the fewest good nodes. Many interesting symmetries are found for the graph of this solution (See Figure 1).

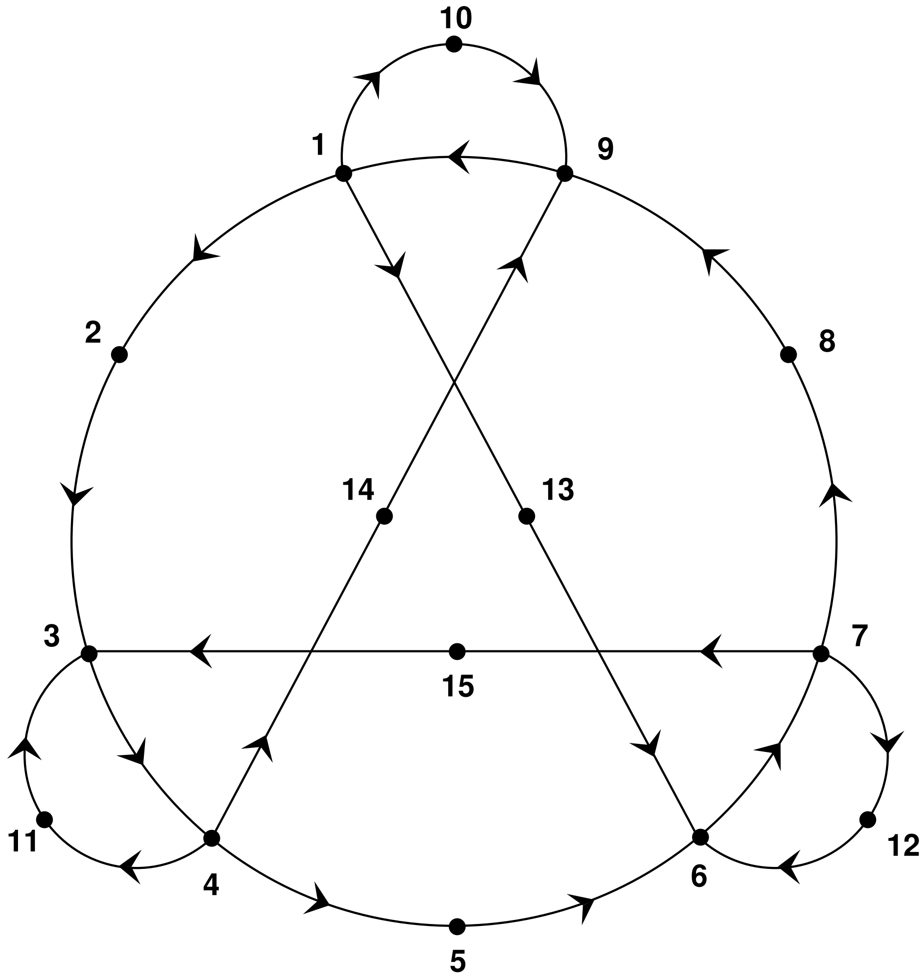


Figure 1
Graph Γ describing ΦS3 (ɡk(5, 1, 13))

A UNIX utility called "script" is used to automatically capture the output in the screen to a log file with start and end times included. Also, the CPU time used at each step was recorded. The total run time can be obtained by using the "time" command of UNIX or by summing the time at each step.

It was not possible to complete many experiments by using the first approach in which a sparse matrix was used while R is equal to 5. However, the performance was improved significantly when a full matrix was used. Many unfinished runs using the first

approach could be completed by using the second approach, although some difficulties were still encountered. Table 1 is a timing table for each step with different R, p, q and r settings:

	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Total Time	Good Nodes
R3p3q0r3	0.05	0.01	0	0	0	0	0.06	9
R4p3q0r3	15.82	1.47	0.03	0.52	0.69	0.02	18.55	105
R5p3q0r3	11182.1	1021.38	0.7	236.6	17757.6	3.92	30202.3	2145
R3p2q1r2	0.06	0.02	0	0.01	0	0.01	0.1	9
R4p2q1r2	14.95	2.35	0.05	0.78	1.71	0.01	19.85	129
R5p2q1r2	10663.5	1221.14	0.83	230.58	17316.7	3.05	29435.8	2129
R3p1q2r3	0.06	0.01	0	0	0	0.01	0.08	1
R4p1q2r3	0.06	0.98	0.01	0.02	0.01	0	1.08	20
R5p1q2r3	10302.2	1230.6	0.54	113.72	4492.76	1.27	16141.09	1522

Table 2
Run Time at Each Step (Unit: Second)

Many more experiments were performed, only 3 different sets are listed here for the purpose of comparison. Note that R3p3q0r3 means R = 3; p = 3, q = 0 and r =3 in the Pretzel knot representation. Each choice of non-negative p, q, and r produces alternating knots. A special investigation into alternating knots was performed because of its importance in knot theory. For a given alternating knot, the performance analysis is given in Table 2 for each step with different R values. In Figure 2 to Figure 10 we compare the time required to finish each step when R=3, 4 and 5 respectively. From the figures, we see that there is not much difference between R = 3 and R = 4 while the time increases dramatically when R = 5. There are 6 permutations in S_3 and 24 permutations in S_4 while there are 120 permutations in S_5. There are 36 and 576 possible good nodes when r = 3 and R =4 while there are 14400 possible good nodes when R = 5. This explains the large increase in execution time from R = 4 to R = 5 at each step (See

Figure 2 - Figure 10). Also, a similar distribution is shown for the number of good nodes (See Figure 9). The growth is exponential as expected. From Figure 9, on the other hand, we see that a large amount of time is allocated at Step 1 and Step 5 because the algorithm loops 120^4 times at Step 1 and it must raise a 2145x2145 matrix to the power 12 at Step 5.

