Summer 1998

Improving the run time of the decomposition algorithm for fault tolerant Clos interconnection networks through swap re-ordering

Andrea Laura McMakin
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

IMPROVING THE RUN TIME OF THE DECOMPOSITION ALGORITHM FOR FAULT TOLERANT CLOS INTERCONNECTION NETWORKS THROUGH SWAP RE-ORDERING

by
Andrea Laura McMakin

Clos interconnection networks, used in data networks and computing systems, can contain extra switches to be used in faulty conditions. The speed of such fault tolerant Clos interconnection networks is improved through the use these switches in no-fault situations. The network can be represented by a matrix, which is then decomposed using an algorithm, and the switch settings are thus assigned.

The original decomposition algorithm consisted of four element swaps in the following order: wild swap, simple swap, next simple swap, and successive swap. However, by re-arranging these swaps with the simple swap first, followed by the next simple and successive swaps with the wild swap coming either before or after the next simple, the number of total swaps needed to fully decompose the matrix is significantly reduced.

IMPROVING THE RUN TIME OF THE DECOMPOSITION ALGORITHM FOR FAULT TOLERANT CLOS INTERCONNECTION NETWORKS THROUGH SWAP RE-ORDERING

by
Andrea Laura McMakin

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

August 1998

APPROVAL PAGE

IMPROVING THE RUN TIME OF THE DECOMPOSITION ALGORITHM FOR FAULT TOLERANT CLOS INTERCONNECTION NETWORKS THROUGH SWAP RE-ORDERING

Andrea Laura McMakin

Dr. John D. Carpinelli, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. MengChu Zhou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Andrea Laura McMakin

Degree: Master of Science in Computer Engineering

Date: August 1998

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1998

- Bachelor of Science in Mathematics,
  University of Notre Dame, Notre Dame, IN, 1996

Major: Computer Engineering

This thesis is dedicated to
Irene Morrah and Edna Kate Ingold
Thanks for making the past 8 months interesting!

ACKNOWLEDGMENT

The author wishes to express her gratitude to her advisor, Dr. John Carpinelli, for his guidance and topic suggestion and to Dr. Edwin Hou and Dr. MengChu Zhou for serving on the defense committee.

Special thanks to the author's parents, for obvious reasons.

The author must also recognize Lex Spoon for listening to countless rants and for reading seemingly endless e-mails on the subject of swaps. Also, without Lex's superior knowledge of C, the author could never have made it through CSE 232, Introduction to Programming.

TABLE OF CONTENTS

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF TABLES
(Continued)

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Clos Interconnection Networks

1.1.1 Importance of Clos Networks

Clos interconnection networks are important because of their potential uses in data networks and computing systems (Lee, Hwang, and Carpinelli 1996, 1572). These networks have also been receiving attention because of their simple structures, rearrangeability, and nonblocking properties (Wang 1997, 1). The concept of the Clos network was first developed by C. Clos and reported in the Bell Systems Technical Journal (1953).

1.1.2 Basic Design

A general ordinary Clos network is shown in Figure 1. This three stage network has two symmetrical outer stages of rectangular switches and an inner stage of square switches. The first stage contains r switches, each with n inputs and m outputs. Each switch is a simple crossbar switch that can realize any mapping of its inputs onto its outputs on a one-to-one basis. The second stage contains m switches, each with r inputs and r outputs. Each switch receives exactly one input from each first stage switch and sends exactly one output to each third stage switch. The third, or output, stage contains r switches, each with m inputs and n outputs. Each output stage switch receives exactly one input from each second stage switch. Since there are r first stage switches, each with n inputs, the total number of inputs to this network is $N = nr$ (Lee, Hwang, and Carpinelli 1996, 1572).

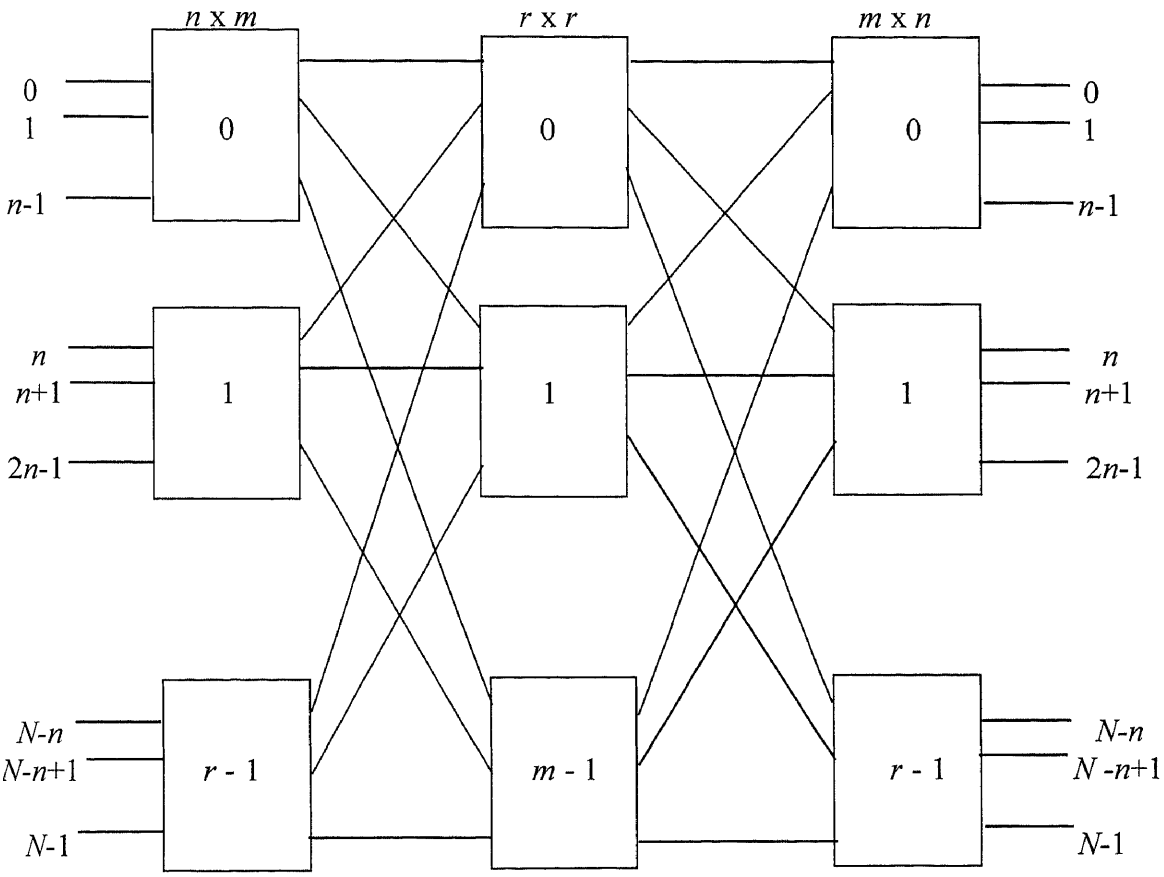

Figure 1 A Three Stage General Ordinary Clos Network
(Wang 1997, 1)