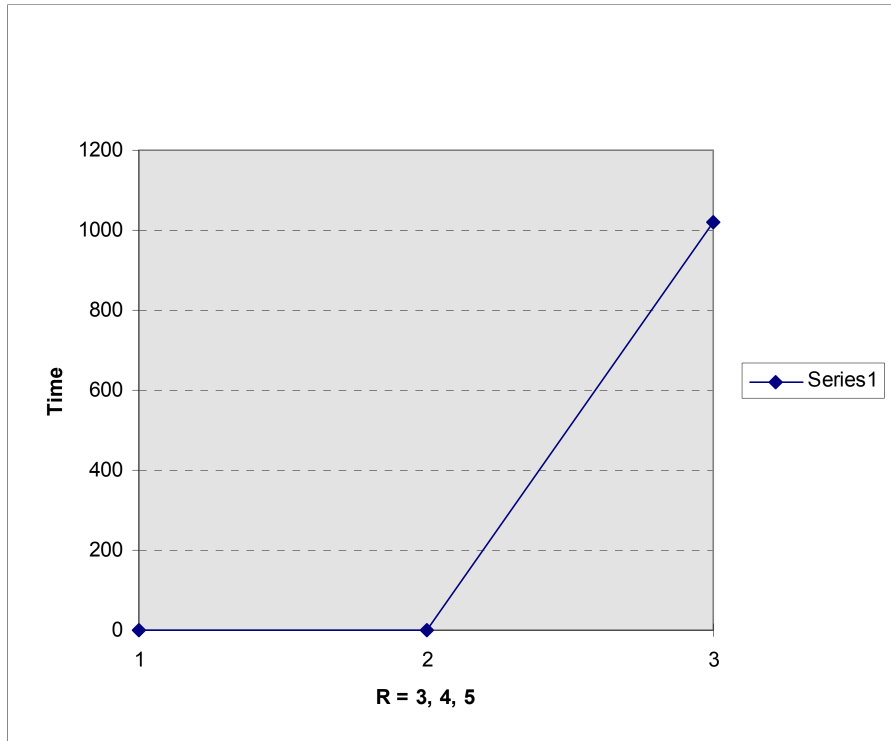


Figure 2
Step 1 – p3q0r3

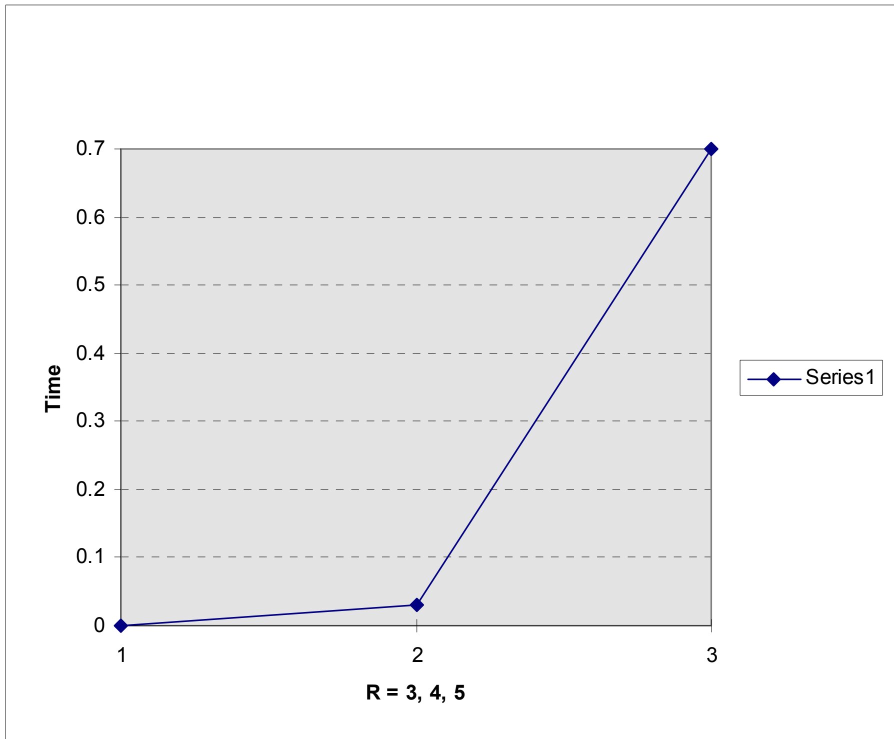


Figure 3
Step 2 – p3q0r3

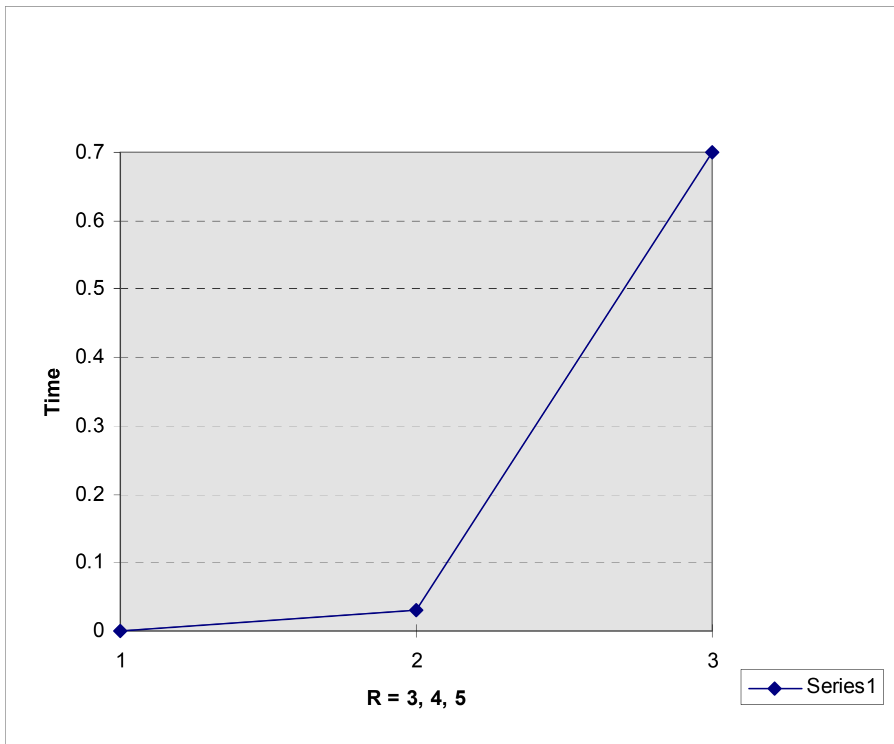


Figure 4
Step 3 – p3q0r3

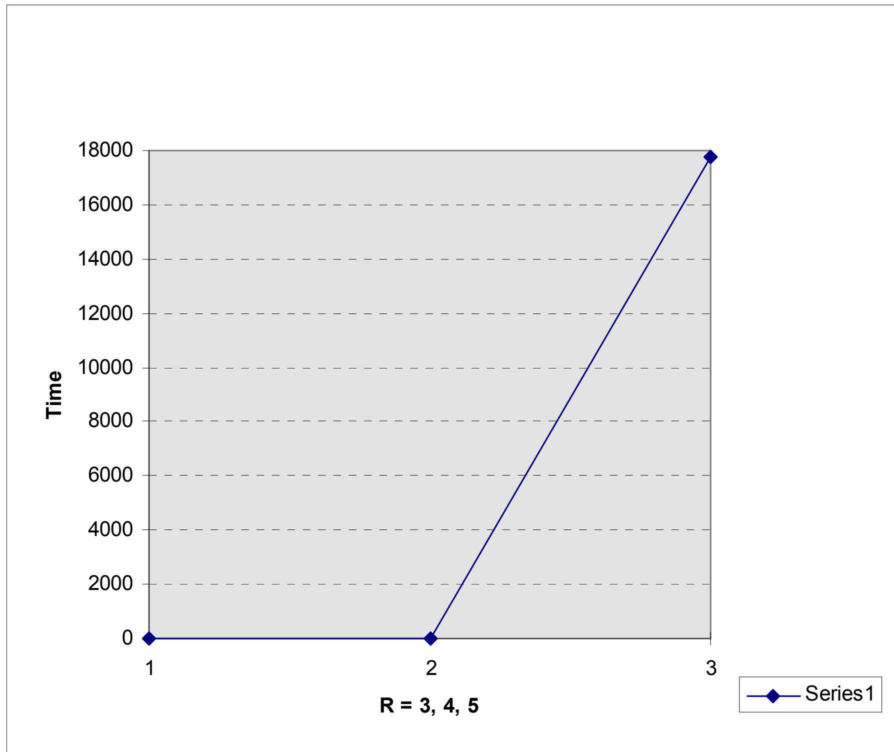


Figure 5
Step 4 – p1q0r3

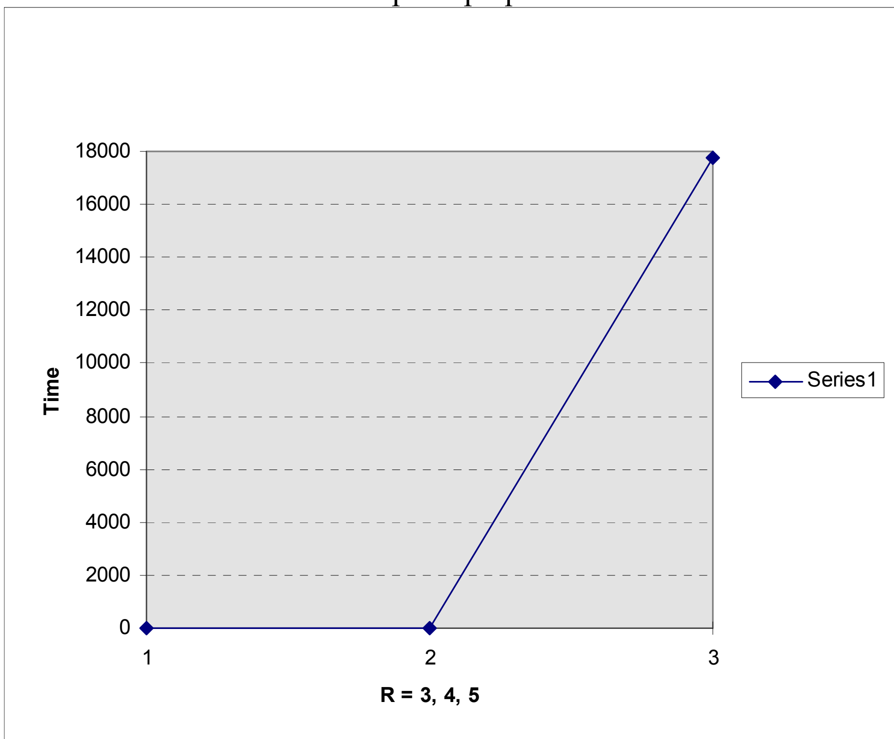


Figure 6
Step 5 – p1q0r3

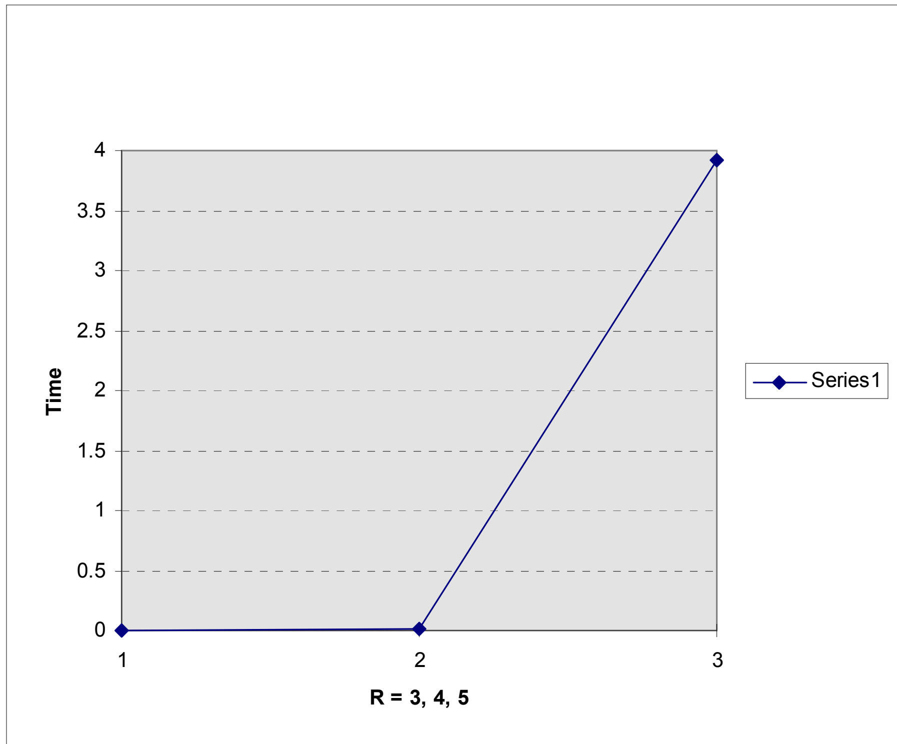


Figure 7
Step 6 – p3q0r3

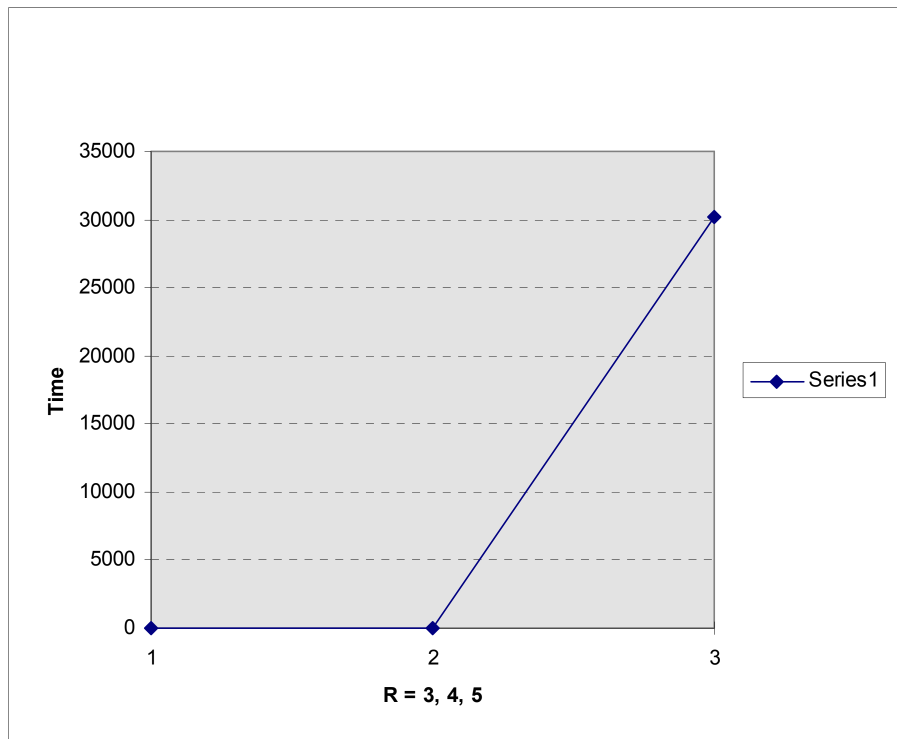


Figure 8
Total Run Time – p3q0r3

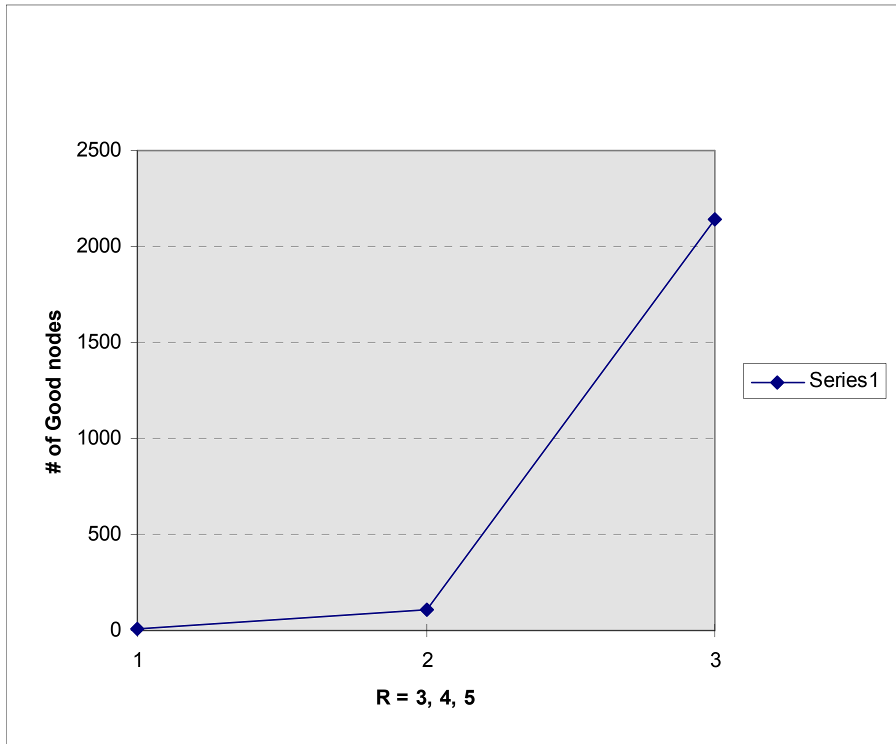


Figure 9
Good Nodes – p3q0r3

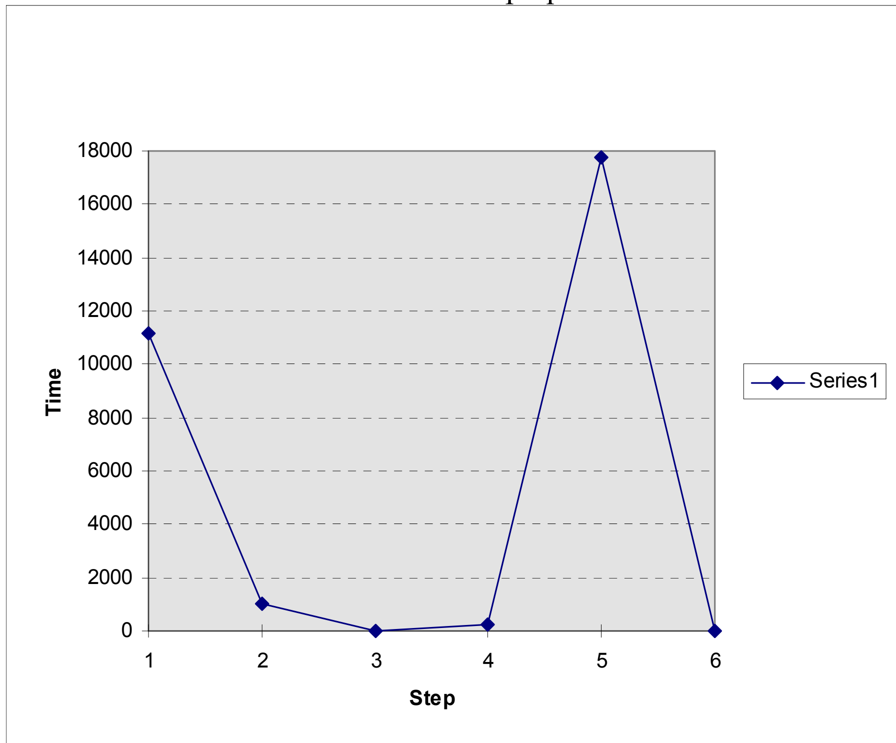


Figure 10
Run Time at Each Step – R5p3q0r3

The performance charts for p2q1r2 and p1q2r3 are listed in APPENDICES A. It is not hard to see that they exhibit a similar behavior to p3q0r3 as discussed above.