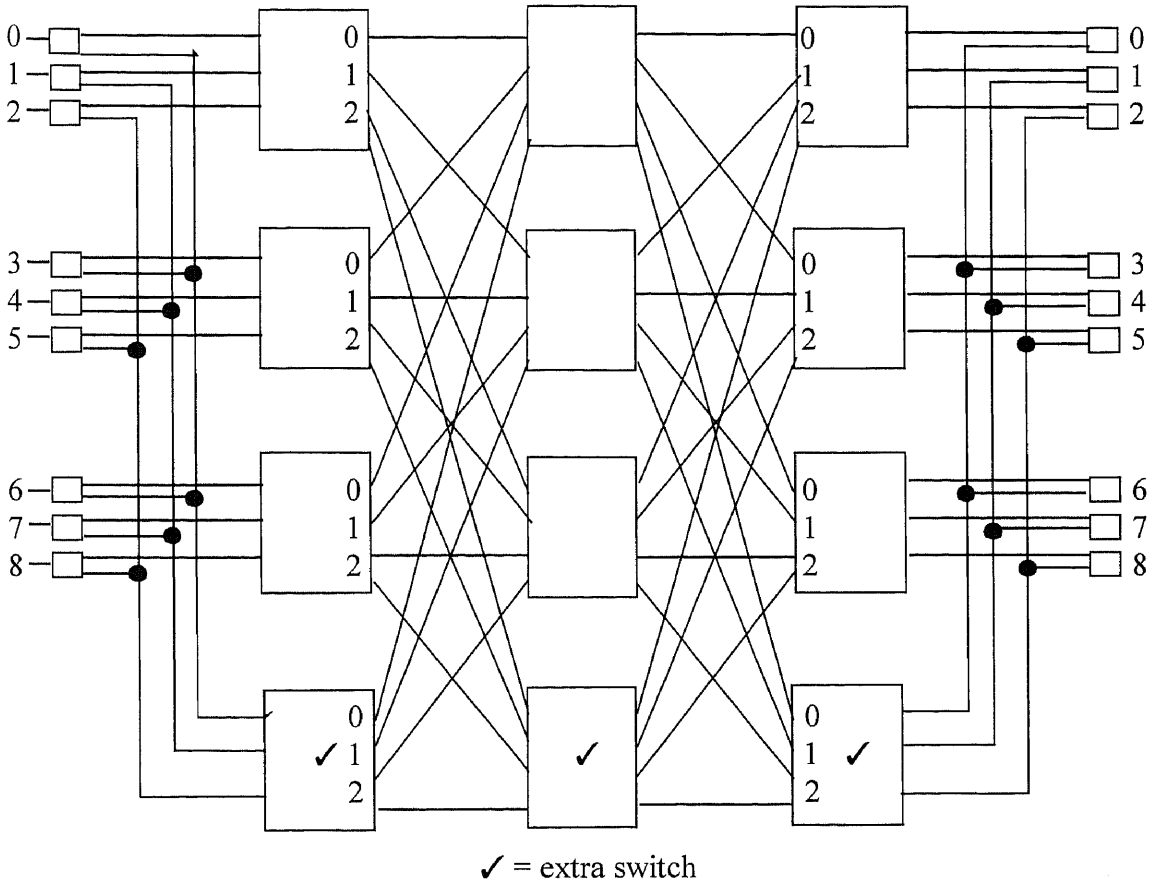

Figure 2 A Fault-Tolerant 9 x 9 Clos Network with One Extra Switch in Each Stage (Wang 1997, 3)

1.1.3 Making the Design Fault Tolerant

This general ordinary Clos network can be expanded to make it fault tolerant. This is important, as a single fault in the interconnection network can cause a severe degradation in performance unless such fault tolerant measures are provided. The fault tolerant Clos (FTC) network was developed by Hamed Nassar (Nassar and Carpinelli, 1995) and is created by adding extra switches in each stage of the Clos network, as shown in Figure 2.

The Clos network was made fault tolerant by adding the extra switches in each stage, as well as multiplexers/demultiplexers before the first stage and after the output stage. For these FTC networks, the following fault model is assumed:

1) Any switch can fail.

2) Any interstage link can fail.

3) The failure rate of multiplexers, demultiplexers, and external links is negligible (Lee, Hwang, and Carpinelli 1996, 1575).

All faults are assumed to occur independently, and faulty components are unuseable. Fault tolerance is achieved in the FTC network by redirecting inputs to multiplexers, demultiplexers, and extra switches in the outer stages when there is a fault in the outer stage switches. Should the fault occur in a middle stage switch, inputs heading for the faulty switch are redirected to one of the extra switches in the middle stage. FTC networks with y extra switches in each outer stage and x extra switches in the center stage can tolerate up to $2y + x$ switch failures, provided that no stage has more faulty switches that it has spare switches (Carpinelli and Wang 1997, 3).

1.1.4 How Fault Tolerance Might Improve Network Performance

Lee and Carpinelli have given an algorithm for routing FTC networks, and have also shown that using these extra switches even when the system displays few or no faults can significantly improve the algorithm run time in fault tolerant Clos networks (Lee, Hwang, and Carpinelli 1996, 1572-3). It will be shown that by changing the order of the swaps in the Lee and Carpinelli algorithm, the FTC network's performance can be maximized.

1.2 Outline

The rest of this thesis is organized as follows. In Chapter 2, the matrix decomposition algorithm is discussed. It is shown how the specification matrix is formed and how expansion to include fault tolerance affects both the matrix and the network hardware. The algorithm itself is then presented in detail. In Chapter 3, it is explained how the algorithm is programmed and how input and output data files are created. In Chapter 4, the order of swaps is changed and a comparison is made between three cases of the matrix decomposition algorithm. Conclusions are presented in Chapter 5.

CHAPTER 2

THE MATRIX DECOMPOSITION ALGORITHM

2.1 The Specification Matrix

2.1.1 Permutations

As shown in the diagram of Clos networks, the connections involve every input and every output, making them representable by a permutation of the form

$$P = \begin{bmatrix} 0 & 1 & \dots & i & \dots & N-1 \\ \pi(0) & \pi(1) & \dots & \pi(i) & \dots & \pi(N-1) \end{bmatrix}$$

This matrix shows that an input i is connected to an output $\pi(i)$, with the condition that $0 \le i \le N\text{-}1$, where $N = nr$, as shown earlier (Lee, Hwang, and Carpinelli 1996, 1573).

Now suppose that, for a network where $r = 4$ and $n = 3$,

$$P = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 2 & 10 & 3 & 5 & 6 & 11 & 7 & 1 & 9 & 4 & 0 & 8 \end{bmatrix}$$

Since each switch is nonblocking, the permutation can be transformed to

$$P^* = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 \\ 0 & 3 & 1 & 1 & 2 & 3 & 2 & 0 & 3 & 1 & 0 & 2 \end{bmatrix}$$

(Lee, Hwang, and Carpinelli 1996, 1573).

This is done by substituting the input numbers in the first row with the number of the switch that particular input enters in the first stage, which will always be between 0 and $r - 1$. Then, the numbers in the second row are replaced by the number of the third switch that leads to that desired output.

2.1.2 The S Matrix

The specification matrix, $S = (s_{ij})$, as developed by J. Gordon and S. Srikanthan (1990), is now introduced. This is an r x n matrix where the rows are indexed by input switches and columns by center switches, while the entries represent output switches. This means that $s_{ij} = e$ implies that a connection from the i^{th} input switch to the e^{th} output switch is sent through the j^{th} center stage switch (Carpinelli and Wang 1997, 5). As an example, for the P* given before, the S matrix would be:

$$S = \begin{bmatrix} 0 & 3 & 1 \\ 1 & 2 & 3 \\ 2 & 0 & 3 \\ 1 & 0 & 2 \end{bmatrix}$$

2.1.3 The Algorithm's Effect on the S Matrix

In this way, the S matrix is a convenient way to express a routing configuration, but this routing is only feasible if and only if S is **complete**, that is, if each column contains each output exactly once (Lee, Hwang, and Carpinelli 1996, 1573). If the initial S is not complete, the routing algorithm will find a complete S through a series of element swaps.