As described in Chapter 4, an alternative method could be used to speed up the execution if we are only interested in finding if [0] contains any nodes other than 0. The procedure is as follows:

1. After all good nodes are found in step 3, step 4 is skipped. Instead, a test is performed to see if both $(b_1^{-1} a_1)^{q+1} a_1^p$ and $b_1^{r+1} (b_1^{-1} a_1)^{-q}$ are equal to the identity. If YES, exit. If NO, go to the next step.
2. Test if both $(b_0 a_0^{-1})^{-q} a_0^{p+1}$ and $b_0^r (b_0 a_0^{-1})^{q+1}$ are equal to identity. If YES, exit. If NO, that means [0] contains no any other nodes besides node 0.

Actually, step 1 above attempts to find if node 0 has any out-edges and step 2 checks if node 0 has any in-edges. This approach by passes the use of any matrix method and can only find if [0] contains any nodes other than node 0, it will not tell which nodes [0] contains. The 6-steps method given in chapter 4 will give us the detail information about [0], that is, the program has to be run again if some interesting behavior is observed from this method.

6. CONCLUSIONS AND RECOMMENDATIONS

Three significant results are derived from the work presented in this thesis. The first was to develop a general purpose tool, using C++ classes to implement an algorithm used to study knot invariants. Algorithms that were already developed for this work were considered necessary to push the frontiers of group theory and knot theory in new directions. However, the algorithms were of limited practical value because of their computational complexity. The tool described in this thesis will allow knot invariants for large families of knot to be carefully investigated.

The second contribution of this work was to compare the time and space complexity of several different implementations for known mathematical algorithms. In this work, we were able to make significant improvement to the run time required to solve 'large' problem. Without this analysis, even our computer implementation would have resulted in run time that were impractical for arbitrary, large groups.

The third significant result derived in this work was to provide an example of an alternating knot that could address issues related to an open conjecture in that area. The utility of our toolkit was demonstrated by its ability to be modified to address and find, by trail and error, a possible solution to an open problem. This approach of trying many possible solutions would not have been possible without this tool.

As discussed in chapter 5, the complexity of these types of calculations is naturally high. The program runs very well when the degree of the symmetric group is relatively small, e.g. for R = 3 or 4. There are some difficulties when R =5, it usually takes several days to complete all the calculations. The performance could not be improved significantly due to our limited resources. Fortunately, a small degree of the symmetric group ($R \leq 4$) is often sufficient to satisfy our needs in many cases.

An easy-to-use interface for the pretzel knot family was completed. It is not difficult to built a good interface for any specific family such as the pretzel knots as long as a consistent format of generators and relators is given. Since the code was designed for general use, only a few changes must be made to represent for any given group presentation. In a word, we could use this program to run any given group presentation by modifying a few lines of source code. Further implementation efforts should be made to create a general user interface.

The main difficulty for a general user interface of all knots is that there is no consistent format of presentations while there is an unlimited number of possible presentations. One possible solution is to use a parser or interpreter to convert a presentation to C++ code. The possibility of building a general user interface would be a suitable topic for further research.

REFERENCE LIST

REFERENCE LIST

[BC] B. Baumslag and B. Chandler, Theory and Problems of Group Theory, McGraw-Hill Book Company1968

[BC1] G. Butler, J.J. Cannon, "Computing in Permutation and Matrix Groups I: Normal Closure, Commutator Subgroups, Series," *Mathematics of Computation* **39**, 663-670.

[BC2] G. Butler, J.J. Cannon, "Computing Sylow Subgroups of Permutation Groups Using Homomorphic Image of Centralizers," *J. Symbolic Computation* (1991) **12**, 443-458

[Bu] G. Butler, "Computing in Permutation and Matrix Groups II: Backtrack Algorithm," *Mathematics of Computation* **39**, 671-680.

[BZ] G. Burde, H. Zieschang, Knots, de Gruyte, *Studies in Mathematics* **5**, Berlin, 1985.

[Ca] J.J. Cannon, "A Computational Toolkit for Finite Permutation Groups," *Proceedings of the Rutgers group theory year, 1983-1984*. 1-18.

[Ca2] J.J. Cannon, "Software Tools for Group Theory," *Proc. Sympos. Pure Math*., Vol. **37**, Amer. Math. Soc., Providence, R.I., 1980, 495-502

[Ca3] J.J. Cannon, "An Introduction to the Group Theory Language, Cayley," *Computational Group Theory*, Acdemic Press (1984), 145-183.

[Ha] Thomas F. Hain, "Least-Time and Minimum-Hop Delayed-paths in Clustered Temporal Networks," Ph.D. dissertation, 1992.

[HK] J. C. Hausmann, M. Kercaire, "Sous-groupes derives des groupes de noeuds," *L'Enseignement Math*. **24**, (1978), 111- 123.

[Ka] W.M. Kantor, "Finding Composition Factors of Permutation Groups of Degree $n \leq 10^6$," *J. Symbolic Computation* (1991) **12**, 517-526.

[KSP] K. E. Gorlen, A. M. Orlow and P. S. Plexico, Data Abstraction and Object-Oriented Programming in C++, John Wiley & Sons 1990, Pp 5.

[Le] J. S. Leon, "Permutaton Group Algorithms Based on Partitions, I: Theory & Algorithms," *J. Symbolic Computation* (1991) **12**, 533-583.

[LM] D. Lind, B. Marcus, Introduction to Symbolic Dynamics, Cambridge University Press, Cambridge, 1995.

[LPS] C.R. Leedham-Green, C.E. Praeger, L.H. Soicher, "Computing with Group Homomorphisms," *J. Symbolic Computation* (1991) **12**, 527-532.

[LS] R.C. Lyndon, P.E. Schupp, Combinatorial Group Theory, Springer-Verlag, Berlin 1977.

[Mc] B.D. Mckay, "Nauty User's Guide," Technical Report TR-CS-90-02, Computer Science Department, Australian National University (1978).

[Mc1] B.D. McKay, "Computing Automorphisms and Canonical Labellings of Graphs," *Lecture Notes in Math*. **686**, 223-232.

[Mc2] B.D. McKay, "Pratical graph is isomorphism," *Congressus Numerantium* **30**, 45- 87.

[Ne] P.M. Neumann, "Some algorithms for Computing with Finite Permutation Groups," London Math. Society Lecture Series, 59-92

[Ro] D. Rolfsen, "Knots and Links," Mathematics Lecture Series 7, Publish or Perish, Inc., Berkeley, 1976.

[Se] R. Sedgewick, Algorithms, Addison-wesley Publishing company, 1988, Pp197.

[Si1] C.C. Sims, "Some Group-theoretic Algorithms," Topics in Algebra, *Leture Notes in Math*. **697**, 108-124

[Si2] C.C. Sims, "Computational Methods in the Study of Permutation Groups," Computational Problems in Abstract Algebra. 1970, 169-183.

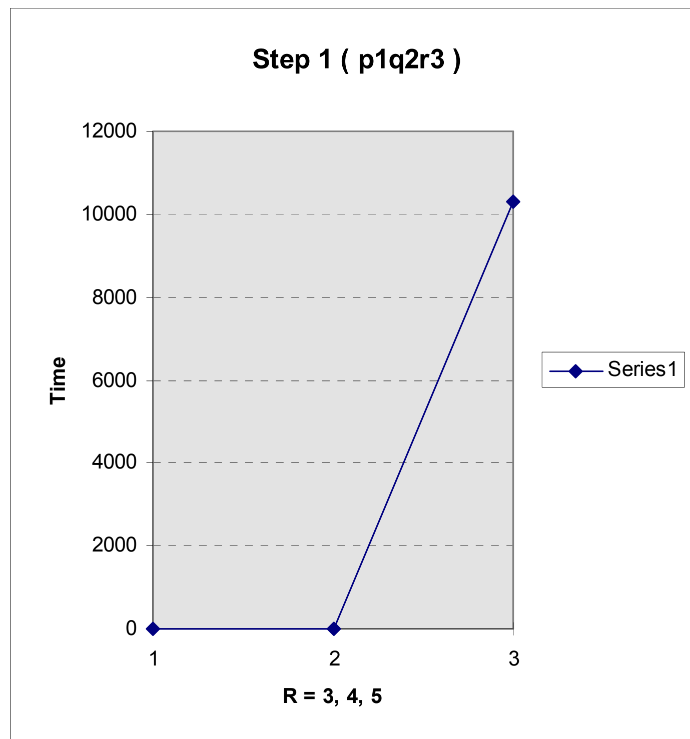
[SW] D. Silver, S. Williams, "Augmented group systems and shifts of finite type," Israel Journal of Mathematics, in press.

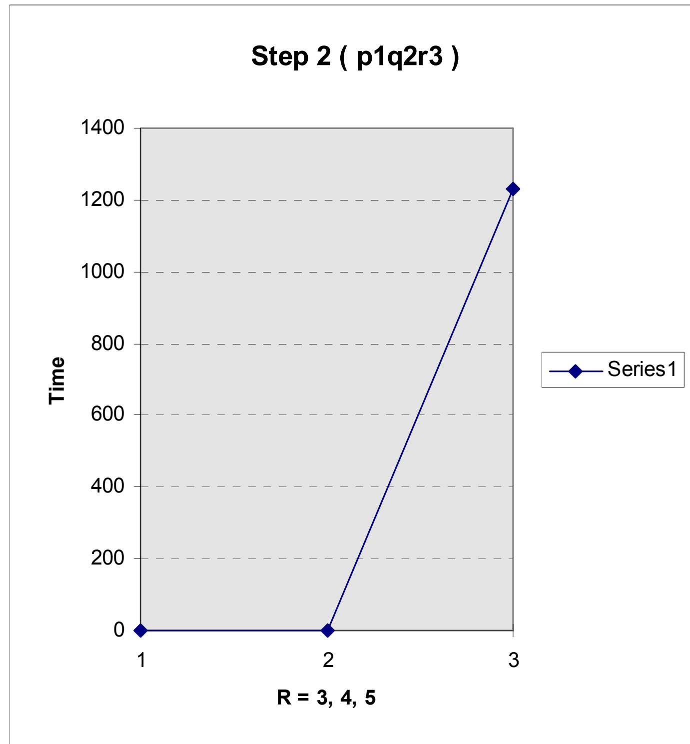
[Ta] R. E. Tarjan, Data Structures and Network Algorithm, Society for industrial & Applied mathematics 1983