2.2 Fault Tolerance

2.2.1 Hardware and Matrix Expansion

As mentioned before, when the Clos network is made fault tolerant, y is the number of extra switches in each outer stage and x represents the number of extra switches in the center stage. In cases where x and/or y are not equal to 0, the extra y switches add an additional y rows to the S matrix, providing ny spare elements denoted as α, and the extra

x switches add x extra columns to the S matrix, giving rx extra spare elements represented as β (Carpinelli and Wang 1997, 5). To illustrate, the S matrix for the earlier permutation with $x = y = 1$ would be:

$$S = \begin{bmatrix} 0 & 3 & 1 & \beta \\ 1 & 2 & 3 & \beta \\ 2 & 0 & 3 & \beta \\ 1 & 0 & 2 & \beta \\ \alpha & \alpha & \alpha & \end{bmatrix}$$

2.2.2 Algorithm Expansion

The α spares correspond with paths created by the multiplexers/demultiplexers and extra switches in the outer stages. These spares may swap with any element in the same column except for β spares (Carpinelli and Wang 1997, 6). By exchanging with another element in the same column, the input switch will change, but the center switch remains constant. The β spares correspond with paths generated by extra switches in the second stage and may swap with any element in the same row except for α spares (Carpinelli and Wang 1997, 6). By exchanging with another element in the same row, the center switch will change, but the input switch remains constant. Also, the α and β spares can only be swapped if the resulting number of α spares in each column remains the same. Any violation of these rules would represent physically changing the connections and switch locations in the FTC network.

2.2.3 Fault Checking

Fault checking would usually be performed prior to the execution of the decomposition algorithm. If a non-spare switch is found to be faulty, any spare switch in that stage would be assigned as a replacement (Lee, Hwang, and Carpinelli 1996, 1576). However, in this paper, it is assumed that there are no faults in the system, as network performance using the spare switches under this no-fault condition is being explored.

2.3 The Decomposition Algorithm

2.3.1 Preprocessing

Recall that the Lee and Carpinelli decomposition algorithm provides a way to rearrange the specification matrix S to make it complete, meaning each column contains each output e exactly once. To achieve this balance, up to four kinds of swaps are performed on S, as shown below. However, before the algorithm can be executed, some preprocessing must be done and the following sets constructed from S:

1) $0(e)$, $e = 0, 1, ..., r\text{-}1$, is the set of columns $\{j\}$ such that S_j does not contain e.

   This is the set of all **e-deficient** columns.

2) $2(e)$, $e = 0, 1, ..., r\text{-}1$, is the set of columns $\{j\}$ such that S_j contains e at least twice.

   This is the set of all **e-excessive** columns.

3) (j, e), $j = 0, 1, ..., n\text{-}1$, $e = 0, 1, ..., r\text{-}1$, is the set of rows $\{i\}$ such that $S_{ij} = e$.

   (Carpinelli and Wang 1997, 5).

2.3.2 Steps of the Algorithm

These sets are used in the FTC network algorithm stated below.

Initialize by setting $e = 0$.

Step 1) If $2(e)$ is empty, i.e., $|2(e)| = 0$, set $e = e + 1$. Stop if $e = r$, otherwise repeat Step 1.

Step 2) If $2(e)$ is not empty, i.e., $|2(e)| > 0$, take its first element j. Also, take the first element k of $0(e)$.

Step 3) (Wild Swap) Set i to be the first element of (j, e). If u is an element of (j, α), i.e., $u \in (j, \alpha)$, and $u \geq r$, swap s_{ij} with s_{uj}. Remove i from (j, e), u from (j, α) and add i to (j, α), u to (j, e). For any $i \in (v, \beta)$, and $i < r$, $v \geq n$, swap s_{ij} with s_{iv}. Remove i from (j, e) and (v, β) and add i to (j, β) and (v, e). If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. If $|(v, \beta)| = x - 1$, add v to $0(\beta)$. If $|(v, \beta)| = x$, remove v from $2(\beta)$. Remove v from $0(e)$. If $|(j, e)|^* = 1$ in any of the two cases, remove j from $2(e)$. Go to Step 1. If no spares are available, go to Step 4.

Step 4) (Simple Swap) Set i to be the first element of (j, e). If $e < s_{ik}$, or $s_{ik} = \beta$, swap s_{ij} with s_{ik}. Suppose $s_{ik} = e'$. Remove i from (j, e) and (k, e'), and add i to (j, e') and (k, e). If $|(j, e)|^* = 1$, remove j from $2(e)$. If $s_{ik} = \beta$, do { If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. If $|(k, \beta)| = x - 1$, add k to $0(\beta)$. If $|(k, \beta)| = x$, remove k from $2(\beta)$. } else do { If $|(j, e')|^* = 1$, remove j from $0(e')$. If $|(j, e')|^* = 2$, add j to $2(e')$. If $|(k, e')|^* = 0$, add k to $0(e')$. If $|(k, e')|^* = 1$, remove k from $2(e')$. } Remove k from

0(e) and go to Step 1.

Step 5) (Next Simple Swap) If $e > s_{ik}$, repeat Step 4 on the second element i' of (j, e). If $e > s_{i'k}$ or $s_{i'k} = \alpha$, go to Step 6.

Step 6) (Successive Swap) Divide into substeps:

A) Set $u = e$. Remove k from 0(u). If $|(j, u)|^* = 2$, remove j from 2(u).

B) Set $v = s_{ik}$. Swap s_{ij} with s_{ik}. Remove i from (j, u) and (k, v), and add i to (j, v) and (k, u).

C) Suppose $e < v$ or $v = \beta$ or $|(j, v)|^* = 1$. If $v = \beta$, do { If $|(k, \beta)| = x - 1$, add k to 0(β). If $|(j, \beta)| = x$, remove k from 2(β). If $|(j, \beta)| = x$, remove j from 0(β). If $|(j, \beta)| = x + 1$, add j to 2(β). } else do { If $|(k, v)|^* = 0$, add k to 0(v). If $|(k, v)|^* = 1$, remove k from 2(v). If $|(j, v)|^* = 1$, remove j from 0(v). If $|(j, v)|^* = 2$, add j to 2(v). } Go to Step 1.

D) Suppose $e > v$ or $v = \alpha$. Set $u = v$. Take i from (j, v) and go to Step 6B.

Annotation: $|(j, e)|$ is the cardinality of (j, e).

$|(j, e)|^*$ is the number of entries which are less than r in the set (j, e)

(Carpinelli and Wang 1997, 6-7).

2.3.3 An Example

The following is an example of how this algorithm works:

Consider a Clos network with $r = 5$ and $n = 3$, that is, a network with five 3 x 3 switches in the outer stages and three 5 x 5 switches in the center stage. A sample permutation for such a network could be:

$$P = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 6 & 11 & 3 & 8 & 5 & 12 & 1 & 14 & 0 & 7 & 13 & 2 & 9 & 4 & 10 \end{bmatrix}$$

which is then transformed into:

$$P^* = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 & 4 & 4 & 4 \\ 2 & 3 & 1 & 2 & 1 & 4 & 0 & 4 & 0 & 2 & 4 & 0 & 3 & 1 & 3 \end{bmatrix}$$

The specification matrix S is thus:

$$S = \begin{bmatrix} 2 & 3 & 1 \\ 2 & 1 & 4 \\ 0 & 4 & 0 \\ 2 & 4 & 0 \\ 3 & 1 & 3 \end{bmatrix}$$

Extending this network to a FTC network with $x = 1$ and $y = 2$, S becomes:

$$S = \begin{bmatrix} 2 & 3 & 1 & \beta \\ 2 & 1 & 4 & \beta \\ 0 & 4 & 0^* & \beta \\ 2 & 4 & 0 & \beta \\ 3 & 1 & 3 & \beta \\ \alpha & \alpha & \alpha^* & \\ \alpha & \alpha & \alpha & \end{bmatrix}$$

Now the three types of sets can be constructed as follows:

$0(0) = \{ 1, 3 \}$
$0(1) = \{ 0, 3 \}$
$0(2) = \{ 1, 2, 3 \}$
$0(3) = \{ 3 \}$
$0(4) = \{ 0, 3 \}$
$0(\alpha) = \{ 3 \}$
$0(\beta) = \{ 0, 1, 2 \}$

$2(0) = \{ 2 \}$
$2(1) = \{ 1 \}$
$2(2) = \{ 0 \}$
$2(3) = \{ \}$
$2(4) = \{ 1 \}$
$2(\alpha) = \{ \}$
$2(\beta) = \{ 3 \}$

(0, 0) = { 2 }	(1, 0) = { }	(2, 0) = { 2, 3 }	(3, 0) = { }
(0, 1) = { 1}	(1, 1) = { 1, 4 }	(2, 1) = { 0 }	(3, 1) = { }
(0, 2) = {0,1,3}	(1, 2) = { }	(2, 2) = { }	(3, 2) = { }
(0, 3) = { 4 }	(1, 3) = { 0 }	(2, 3) = { 4 }	(3, 3) = { }
(0, 4) = { }	(1, 4) = { 2, 3 }	(2, 4) = { 1 }	(3, 4) = { }
(0, α) = { 5, 6 }	(1, α) = { 5, 6 }	(2, α) = { 5, 6 }	(3, α) = { }
(0, β) = { }	(1, β) = { }	(2, β) = { }	(3,β)={0,1,2,3,4}

Now the swapping can begin.

1) Wild Swap, $e = 0, j = 2$. One alpha swap is performed.

$$S = \begin{bmatrix} 2 & 3 & 1 & \beta \\ 2 & 1^* & 4 & \beta \\ 0 & 4 & \alpha & \beta \\ 2 & 4 & 0 & \beta \\ 3 & 1 & 3 & \beta \\ \alpha & \alpha^* & 0 & \\ \alpha & \alpha & \alpha & \end{bmatrix}$$

0(0) = { 1, 3 }
0(1) = { 0, 3 }
0(2) = { 1, 2, 3 }
0(3) = { 3 }
0(4) = { 0, 3 }
0(α) = { 3 }
0(β) = { 0, 1, 2 }

2(0) = { }
2(1) = { 1 }
2(2) = { 0 }
2(3) = { }
2(4) = { 1 }
2(α) = { }
2(β) = { 3 }

(0, 0) = { 2 }	(1, 0) = { }	(2, 0) = {3, 5}	(3, 0) = { }
(0, 1) = { }	(1, 1) = {1, 4}	(2, 1) = { 0 }	(3, 1) = { }
(0, 2) = {0,1,3}	(1, 2) = { }	(2, 2) = { }	(3, 2) = { }
(0, 3) = { 4 }	(1, 3) = { 0 }	(2, 3) = { 4 }	(3, 3) = { }
(0, 4) = { }	(1, 4) = {2, 3}	(2, 4) = { 1 }	(3, 4) = { }
(0, α) = {5, 6}	(1, α) = {5, 6}	(2, α) = {2, 6}	(3, α) = { }
(0, β) = { }	(1, β) = { }	(2, β) = { }	(3,β)={0,1,2,3,4}

2) Wild Swap: $e = 1, j = 1$. One alpha swap is performed.

$$S = \begin{bmatrix} 2^* & 3 & 1 & \beta \\ 2^* & \alpha & 4 & \beta^* \\ 0 & 4 & \alpha & \beta \\ 2 & 4 & 0 & \beta \\ 3 & 1 & 3 & \beta \\ \alpha^* & 1 & 0 & \\ \alpha & \alpha & \alpha & \end{bmatrix}$$

$0(0) = \{ 1, 3 \}$
$0(1) = \{ 0, 3 \}$
$0(2) = \{ 1, 2, 3 \}$
$0(3) = \{ 3 \}$
$0(4) = \{ 0, 3 \}$
$0(\alpha) = \{ 3 \}$
$0(\beta) = \{ 0, 1, 2 \}$

$2(0) = \{ \}$
$2(1) = \{ \}$
$2(2) = \{ 0 \}$
$2(3) = \{ \}$
$2(4) = \{ 1 \}$
$2(\alpha) = \{ \}$
$2(\beta) = \{ 3 \}$

$(0, 0) = \{ 2 \}$	$(1, 0) = \{ \}$	$(2, 0) = \{3, 5\}$	$(3, 0) = \{ \}$
$(0, 1) = \{ \}$	$(1, 1) = \{4, 5\}$	$(2, 1) = \{ 0 \}$	$(3, 1) = \{ \}$
$(0, 2) = \{0,1,3\}$	$(1, 2) = \{ \}$	$(2, 2) = \{ \}$	$(3, 2) = \{ \}$
$(0, 3) = \{ 4 \}$	$(1, 3) = \{ 0 \}$	$(2, 3) = \{ 4 \}$	$(3, 3) = \{ \}$
$(0, 4) = \{ \}$	$(1, 4) = \{2, 3\}$	$(2, 4) = \{ 1 \}$	$(3, 4) = \{ \}$
$(0, \alpha) = \{5, 6\}$	$(1, \alpha) = \{1, 6\}$	$(2, \alpha) = \{2, 6\}$	$(3, \alpha) = \{ \}$
$(0, \beta) = \{ \}$	$(1, \beta) = \{ \}$	$(2, \beta) = \{ \}$	$(3,\beta)=\{0,1,2,3,4\}$

3) Wild Swap: $e = 2, j = 0$. One alpha swap and one beta swap are performed.

$$S = \begin{bmatrix} \alpha & 3 & 1 & \beta \\ \beta & \alpha & 4 & 2 \\ 0 & 4^* & \alpha & \beta \\ 2 & 4 & 0 & \beta \\ 3 & 1 & 3 & \beta \\ 2 & 1 & 0 & \\ \alpha & \alpha^* & \alpha & \end{bmatrix}$$

$0(0) = \{ 1, 3 \}$
$0(1) = \{ 0, 3 \}$
$0(2) = \{ 1, 2 \}$
$0(3) = \{ 3 \}$
$0(4) = \{ 0, 3 \}$
$0(\alpha) = \{ 3 \}$
$0(\beta) = \{ 1, 2 \}$

$2(0) = \{ \}$
$2(1) = \{ \}$
$2(2) = \{ \}$
$2(3) = \{ \}$
$2(4) = \{ 1 \}$
$2(\alpha) = \{ \}$
$2(\beta) = \{ 3 \}$

(0, 0) = {1, 3}	(1, 0) = { }	(2, 0) = {3, 5}	(3, 0) = { }
(0, 1) = {0, 3}	(1, 1) = {4, 5}	(2, 1) = { 0 }	(3, 1) = { }
(0, 2) = {3,5,6}	(1, 2) = { }	(2, 2) = { }	(3, 2) = { 1 }
(0, 3) = { 4 }	(1, 3) = { 0 }	(2, 3) = { 4 }	(3, 3) = { }
(0, 4) = { }	(1, 4) = {2, 3}	(2, 4) = { 1 }	(3, 4) = { }
(0, α) = { 0 }	(1, α) = {1, 6}	(2, α) = {2, 6}	(3, α) = { }
(0, β) = { 1 }	(1, β) = { }	(2, β) = { }	(3,β)= {0,2,3,4}

4) 2(3) is empty. Therefore, no swaps are performed when e = 3.

Let $e = e + 1$.