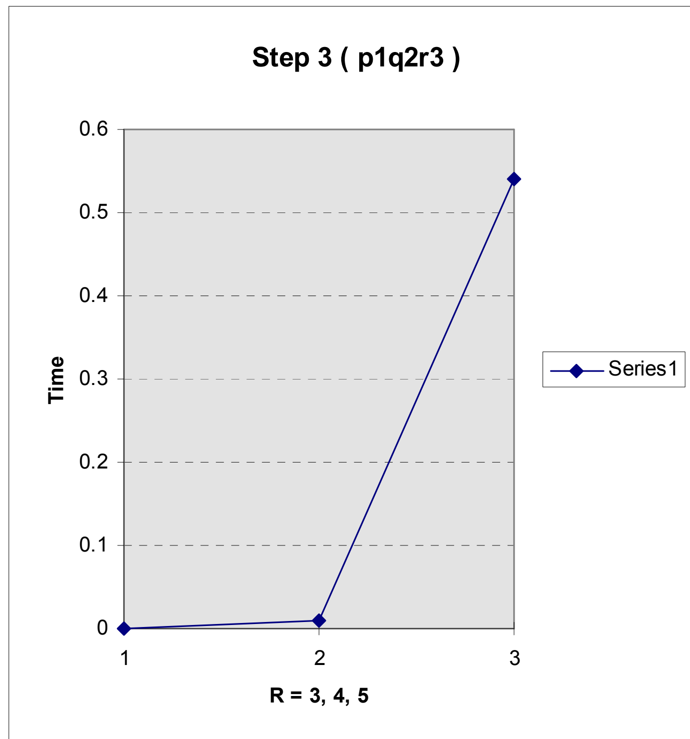
APPENDICES

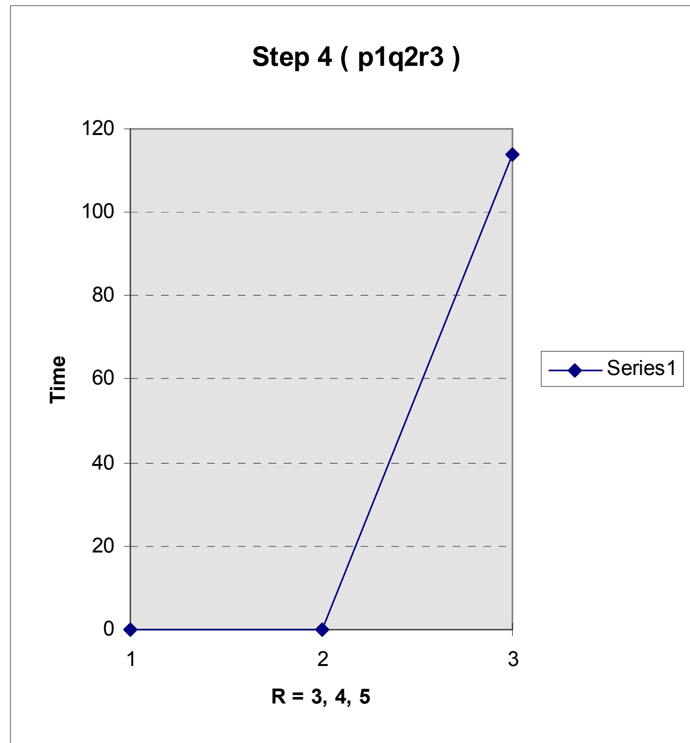
APPENDIX A:
PERFORMANCE CHART FOR p1q2r3
(Unit: second)

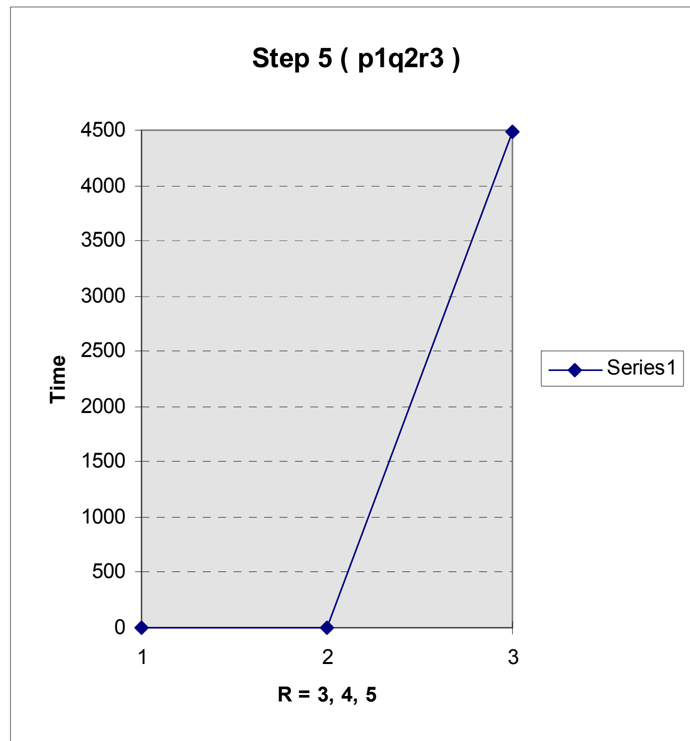
	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Total Time	Good Nodes
R3p1q2r3	0.06	0.01	0	0	0	0.01	0.08	1
R4p1q2r3	0.06	0.98	0.01	0.02	0.01	0	1.08	20
R5p1q2r3	10302.2	1230.6	0.54	113.72	4492.76	1.27	16141.09	1522

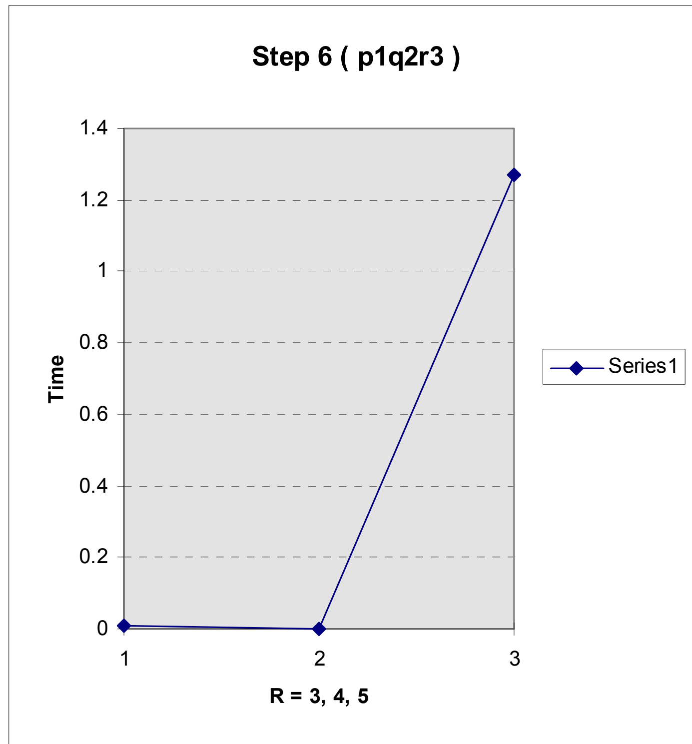


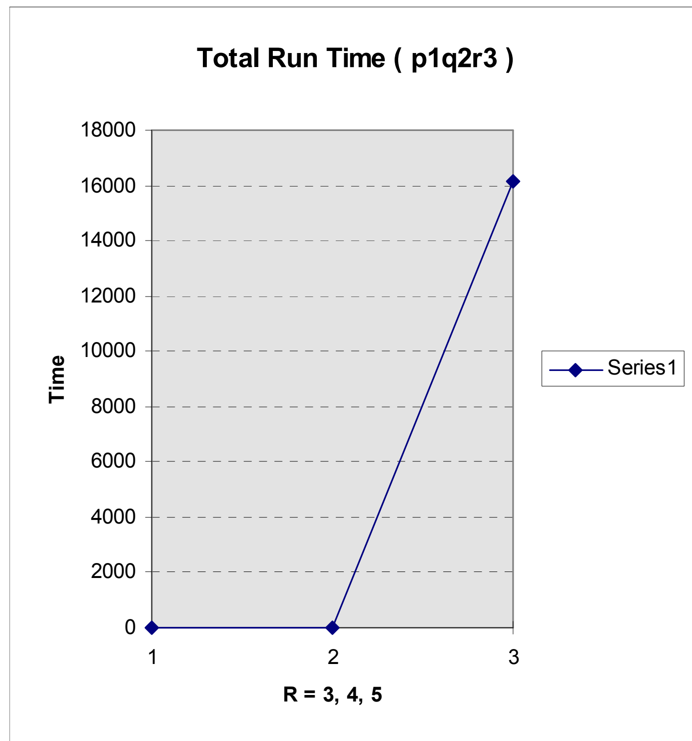

Step 2 (p1q2r3)
1400
1200
1000
800
600
400
200
0
Time
1
2
3
R = 3, 4, 5
Series1

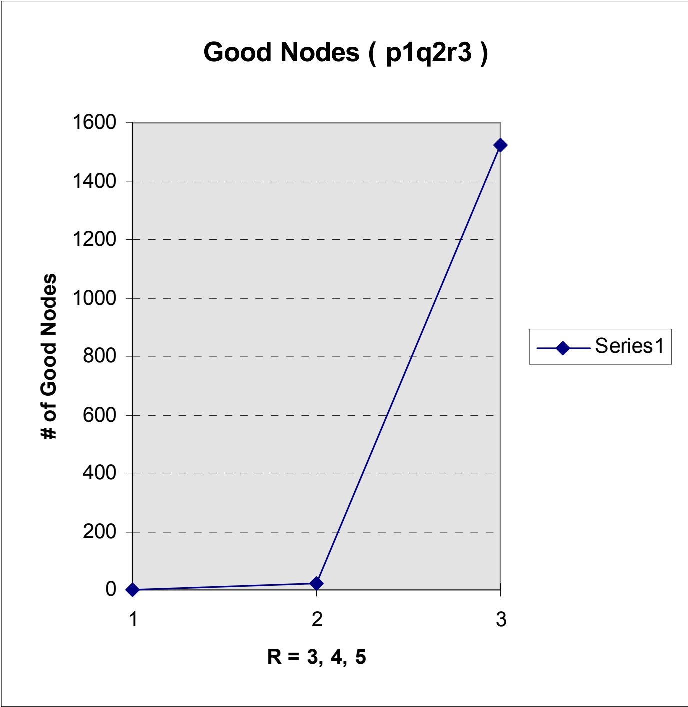

Step 3 (p1q2r3)
0.6
0.5
0.4
0.3
0.2
0.1
0
Time
1
2
3
R = 3, 4, 5
Series1

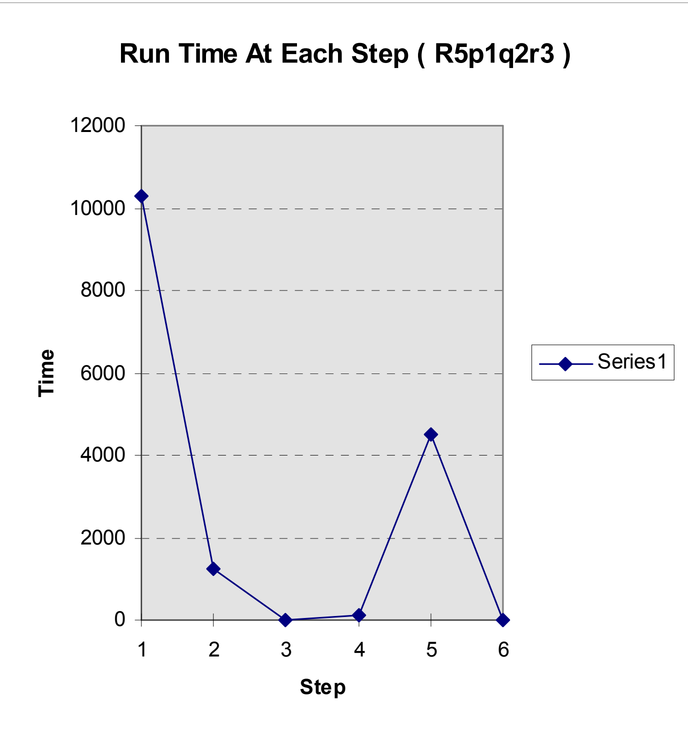

Step 4 (p1q2r3)
120
100
80
60
40
20
0
Time
1
2
3
R = 3, 4, 5
Series1


Step 5 (p1q2r3)
4500
4000
3500
3000
2500
2000
1500
1000
500
0
Time
1
2
3
R = 3, 4, 5
Series1


Step 6 (p1q2r3)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Time
1
2
3
R = 3, 4, 5
Series1