5) Wild Swap: $e = 4, j = 1$. One alpha swap is performed.

$$S = \begin{bmatrix} \alpha & 3 & 1 & \beta \\ \beta & \alpha & 4 & 2 \\ 0 & \alpha & \alpha & \beta \\ 2 & 4 & 0 & \beta \\ 3 & 1 & 3 & \beta \\ 2 & 1 & 0 & \\ \alpha & 4 & \alpha & \end{bmatrix}$$

0(0) = { 1, 3 }
0(1) = { 0, 3 }
0(2) = { 1, 2 }
0(3) = { 3 }
0(4) = { 0, 3 }
0(α) = { 3 }
0(β) = { 1, 2 }

2(0) = { }
2(1) = { }
2(2) = { }
2(3) = { }
2(4) = { }
2(α) = { }
2(β) = { 3 }

(0, 0) = { 2 }	(1, 0) = { }	(2, 0) = {3, 5}	(3, 0) = { }
(0, 1) = { }	(1, 1) = {4, 5}	(2, 1) = { 0 }	(3, 1) = { }
(0, 2) = {3,5,6}	(1, 2) = { }	(2, 2) = { }	(3, 2) = { 1 }
(0, 3) = { 4 }	(1, 3) = { 0 }	(2, 3) = { 4 }	(3, 3) = { }
(0, 4) = { }	(1, 4) = {3, 6}	(2, 4) = { 1 }	(3, 4) = { }
(0, α) = { 0 }	(1, α) = {1, 2}	(2, α) = {2, 6}	(3, α) = { }
(0, β) = { 1 }	(1, β) = { }	(2, β) = { }	(3,β)= {0,2,3,4}

Total number of alpha swaps: 4.

Total number of beta swaps: 1.

Total number of simple/next simple swaps: 0.

Total number of successive swaps: 0.

Total number of swaps: 5.

CHAPTER 3

PROGRAMMING AND EXECUTING THE ALGORITHM

3.1 The Program

3.1.1 Program Structure

The program clos.c was written to implement this version of the Lee and Carpinelli matrix decomposition algorithm. This program was coded using the C programming language and arrays were used to represent matrix S and the sets $0(e)$, $2(e)$, and (j, e) as described below.

The matrix S is implemented as a two dimensional array. Element S[i][j] of the array would represent element s_{ij} of the S matrix with i designating the row and j the column.

Set $0(e)$ is implemented as a two dimensional array Zero[i][j], where i represents e and j is a counting index. For example, if $0(4) = \{0, 1, 3\}$, then Zero[4][0] = 0, Zero[4][1] = 1, and Zero[4][2] = 3.

Set $2(e)$ is implemented in the same way. As an example, if $2(2) = \{1, 4\}$, then Two[2][0] = 1 and Two[2][1] = 4.

Set (j, e) is implemented as a three dimensional array JE[x][y][z] where $x = j$, $y = e$ and z is the counting index. For example, if $(j, e) = (2, 3) = \{0, 4\}$, then JE[2][3][0] = 0 and JE[2][3][1] = 4.

All of the above arrays are initialized to -1, which was chosen rather than 0 because 0 is a valid entry in the arrays.

3.1.2 Special Cases for α and β

Special cases of the arrays were created for α and β, as in 0(e) and 2(e), $e = i$ must be between 0 and r. For (j, e), $x = j$ must be between 0 and $n + x$ and $y = e$ is between 0 and r. The numerical representations of α and β are 65 and 66 (corresponding to the integer values the C language assigns to characters 'A' and 'B'), which are outside the above intervals. The set 0(α) = {1, 4} is represented by the array Zero[65][0] = 1 and Zero[65][1] = 4 and is created with a special part of the function outside the regular loop that looks for values of 'A' = 65 in the matrix.

3.2 Program Structure

3.2.1 Input File

The input file for clos.c is straightforward and in the following form:

4 3 1 1

2 1 2 1

0 2 2 1

1 0 0 0

The first number given is n, the second is r, the third is x, and the fourth is y. The remaining lines give the original input matrix S. The program then sends this input matrix to a function that adds the appropriate number of rows of α's (or 65's) and β's (or 66's) as specified by y and x, respectively. For the elements located where the rows of α's and columns of β's intersect, the value 120 (the number the C programming language assigns to the character "x") is inserted. This value, essentially a placeholder, lets the program

know that there is really no element present. Figure 3 shows a comparison between an actual S matrix and the computer representation.

Actual S Matrix **Computer Representation**

Figure 3 Comparison of Actual Matrix and Computer Representation

After the matrix is created, $0(e)$, $2(e)$, and (j, e) are found and the swapping begins, as detailed in the earlier algorithm.

3.2.2 Testing Cardinality

When the swapping function checks to see if $2(e)$ is empty, it simply tests if Two[e][0] = -1. Since these arrays were initialized to -1, this value at the top of the array would indicate that the array is empty. To find the cardinality of an array, the program checks each element until it finds one of value -1. Then, if JE[j][e][x-1] is the first element equal to -1, the cardinality of $(j, e) = |(j, e)| = x - 1$.

3.2.3 Creating the Data File

The original algorithm as presented above was then run on 38 input sets of 499 randomly generated matrices to create a baseline set of data for the number of swaps needed to decompose each type of matrix on average. The program keeps track of the total number

of each kind of swap and sends that information to a file called output upon completion. The dimensions of the test matrices are given in Table1.

A separate program called matrix.c created the input file of 499 matrices using a random number generator. A copy of this program can be found in the appendix. After clos.c ran on this input file and wrote the results to file "output", one last manipulation had to be performed to convert this raw data into an average. Another program called average.c, which can also be found in the appendix, added up the number of each swap type, divided that by the total number (which in this case was 499), and gave the result correct to four decimal places. The results for each particular matrix dimension can be found on pages 28 - 65.

Table 1 Dimensions of the Test Matrices

n	r	x	y		n	r	x	y
5	6	0 1 2 2 2	0 1 2 3 4		2	5	0 3 2 1 1	0 1 2 4 1
3	4	0 1 2 1 2	0 1 3 4 2		4	4	0 1 2 3 1	0 1 2 2 4
6	5	0 3 1 2 2	0 2 1 4 2		6	4	0 1 2 4	0 1 2 1
5	3	0 1 2 4 2	0 1 3 1 2		8	3	0 1 1 2	0 1 4 2

CHAPTER 4

RE-ORDERING THE SWAPS

4.1 Adjustments to the clos.c Program

4.1.1 Rewriting the Swapping Algorithm

After running the original algorithm, the order of swaps was then changed in an effort to reduce the total number necessary to completely decompose the matrix and, thus, configure the fault tolerant Clos network to most efficiently use the extra switches in a no-fault case. The original order of swaps was as follows:

(Case 1) 1) Wild Swap, 2) Simple Swap, 3) Next Simple Swap, 4) Successive Swap.

The algorithm for this order of swaps is detailed in Chapter 2.

Now the swapping function in clos.c was modified to reflect the following:

(Case 2) 1) Simple Swap, 2) Wild Swap, 3) Next Simple Swap, 4) Successive Swap

The algorithm now becomes:

Initialize by setting $e = 0$.

Step 1) If $2(e)$ is empty, i.e., $|2(e)| = 0$, set $e = e + 1$. Stop if $e = r$, otherwise repeat Step 1.

Step 2) If $2(e)$ is not empty, i.e., $|2(e)| > 0$, take its first element j. Also, take the first element k of $0(e)$.

Step 3) (Simple Swap) Set i to be the first element of (j, e). If $e < s_{ik,}$ or $s_{ik} = \beta$, swap s_{ij} with s_{ik}. Suppose $s_{ik} = e'$. Remove i from (j, e) and (k, e'), and add i to (j, e') and (k, e). If $|(j, e)|^* = 1$, remove j from $2(e)$. If $s_{ik} = \beta$, do { If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. If $|(k, \beta)| =$

x - 1, add k to $0(\beta)$. If $|(k, \beta)| = x$, remove k from $2(\beta)$. } else do { If $|(j, e')|^* = 1$, remove j from $0(e')$. If $|(j, e')|^* = 2$, add j to $2(e')$. If $|(k, e')|^* = 0$, add k to $0(e')$. If $|(k, e')|^* = 1$, remove k from $2(e')$. } Remove k from $0(e)$ and go to Step 1.

Step 4) (Wild Swap) Set i to be the first element of (j, e). If u is an element of (j, α), i.e., $u \in (j, \alpha)$, and $u \geq r$, swap s_{ij} with s_{uj}. Remove i from (j, e), u from (j, α) and add i to (j, α), u to (j, e). For any $i \in (v, \beta)$, and $i < r$, $v \geq n$, swap s_{ij} with s_{iv}. Remove i from (j, e) and (v, β) and add i to (j, β) and (v, e). If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. If $|(v, \beta)| = x$ - 1, add v to $0(\beta)$. If $|(v, \beta)| = x$, remove v from $2(\beta)$. Remove v from $0(e)$. If $|(j, e)|^* = 1$ in any of the two cases, remove j from $2(e)$. Go to Step 1. If no spares are available, go to Step 5.

Step 5) (Next Simple Swap) If $e > s_{ik}$, repeat Step 3 on the second element i' of (j, e). If $e > s_{i'k}$ or $s_{i'k} = \alpha$, go to Step 6.

Step 6) (Successive Swap) Divide into substeps:

A) Set $u = e$. Remove k from $0(u)$. If $|(j, u)|^* = 2$, remove j from $2(u)$.

B) Set $v = s_{ik}$. Swap s_{ij} with s_{ik}. Remove i from (j, u) and (k, v), and add i to (j, v) and (k, u).

C) Suppose $e < v$ or $v = \beta$ or $|(j, v)|^* = 1$. If $v = \beta$, do { If $|(k, \beta)| = x$ - 1, add k to $0(\beta)$. If $|(j, \beta)| = x$, remove k from $2(\beta)$. If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. } else do { If $|(k, v)|^* = 0$, add k to $0(v)$. If $|(k, v)|^* = 1$, remove k from $2(v)$. If $|(j, v)|^* = 1$,

remove j from $0(v)$. If $|(j, v)|^* = 2$, add j to $2(v)$. } Go to Step 1.

D) Suppose $e > v$ or $v = \alpha$. Set $u = v$. Take i from (j, v) and go to Step 6B.

Annotation: $|(j, e)|$ is the cardinality of (j, e).

$|(j, e)|^*$ is the number of entries which are less than r in the set (j, e).

(Case 3) 1) Simple Swap, 2) Next Simple Swap, 3) Wild Swap, 4) Successive Swap.

The algorithm now becomes:

Initialize by setting $e = 0$.

Step 1) If $2(e)$ is empty, i.e., $|2(e)| = 0$, set $e = e + 1$. Stop if $e = r$, otherwise repeat Step 1.

Step 2) If $2(e)$ is not empty, i.e., $|2(e)| > 0$, take its first element j. Also, take the first element k of $0(e)$.

Step 3) (Simple Swap) Set i to be the first element of (j, e). If $e < s_{ik}$, or $s_{ik} = \beta$, swap s_{ij} with s_{ik}. Suppose $s_{ik} = e'$. Remove i from (j, e) and (k, e'), and add i to (j, e') and (k, e). If $|(j, e)|^* = 1$, remove j from $2(e)$. If $s_{ik} = \beta$, do { If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. If $|(k, \beta)| = x - 1$, add k to $0(\beta)$. If $|(k, \beta)| = x$, remove k from $2(\beta)$. } else do { If $|(j, e')|^* = 1$, remove j from $0(e')$. If $|(j, e')|^* = 2$, add j to $2(e')$. If $|(k, e')|^* = 0$, add k to $0(e')$. If $|(k, e')|^* = 1$, remove k from $2(e')$. } Remove k from $0(e)$ and go to Step 1.

Step 4) (Next Simple Swap) If $e > s_{ik}$, repeat Step 3 on the second element i' of (j, e). If $e > s_{i'k}$ or $s_{i'k} = \alpha$, go to Step 6.

Step 5) (Wild Swap) Set i to be the first element of (j, e). If u is an element of (j, α), i.e., $u \in (j, \alpha)$, and $u \geq r$, swap s_{ij} with s_{uj}. Remove i from (j, e), u from (j, α) and add i to (j, α), u to (j, e). For any $i \in (v, \beta)$, and $i < r$, $v \geq n$, swap s_{ij} with s_{iv}. Remove i from (j, e) and (v, β) and add i to (j, β) and (v, e). If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. If $|(v, \beta)| = x - 1$, add v to $0(\beta)$. If $|(v, \beta)| = x$, remove v from $2(\beta)$. Remove v from $0(e)$. If $|(j, e)|^* = 1$ in any of the two cases, remove j from $2(e)$. Go to Step 1. If no spares are available, go to Step 6.

Step 6) (Successive Swap) Divide into substeps:

A) Set $u = e$. Remove k from $0(u)$. If $|(j, u)|^* = 2$, remove j from $2(u)$.

B) Set $v = s_{ik}$. Swap s_{ij} with s_{ik}. Remove i from (j, u) and (k, v), and add i to (j, v) and (k, u).

C) Suppose $e < v$ or $v = \beta$ or $|(j, v)|^* = 1$. If $v = \beta$, do { If $|(k, \beta)| = x - 1$, add k to $0(\beta)$. If $|(j, \beta)| = x$, remove k from $2(\beta)$. If $|(j, \beta)| = x$, remove j from $0(\beta)$. If $|(j, \beta)| = x + 1$, add j to $2(\beta)$. } else do { If $|(k, v)|^* = 0$, add k to $0(v)$. If $|(k, v)|^* = 1$, remove k from $2(v)$. If $|(j, v)|^* = 1$, remove j from $0(v)$. If $|(j, v)|^* = 2$, add j to $2(v)$. } Go to Step 1.

D) Suppose $e > v$ or $v = \alpha$. Set $u = v$. Take i from (j, v) and go to Step 6B.

Annotation: $|(j, e)|$ is the cardinality of (j, e).

$|(j, e)|^*$ is the number of entries which are less than r in the set (j, e).

The new swapping functions for Cases 2 and 3 written in the C programming language can be found in the appendix.

4.1.2 Executing the Adjusted Programs

In order for a valid comparison to be made of the number of swaps needed to decompose the S matrix using each of the three cases, it was very important that the same set of 500 input matrices were used when running each case. Therefore, the matrix.c program was executed only once, and the sample matrices that were produced using the input dimensions detailed in Table 1 were used as inputs for each case. By keeping the inputs and the rest of the clos.c program constant, it was assured that any variation in the results was due to the adjustments made in the swapping function. Once the raw data was produced by the clos.c program, the same average.c program was run to produce a meaningful data file. The full results from running each of the three cases can be found on pages 28 - 65.

4.2 Results of the Swap Re-Ordering

4.2.1 Discussion of Results

Regardless of which swap order is used, the results when $x = y = 0$ are identical. This is because the only type of swap whose place is changed is the wild swap, and when $x = y = 0$, there are no extra switches in the network. Thus, no wild swaps take place. In all three cases, if the wild swap is ignored, the simple swap comes first, followed by the next simple swap and successive swap. However, when x and y are not 0, there are extra switches in the network and wild swaps take place, causing the number of necessary swaps to vary from case to case.