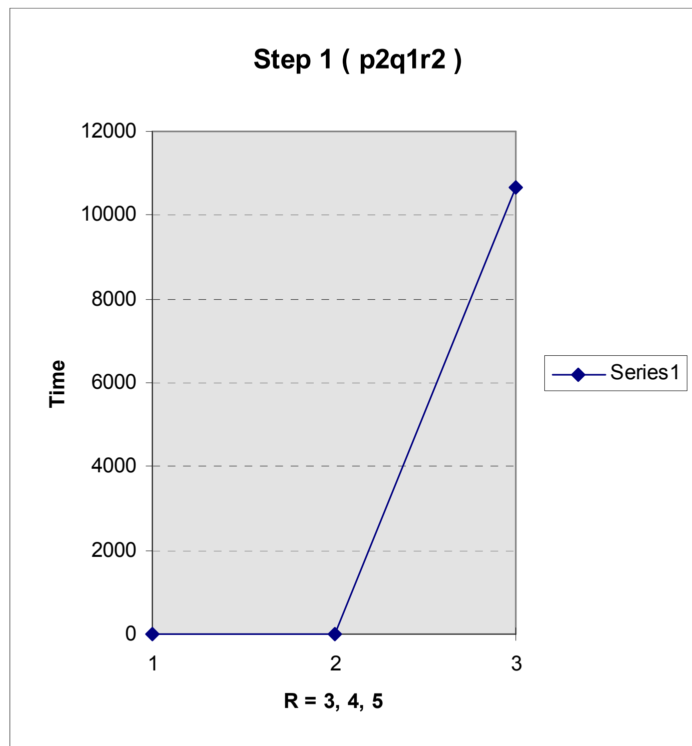

Total Run Time (p1q2r3)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Time
1
2
3
R = 3, 4, 5
Series1

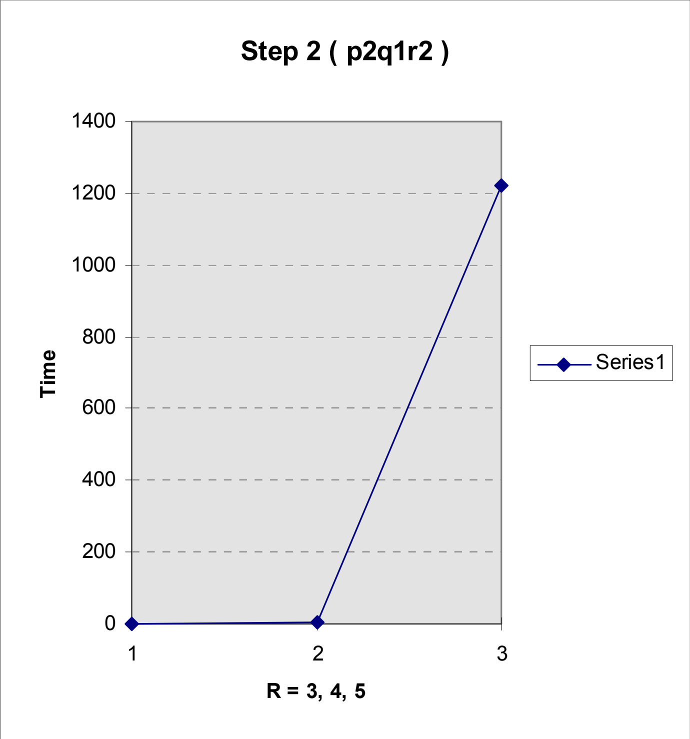

Good Nodes (p1q2r3)
# of Good Nodes
1600
1400
1200
1000
800
600
400
200
0
1
2
3
R = 3, 4, 5
Series1

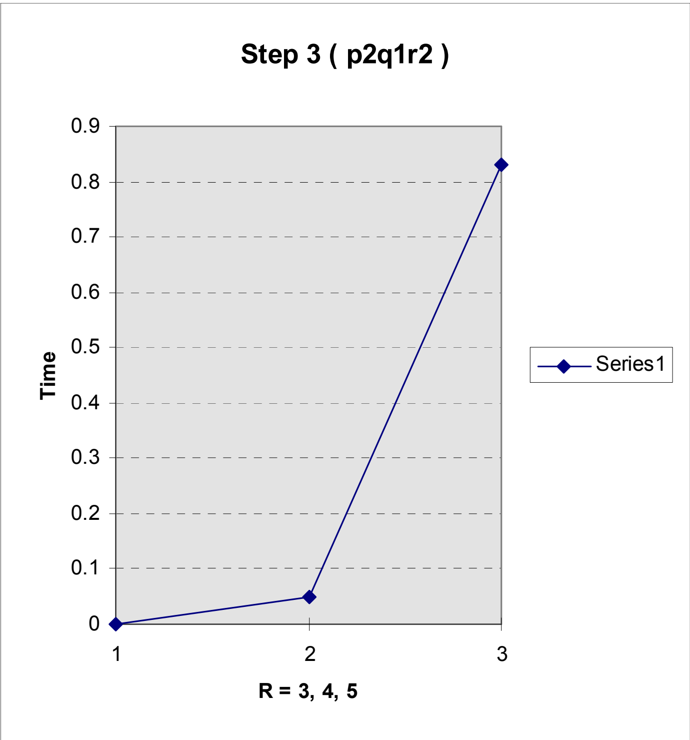

Run Time At Each Step (R5p1q2r3)
Time
12000
10000
8000
6000
4000
2000
0
1
2
3
4
5
6
Step
Series1

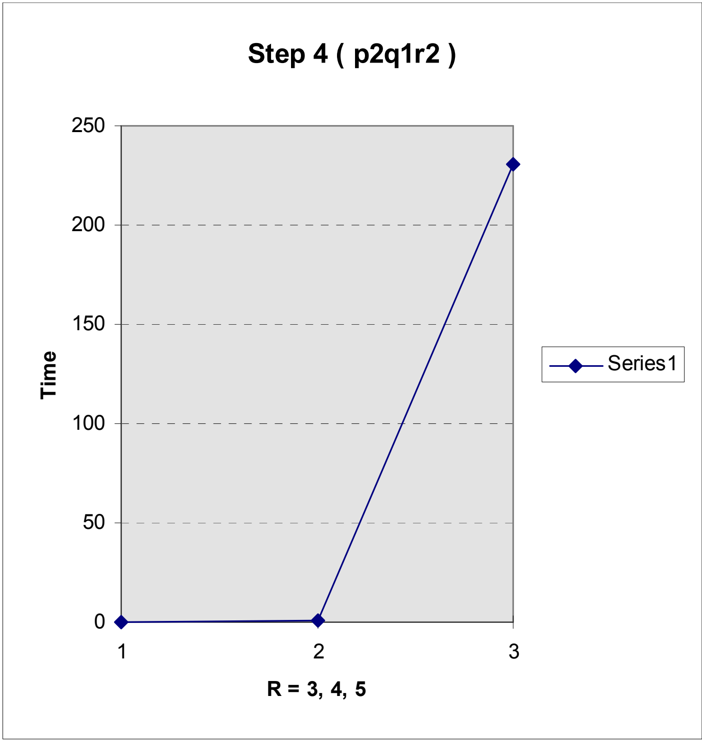
APPENDIX B:
PERFORMANCE CHART FOR p2q1r2
(Unit: second)

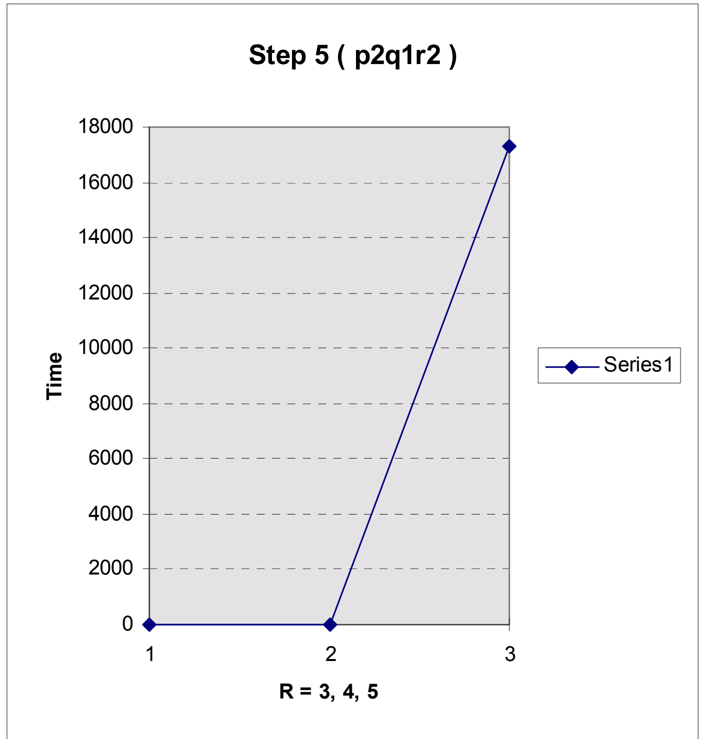
	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Total Time	Good Nodes
R3p2q1r2	0.06	0.02	0	0.01	0	0.01	0.1	9
R4p2q1r2	14.95	2.35	0.05	0.78	1.71	0.01	19.85	129
R5p2q1r2	10663.5	1221.14	0.83	230.58	17316.7	3.05	29435.8	2129

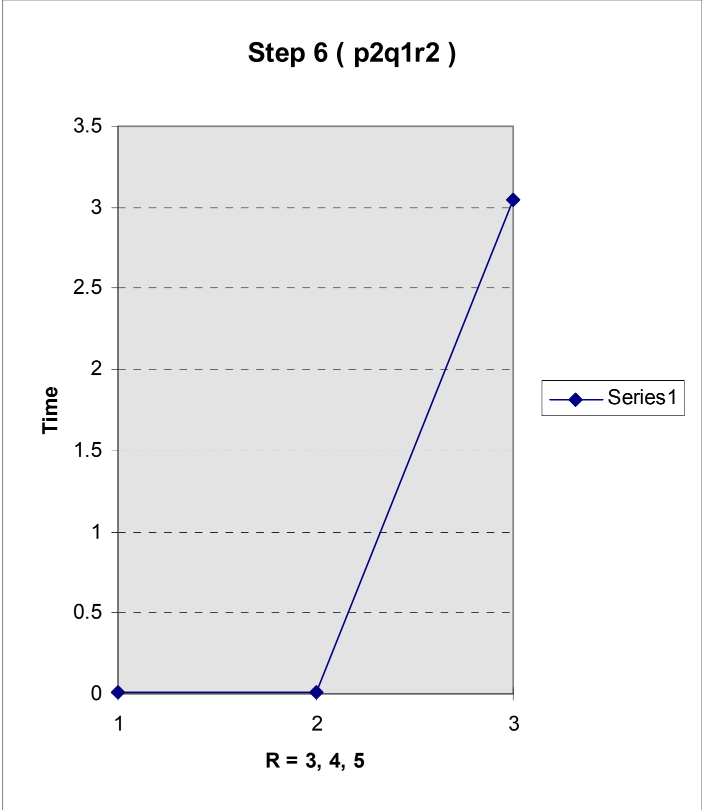



Step 2 (p2q1r2)
1400
1200
1000
800
600
400
200
0
Time
1
2
3
R = 3, 4, 5
Series1

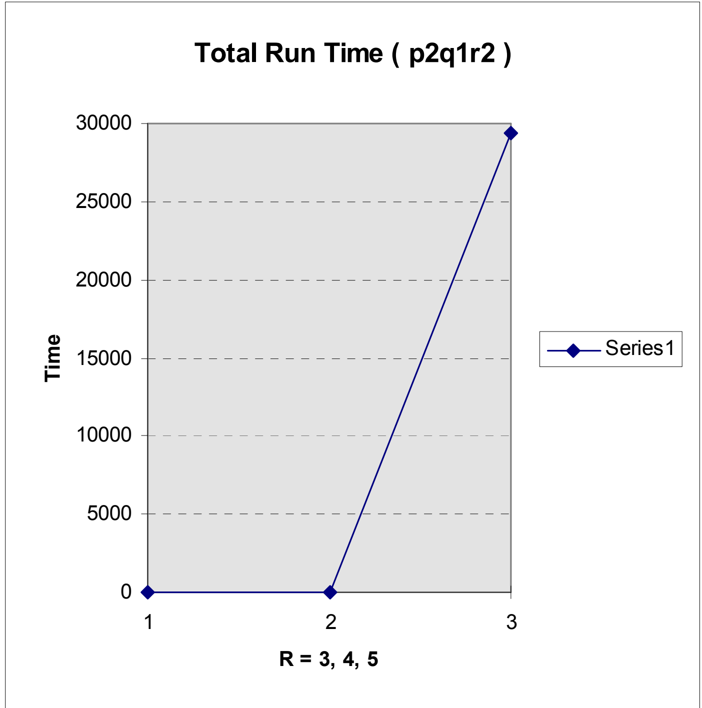

Step 3 (p2q1r2)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Time
1
2
3
R = 3, 4, 5
Series1