As can be seen in the following figures and tables, the number of total swaps in Cases 2 and 3 are nearly the same in the fault tolerant cases. Thus, it can be concluded that once the simple swap is performed at the beginning of the swapping algorithm, the number of additional swaps needed to finish decomposing the S matrix remains fairly constant. Neither Case 2 nor Case 3 required any successive swaps in the decomposition. After completing the simple swaps, Case 2 relied almost entirely on alpha swaps, with a nearly insignificant number of beta swaps. Case 3 used almost an identical number of alpha and next simple swaps (again with a nearly insignificant number of beta swaps) after the simple swaps to finish decomposing the S matrix. Another interesting note about Cases 2 and 3 is that the number of each type of swap remains fairly uniform when n and r are held constant. Varying x and y had little effect on the amount of swaps required for decomposition of matrices with extra switches.

It can also be seen from the following graphs that the total number of swaps needed for decomposition is greater for Case 1 than for the other two cases in almost every data set. Case 1 decomposition relies heavily on alpha swaps, with a few beta swaps. Very few simple or next simple swaps occur in the fault tolerant data sets (when x and y are not 0), and successive swaps are virtually never seen.

4.2.2 Graphical Representation of the Results

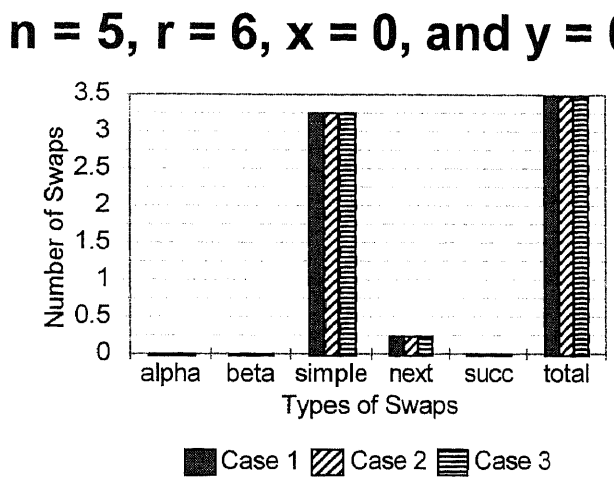

Figure 4 Number of Swaps in Each Case

Table 2
Number of Swaps in Each Case for n = 5, r = 6, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	3.2456	3.2456	3.2456
next simple	.2343	.2343	.2343
successive	0	0	0
total	3.4798	3.4798	3.4798

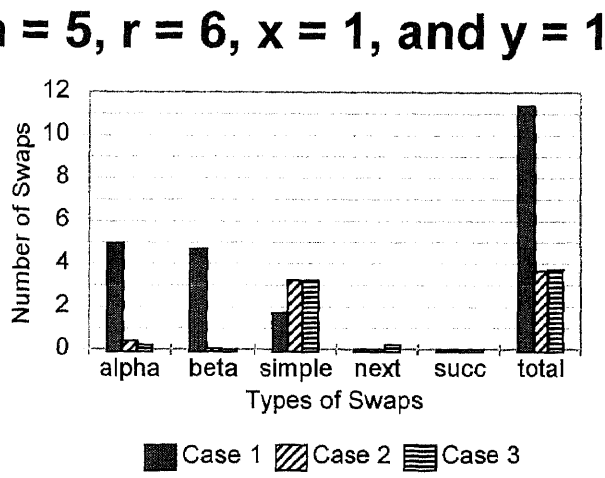

Figure 5 Number of Swaps in Each Case

Table 3
Number of Swaps in Each Case for n = 5, r = 6, x = 1, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	4.9587	.3922	.2263
beta	4.6924	.044	.0263
simple	1.7018	3.2499	3.2403
next simple	.0133	0	.2323
successive	.0077	0	0
total	11.3739	3.6861	3.7251

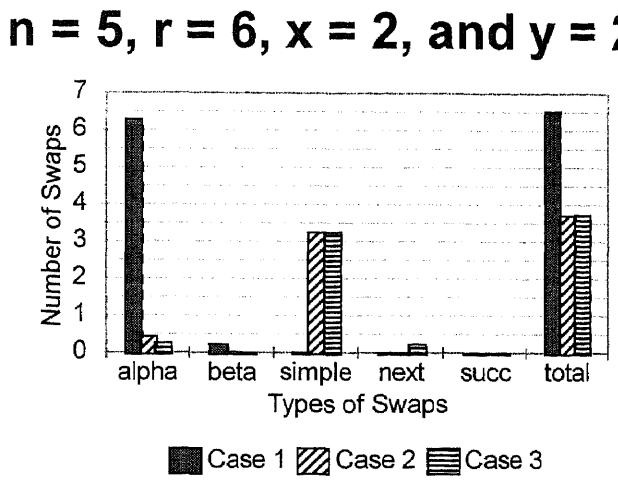

Figure 6 Number of Swaps in Each Case

Table 4
Number of Swaps in Each Case for n = 5, r = 6, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	6.2786	.4345	.2529
beta	.2336	.0273	.0047
simple	0	3.2502	3.2403
next simple	0	0	.2323
successive	0	0	0
total	6.5122	3.7121	3.7301

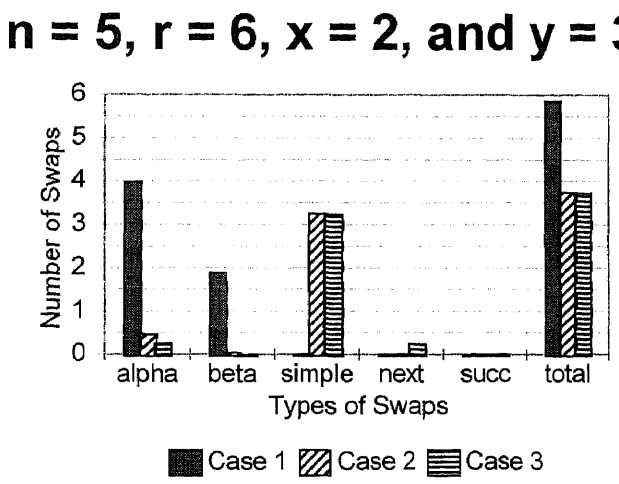

Figure 7 Number of Swaps in Each Case

Table 5
Number of Swaps in Each Case for n = 5, r = 6, x = 2, and y = 3

Type of Swap	Case 1	Case 2	Case 3
alpha	3.9747	.4608	.2572
beta	1.8794	.0273	.0027
simple	0	3.2502	3.2403
next simple	0	0	.2323
successive	0	0	0
total	5.854	3.7384	3.7324

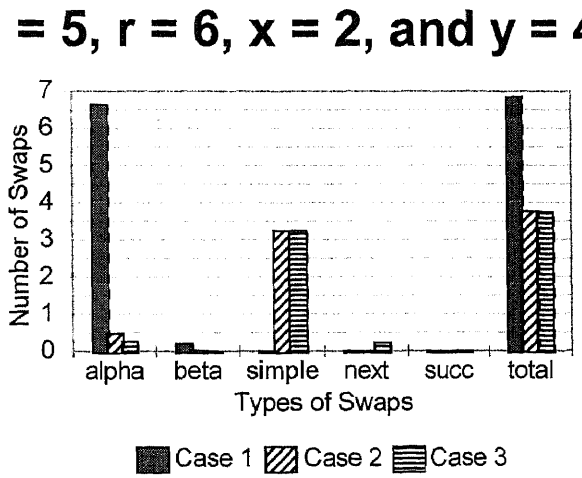

Figure 8 Number of Swaps in Each Case

Table 6
Number of Swaps in Each Case for n = 5, r = 6, x = 2, and y = 4

Type of Swap	Case 1	Case 2	Case 3
alpha	6.6285	.4872	.2596
beta	.2116	.0273	.0027
simple	0	3.2502	3.2403
next simple	0	0	.2323
successive	0	0	0
total	6.8401	3.7647	3.7348