Step 4 (p2q1r2)
Time
250
200
150
100
50
0
1
2
3
R = 3, 4, 5
Series1

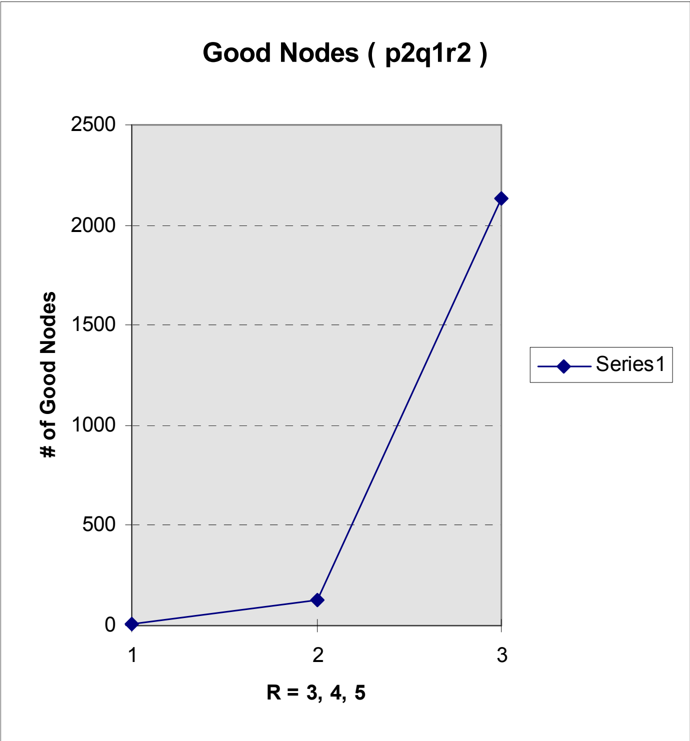

Step 5 (p2q1r2)
Time
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1
2
3
R = 3, 4, 5
Series1

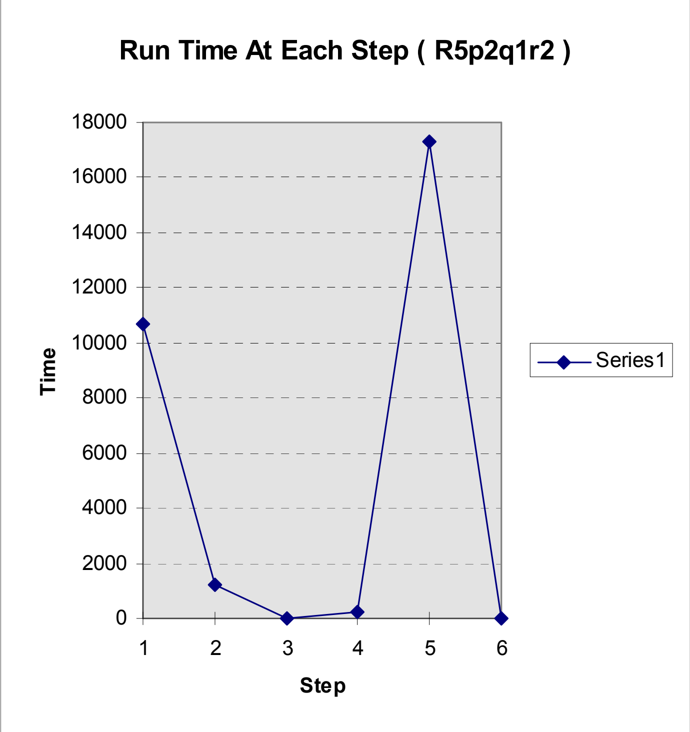


Step 6 (p2q1r2)
3.5
3
2.5
2
1.5
1
0.5
0
Time
1
2
3
R = 3, 4, 5
Series1



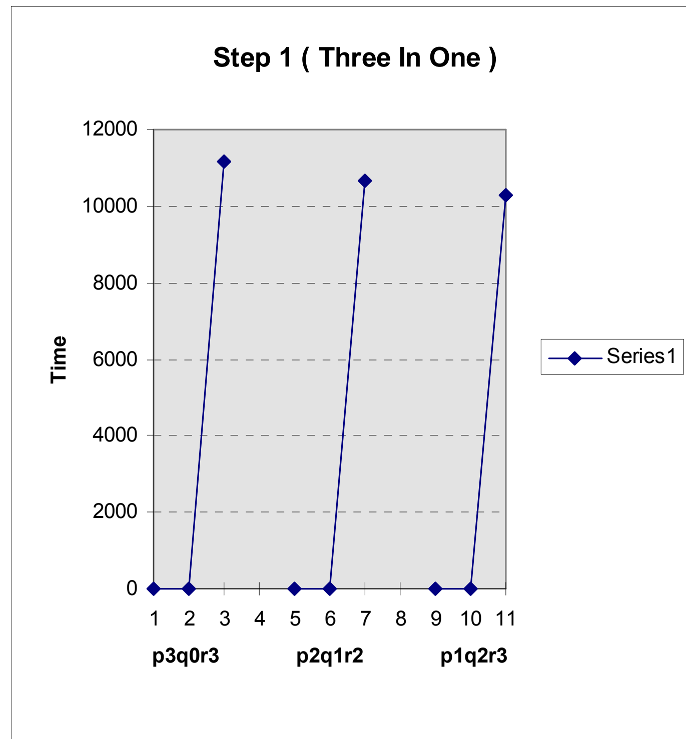
Total Run Time (p2q1r2)
30000
25000
20000
15000
10000
5000
0
Time
1
2
3
R = 3, 4, 5
Series1

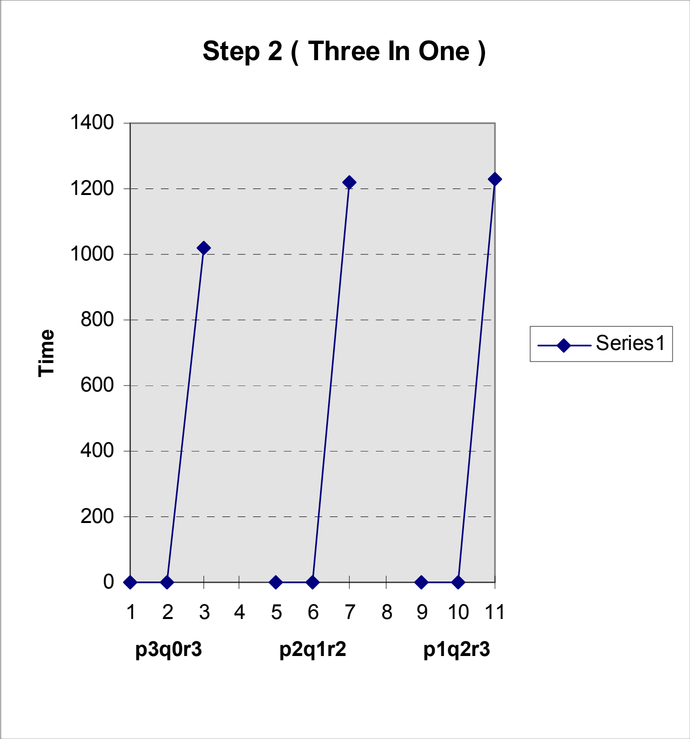

Good Nodes (p2q1r2)
# of Good Nodes
2500
2000
1500
1000
500
0
1
2
3
R = 3, 4, 5
Series1

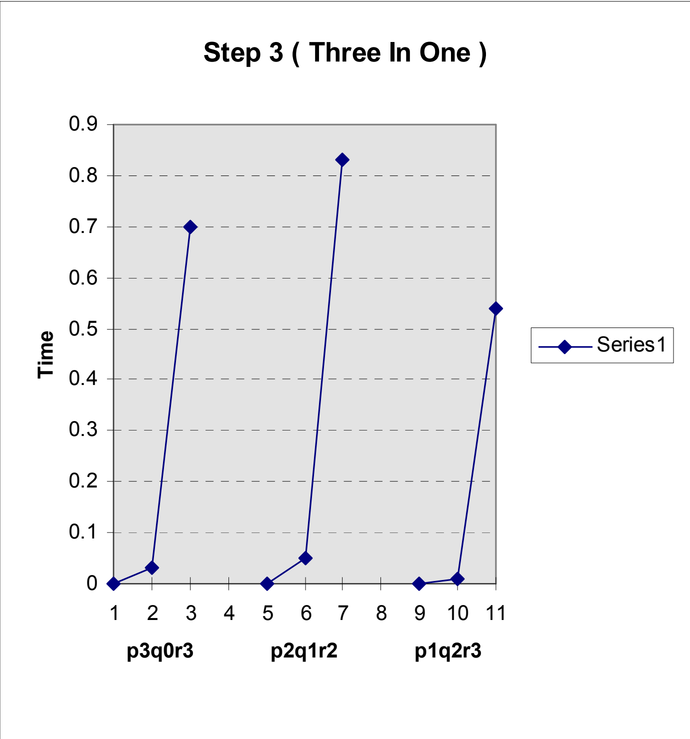

Run Time At Each Step (R5p2q1r2)
Time
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1
2
3
4
5
6
Step
Series1

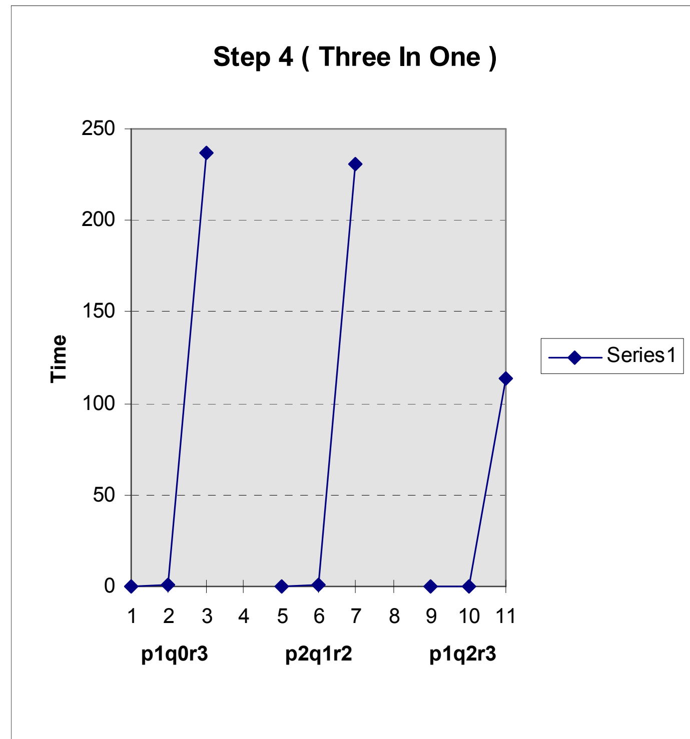
APPENDIX C:
PERFORMANCE CHART FOR ALL 3 SAMPLES
(Unit: second)

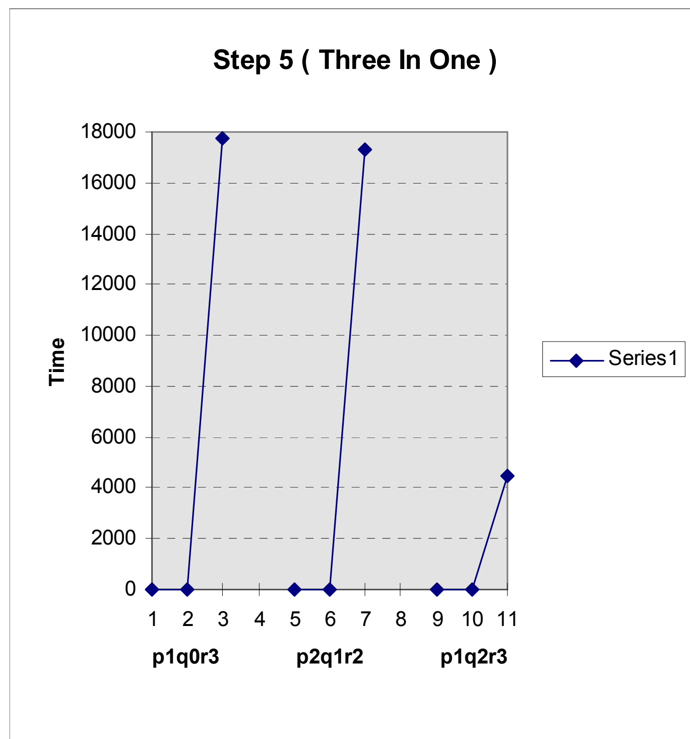
	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Total Time	Good Nodes
R3p3q0r3	0.05	0.01	0	0	0	0	0.06	9
R4p3q0r3	15.82	1.47	0.03	0.52	0.69	0.02	18.55	105
R5p3q0r3	11182.1	1021.38	0.7	236.6	17757.6	3.92	30202.3	2145
R3p2q1r2	0.06	0.02	0	0.01	0	0.01	0.1	9
R4p2q1r2	14.95	2.35	0.05	0.78	1.71	0.01	19.85	129
R5p2q1r2	10663.5	1221.14	0.83	230.58	17316.7	3.05	29435.8	2129
R3p1q2r3	0.06	0.01	0	0	0	0.01	0.08	1
R4p1q2r3	0.06	0.98	0.01	0.02	0.01	0	1.08	20
R5p1q2r3	10302.2	1230.6	0.54	113.72	4492.76	1.27	16141.09	1522

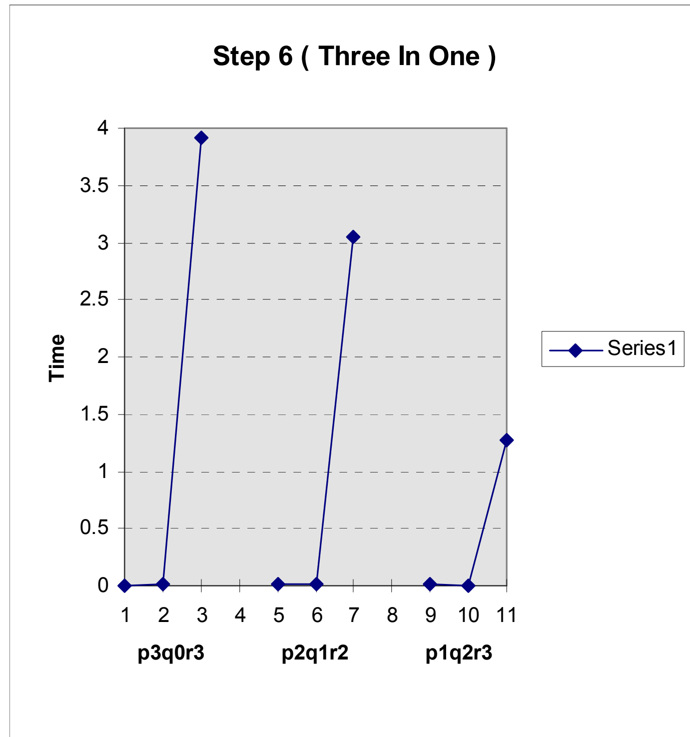



Step 2 (Three In One)
Time
1400
1200
1000
800
600
400
200
0
1 2 3 4 5 6 7 8 9 10 11
p3q0r3
p2q1r2
p1q2r3
Series1

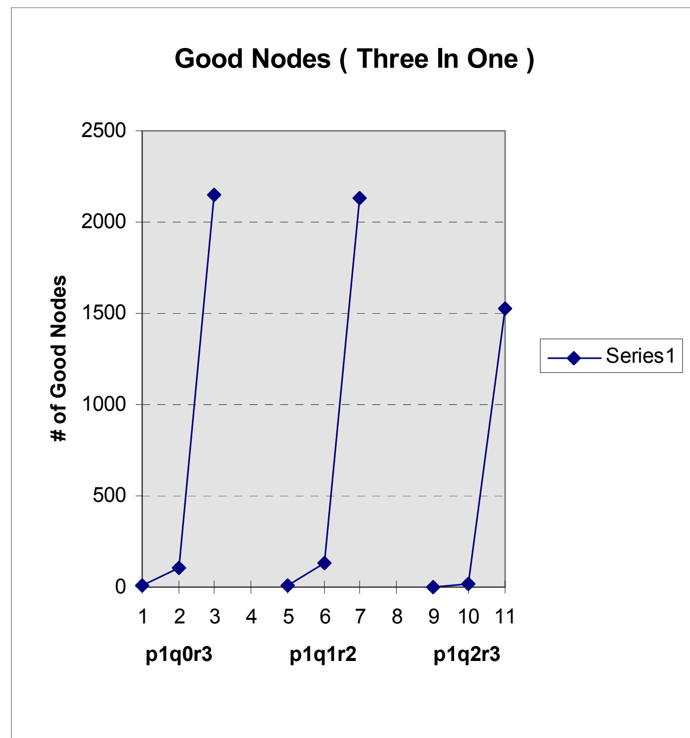

Step 3 (Three In One)
Time
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1 2 3 4 5 6 7 8 9 10 11
p3q0r3
p2q1r2
p1q2r3
Series1


Step 4 (Three In One)
Time
250
200
150
100
50
0
1 2 3 4 5 6 7 8 9 10 11
p1q0r3
p2q1r2
p1q2r3
Series1

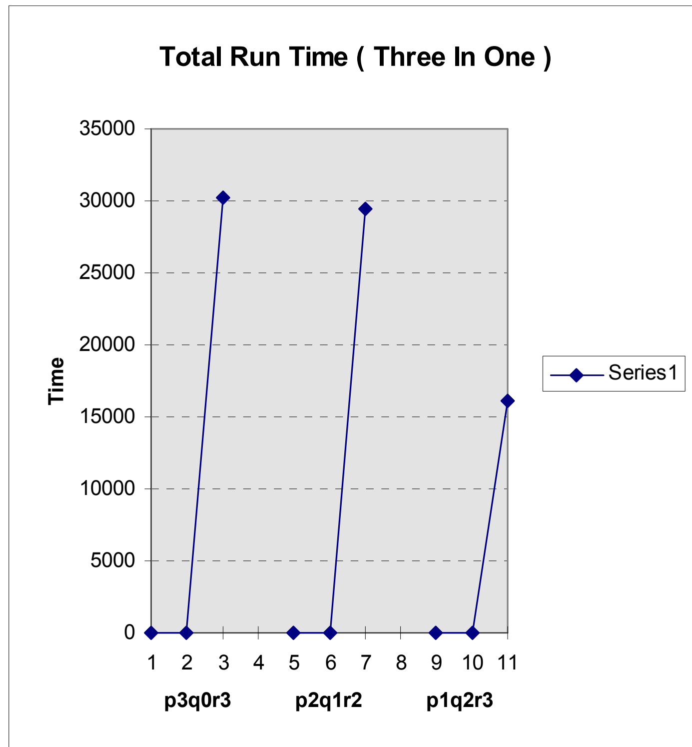

Step 5 (Three In One)
Time
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11
p1q0r3
p2q1r2
p1q2r3
Series1



Step 6 (Three In One)
Time
4
3.5
3
2.5
2
1.5
1
0.5
0
1 2 3 4 5 6 7 8 9 10 11
p3q0r3
p2q1r2
p1q2r3
Series1



Good Nodes (Three In One)
# of Good Nodes
2500
2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10 11
p1q0r3
p1q1r2
p1q2r3
Series1


Total Run Time (Three In One)
35000
30000
25000
20000
15000
10000
5000
0
Time
1 2 3 4 5 6 7 8 9 10 11
p3q0r3
p2q1r2
p1q2r3
Series1

APPDEDIX D:
SOURCE CODE

```
1	// main.cc
2	
3	#include "plist.h"
4	#include "uf.h"
5	#include "timer.h"
6	#include <Integer.h>
7	
8	const int DIM=2;
9	int main()
10	{
11		cout << "Input the necessary parameters you'd like to run:\n";
12		cout << "\n";
13		int R;
14		cout << "Please enter the integer R:\n";
15		cin  >> R;
16	
17		Timer t;
18		SymGrp a0(R), a1(R),b0(R), b1(R);
19		GenSymGrp fnode(R,DIM);
20		GenSymGrp tnode(R,DIM);
21	
22		PermutationList l1(R,DIM);
23		PermutationList l2(R,DIM);
24	
25		int p, q, r;
26	
27		cout << "Please enter the integer p:\n";
28		cin  >> p ;
29	
30		cout << "Please enter the integer q:\n";
31		cin  >> q;
32	
33		cout << "Please enter the integer r:\n";
34		cin  >> r;
35	
36		cout << "\n";
37		cout << "Linking up the potential good nodes to a list:\n";
38		cout << "Please wait.....\n";
39	
40	
41	
42	
43		t.start();
44		const SymGrp IDENTITY(R);
45		while ( a0.more() )		{
```

```
46                a1=IDENTITY;
47                while ( a1.more() )  {
48                  b0 = IDENTITY;
49                  while ( b0.more() ) {
50                   b1=IDENTITY;
51                     while (b1.more() ) {
52                     //  The following code need to be changed for each different knot family.  //
53                     if ( ((b0*(a0.power(-1))).power(-q)
54                               *a0.power(p+1)
55                               *a1.power(-p)
56                               *((b1.power(-1))*a1).power(-q-1)).isIdentity() &&
57                                (b0.power(r)
58                               *(b0*(a0.power(-1))).power(q+1)
59                               *((b1.power(-1))*a1).power(q)
60                               *b1.power(-r-1)).isIdentity() )  {
61                          fnode[0] = a0;
62                          fnode[1] = b0;
63                                    l1.push_end_new(fnode);
64                                    }
65             b1++;
66             }
67                b0++;
68               }
69               a1++;
70               }
71              a0++;
72              }
73
74              cout << "DONE.\n";
75              cout << "\n";
76              cout << "The # of potential good nodes in permutations list is: ";
77              cout << l1.length() << "\n";
78
79              cout << "\n";
80              cout << "The time used to get the permutations list is (Step 1): ";
81              cout << t.lapUtime() << "\n";
82              t.stop();
83              cout << "\n";
84
85              t.start();
86              cout << "Scanning the list and removing the dead-end-nodes:\n";
87              cout << "Please wait.....\n";
88              l1.rmDead1(p, q, r);
89              cout << "DONE.\n";
90              cout << "\n";
91              cout << "There are " <<  l1.length() << " nodes left after the first scan.\n";
92
93              cout << "That was only half-way done, scan the list again:\n";
94              cout << "\n";
95              cout << "Scanning the list again and removing the dead-end-nodes:\n";
96              cout << "Please wait.....\n";
97              l1.rmDead2(p, q, r);
98              cout << "DONE.\n";
99              cout << "\n";
```

```
100                 cout << "There are " << l1.length() << " GOOD NODES left after the second scan.\n";
101                 cout << "The two-way scanning completed.\n";
102                 cout << "\n";
103                 cout << "The time used to remove all the dead-end-nodes is (Step 2):";
104                 cout << t.lapUtime() << "\n";
105                 t.stop();
106                 cout << "\n";
107
108                 t.start();
109                 int N = l1.length();
110                 Data* n_array = new Data[N];
111                 for (register int ii=0; ii < N; ii++)
112                         n_array[ii].setrdim(R, DIM);
113                 l1.fillArray(n_array);
114
115                 cout << "Put all the Good Nodes to an array in increasing order:\n";
116                 cout << "Please wait....\n";
117                 for (register int iii=0; iii<N; iii++) {
118                         cout << iii << n_array[iii] <<"\n";
119                 }
120                 cout << "DONE.\n";
121
122                 cout << "\n";
123                 cout << "The time used to put all the good nodes to an array is (Step 3): ";
124                 cout << t.lapUtime() << "\n";
125                 t.stop();
126                 cout << "\n";
127
128                 t.start();
129                 char **sm = new char*[N];
130                 char **A = new char*[N];
131                 char **tem;
132
133                 for (int i=0; i < N; i++) {
134                         sm[i] = new char[N];
135                         A[i] = new char[N];
136                 }
137
138                 for (i=0; i < N; i++)
139                         for (int j=0; j < N; j++)  {
140                                 sm[i][j] = 0;
141                                 A[i][j] = 0;
142                         }
143
144                 cout << "Building the adjacent matrix:\n";
145                 cout << "Please wait.....\n";
146                 cout << "\n";
147                 cout << "The adjacent matrix is too large to print out, skip printing......\n";
148                 cout << "\n";
149                 cout << "But we can describe the adjacent matrix:\n";
150                 cout << "Describe the graph represented by this matrix:\n";
151                 cout << "\n";
152
153                 for ( i = 0; i< N; i++)
```

```
154                     for (register int j = 0; j< N; j++)
155                             // The following code need to be modified for each different knot family. //
156                             if ( ((((n_array[i][1])
157                               *(n_array[i][0]).power(-1)).power(-q)
158                               *(n_array[i][0]).power(p+1)
159                               *(n_array[j][0]).power(-p)
160                               *((n_array[j][1]).power(-1)
161                               *(n_array[j][0])).power(-q-1)).isIdentity() ) &&
162                               (( (n_array[i][1]).power(r)
163                               *((n_array[i][1])
164                               *(n_array[i][0]).power(-1)).power(q+1)
165                               *((n_array[j][1]).power(-1)
166                               *(n_array[j][0])).power(q)
167                               *(n_array[j][1]).power(-r-1)).isIdentity() ) )  {
168                                 sm[i][j] = 1;
169                                 cout << "There is an edge from " << i << " to " << j << "\n";
170                               }
171
172                 cout << "DONE.\n";
173                 cout << "\n";
174                 cout << "The time used to get the adjacent matrix is (Step 4): ";
175                 cout << t.lapUtime() << "\n";
176                 t.stop();
177                 cout << "\n";
178
179                 delete [] n_array;
180
181                 t.start();
182                 int K = lg(N-1)+1;
183                 cout << "The power K is: " << K << "\n";
184                 cout << "Now Taking the adjacent matrix to the power K = " << K << ":\n";
185                 cout << "Please wait......\n";
186
187                 for ( i = 0; i < N; i++)
188                 sm[i][i] = 1;
189
190                 t.start();
191                 do {
192                         for (register int i = 0; i < N ; i++)
193                                 for (register int j = 0; j < N; j++)
194                                         for ( register int l = 0; l < N; l++)
195                                                 if ( sm[i][l]  && sm[l][j]  ) {
196                                                 A[i][j] = 1;
197                                                 break;
198                                         }
199                           tem = sm;
200                           sm = A;
201                           A = tem;
202                 } while ( --K );
203
204                 cout << "DONE.\n";
205                 cout << "\n";
206                 cout << "The time used to take the adjacent matrix to power K is (Step 5): ";
207                 cout << t.lapUtime() << "\n";
```

```
208             t.stop();
209             cout << "\n";
210
211
212             for ( i = 0; i < N; i++)
213             A[i][i] = 0;
214
215
216             t.start();
217             DisjointSet ds(N);
218             cout << "Finding the strongly connected components:\n";
219             cout << "Please wait.....\n";
220
221             for ( register int m = 0; m < N; m++)
222                 for ( register int n= 0; n < N; n++)
223                         if ( A[m][n]  && A[n][m] )
224                            ds.link(n,m);
225
226             cout << "\n";
227             cout << "The strongly connected components class are:\n";
228             ds.showSets(N);
229
230             cout << "DONE.\n";
231             cout << "\n";
232             cout << "The time used to find the strongly connected components is (Step 6): ";
233             cout << t.lapUtime() << "\n";
234             t.stop();
235             cout << "\n";
236             cout << "WE ARE DONE!!\n";
237             return 0;
238     }
```

```
1       // plist.cc
2
3       #include "plist.h"
4
5       /////////////////////////////////////////////////////////////////////////////////////////////////////////////
6       void PermutationList::copyList(Node *s)
7       {
8               for (Node *q = s->next, *t = first; q != NULL ; q = q->next, t = t->next)
9                       t->next = new Node(q->p);
10              t->next = NULL;
```

```
11      }
12
13      ///////////////////////////  Constructor  /////////////////////////////////////////////////////////////
14      PermutationList::PermutationList(const PermutationList &l): r(l.r),dim(l.dim)
15      {
16              first = new Node(r,dim);
17              first->next = first;
18              tail->next = tail;
19              //curr->next = curr;
20              copyList(l.first);
21      }
22
23      ////////////////////////////////////////////////////////////////////////////////////////////////////
24      const PermutationList& PermutationList::operator = ( const PermutationList& l)
25      {
26              if (r != l.r)
27                      perror("Can't copy PermutationList of different r value");
28              if (this == &l)
29                      return *this;
30              rmAll();
31              copyList(l.first);
32              return *this;
33      }
34
35      ////////////////////////////////////////////////////////////////////////////////////////////////////
36      int PermutationList::length()
37      {
38                      int cnt = 0;
39                      for ( Node* np = this->first; np != NULL; np = np->next, cnt++);
40                      return cnt;
41      }
42
43      ////////////////////////////////////////////////////////////////////////////////////////////////////
44      Data* PermutationList::fillArray(Data* node_array)
45      {
46                      int i;
47                      Node* np;
48                      for (np = this->first, i=0; np != NULL; np = np->next, i++)
49                              node_array[i] = np->p;
50                      return node_array;
51      }
52
53      ////////////////////////////////////////////////////////////////////////////////////////////////////
54      void PermutationList:: rmAll()
55      {
56                      for (Node *pt = first->next; pt != NULL; ) {
57                              Node *s = pt;
58                              pt = pt->next;
59                              delete s;
60                      }
61                      first->next = NULL;
62      }
63
64      ////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
65      void PermutationList::clear()
66      {
67              this->reset();
68              this->free_nodes(this->first);
69              this->first = NULL;
70      }
71
72      ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
73      void PermutationList::free_nodes(Node* np)
74      {
75              do {
76                 Node* next_np = np->next;
77                 np->next = NULL;
78                 delete np;
79                 np = next_np;
80              }   while (np != NULL );
81      }
82
83      ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
84      Bool PermutationList::do_find(Node* np, const Data& x, Node*& cp, Node*& pp) const
85      {
86              Node* prev_np = NULL;
87              for ( ; np != NULL; prev_np = np, np = np->next_node())
88                    if (np->p == x) {
89                          cp = np;
90                          pp = prev_np;
91                          return TRUE;
92                          }
93          cp = NULL;
94          pp = NULL;
95          return FALSE;
96      }
97
98      ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
99
100     void PermutationList::output_data(ostream& os, const Node* np) const
101     {
102             os << ((const Node*)np)->p;
103     }
104
105     ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
106     ostream& operator<<(ostream& os, const PermutationList& l)
107     {
108             os << "(";
109             Node* np = l.first;
110             if ( np != NULL) {
111             l.output_data(os, np);
112                     for ( np = np->next; np != NULL; np = np->next)
113                     {
114                             os << " -> ";
115                             l.output_data(os, np);
116                     }
117             }
118             return os << ")";
```

```
119	}
120
121
122	////////////////////////////////////////////////////////////////////////////////////////////////////////////////
123	void PermutationList::getIntersection(const PermutationList& l)
124	{
125		this->reset();
126		if (l.first == NULL)
127		   this->clear();
128	      else  {
129		   Node* np = this->first;
130		   Node* prev_np = NULL;
131		   Node* next_np;
132		   while ( np != NULL)  {
133		     Node* cp;
134		     Node* pp;
135		     if ( !l.do_find(l.first, np->p, cp, pp))  {
136		       next_np = np->next;
137		       if ( prev_np == NULL)
138				this->first = np->next;
139		       else
140				prev_np->next = np->next;
141				this->free_nodes(np);
142				np = next_np;
143		     }
144		     else {
145				prev_np = np;
146				np = np->next;
147		     }
148		  }
149	}
150	}
151
152	////////////////////////////////////////////////////////////////////////////////////////////////////////////////
153	void PermutationList::rmDead1(int x, int y, int z)
154	{
155		for ( Node* pnp = first; pnp->next != NULL; ) {
156		   Bool found = FALSE;
157		     for ( Node* np = first->next; np != NULL; np = np->next)
158			// The following code need to be modified for each different knot family. //
159			if ( ( (((pnp->next->p)[1]*((pnp->next->p)[0].power(-1))).power(-y)
160				*(pnp->next->p)[0].power(x+1)
161				*(np->p)[0].power(-x)
162				*(((np->p)[1].power(-1))*(np->p)[0]).power(-y-1)).isIdentity()) &&
163				( (((pnp->next->p)[1]).power(z)
164				*((pnp->next->p)[1]*((pnp->next->p)[0].power(-1))).power(y+1)
165				*(((np->p)[1].power(-1))*(np->p)[0]).power(y)
166				*(np->p)[1].power(-z-1)).isIdentity() ))  {
167				     found = TRUE;
168				     break;
169				}
170			if (!found) {
171				Node* tem = pnp->next;
172				pnp->next = pnp->next->next;
```

```
173                         delete tem;
174                     }
175                     else    pnp = pnp->next;
176             }
177     }
178
179     ////////////////////////////////////////////////////////////////////////////////////////////////////////
180     void PermutationList::rmDead2(int x, int y, int z)
181     {
182             for ( Node* pnp = first; pnp->next != NULL; ) {
183                     Bool found = FALSE;
184                     for ( Node* np = first->next; np != NULL; np = np->next)
185                    // The following code need to be modified for each different knot family.//
186                 if  ( (( ( (np->p)[1]*((np->p)[0].power(-1))).power(-y)
187                             *(np->p)[0].power(x+1)
188                             *(pnp->next->p)[0].power(-x)
189                             *(((pnp->next->p)[1].power(-1))
190                             *(pnp->next->p)[0]).power(-y-1)).isIdentity()) &&
191                             ( (   ((np->p)[1]).power(z)
192                             *( (np->p)[1]*((np->p)[0].power(-1))).power(y+1)
193                             *(((pnp->next->p)[1].power(-1))
194                             *(pnp->next->p)[0]).power(y)
195                             *(pnp->next->p)[1].power(-z-1)).isIdentity() )          )  {
196                                 found = TRUE;
197                                 break;
198                             }
199                     if (!found) {
200                             Node* tem = pnp->next;
201                             pnp->next = pnp->next->next;
202                             delete tem;
203                     }
204                     else
205                             pnp = pnp->next;
206             }
207     }
208
209     ////////////////////////////////////////////////////////////////////////////////////////////////////////
210     Node* PermutationList::insert_after_node(const Data& value, Node* prev_np) const
211     {
212       Node* anode = new Node(value);
213       if (prev_np != NULL) {
214          anode->next_node() = prev_np->next_node();
215          prev_np->next_node() = anode;
216        }
217       return anode;
218     }
219
220     ////////////////////////////////////////////////////////////////////////////////////////////////////////
221     Bool PermutationList::push_end(const Data& x) {
222       Node* last_np = this->first;
223       if (last_np == NULL) {
224         this->first = this->curr = this->insert_after_node(x, NULL);
225         this->prev = NULL;
226        }
```

```
227       else {
228         Node* cp = this->curr;
229         if (cp != NULL && cp->next == NULL)
230           last_np = cp;
231         else
232           while(last_np->next != NULL) last_np = last_np->next;
233         this->curr = this->insert_after_node(x, last_np);
234         this->prev = last_np;
235       }
236      return TRUE;
237     }
238
239     ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
240     void PermutationList::push_end_new(const Data& x) {
241       if (!this->do_find(this->first, x, this->curr, this->prev))
242          this->push_end(x);
243       else
244         ;
245     }
246     ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

VITA

VITA

Dong-Biao Zheng was born on October 14, 1968 in Fujian, P. R. of China, son of Wanman Xu and Xuehai Zheng. He graduated from Hua Qiao University, Fujian, P.R. of China with a B.S. in Mathematics in 1990. A graduate assistantship was awarded to Dong-Biao from Department of Mathematics and Statistics of Wichita State University, Kansas at Fall, 1991. He transferred to University of South Alabama with a full financial support from Department of Mathematics and Statistics at Fall, 1992, and obtained a M.S. in mathematics in August, 1994. Surfing the Internet led him to decide to pursue another M.S. in Computer Science. He is still spending lots of time in browsing Internet, way too much.

7.