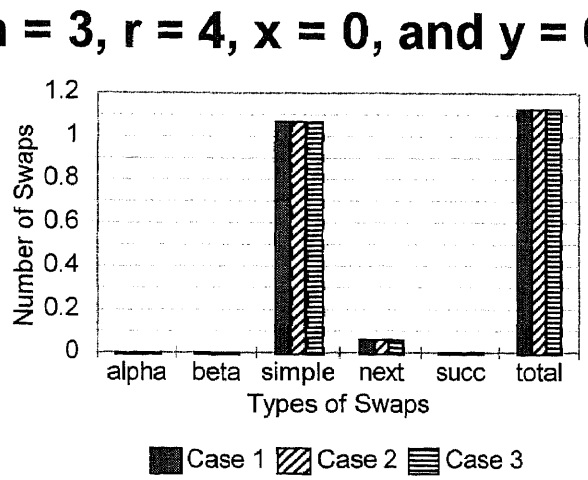

Figure 9 Number of Swaps in Each Case

Table 7
Number of Swaps In Each Case For n = 3, r = 4, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	1.0663	1.0663	1.0663
next simple	.0603	.0603	.0603
successive	0	0	0
total	1.1266	1.1266	1.1266

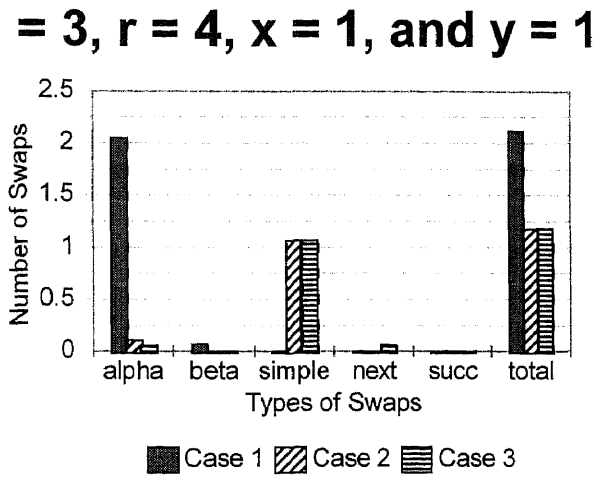

Figure 10 Number of Swaps in Each Case

Table 8
Number of Swaps in Each Case for n = 3, r = 4, x = 1, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	2.0503	.1083	.058
beta	.0653	.002	.0017
simple	0	1.068	1.0666
next simple	0	0	.06
successive	0	0	0
total	2.1156	1.1783	1.1863

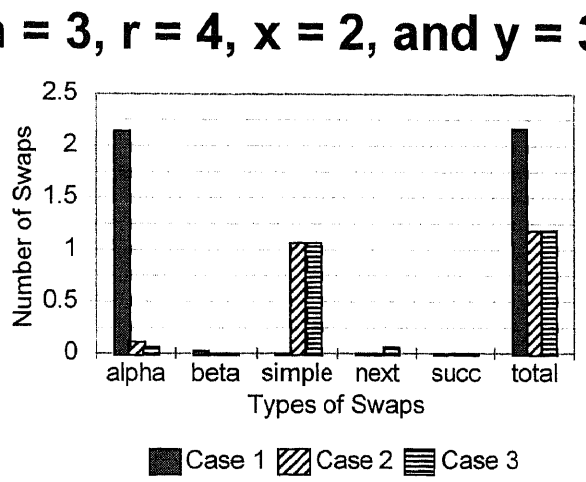

Figure 11 Number of Swaps in Each Case

Table 9
Number of Swaps in Each Case for n = 3, r = 4, x = 2, and y = 3

Type of Swap	Case 1	Case 2	Case 3
alpha	2.1403	.1123	.06
beta	.0247	.002	0
simple	0	1.068	1.0666
next simple	0	0	.06
successive	0	0	0
total	2.1649	1.1823	1.1866

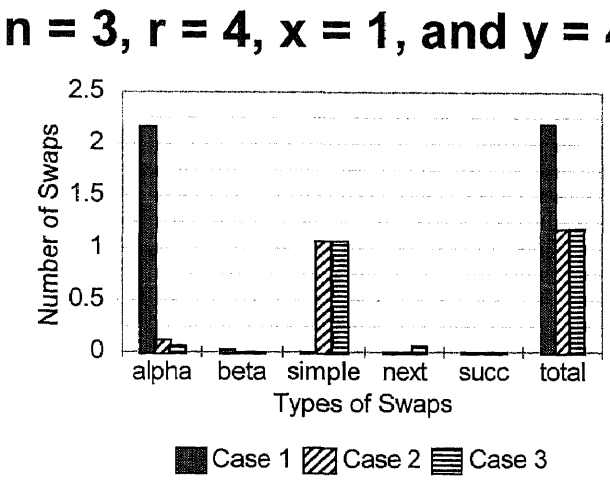

Figure 12 Number of Swaps in Each Case

Table 10
Number of Swaps in Each Case for n = 3, r = 4, x = 1, and y = 4

Type of Swap	Case 1	Case 2	Case 3
alpha	2.1649	.1143	.06
beta	.0247	.002	0
simple	0	1.068	1.0666
next simple	0	0	.06
successive	0	0	0
total	2.1896	1.1843	1.1866

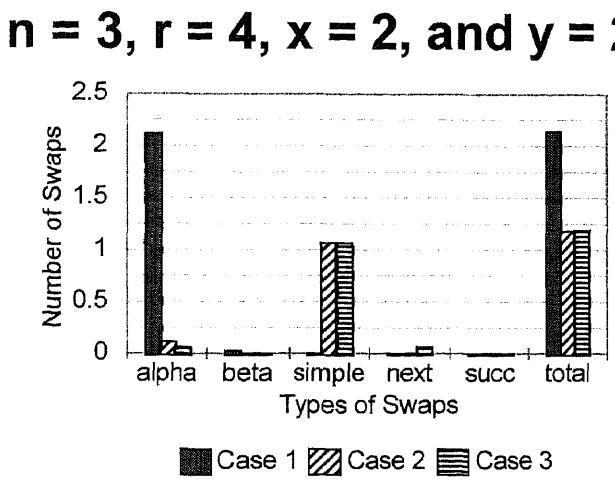

Figure 13 Number of Swaps in Each Case

Table 11
Number of Swaps in Each Case for n = 3, r = 4, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	2.1156	.1103	.06
beta	.0247	.002	0
simple	0	1.068	1.0666
next simple	0	0	.06
successive	0	0	0
total	2.1403	1.1803	1.1866

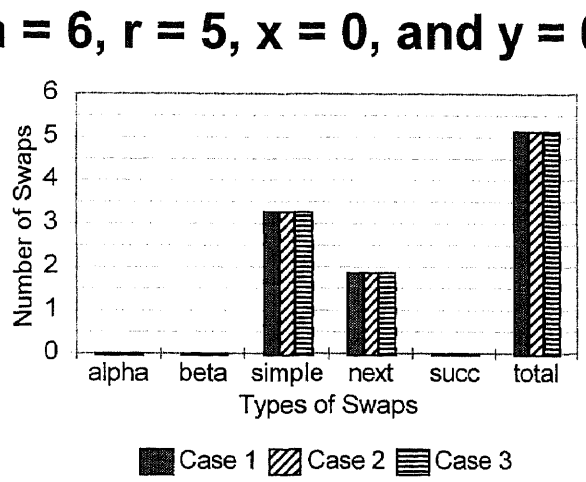

Figure 14 Number of Swaps in Each Case

Table 12
Number of Swaps in Each Case for n = 6, r = 5, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	3.2646	3.2646	3.2646
next simple	1.8677	1.8677	1.8677
successive	0	0	0
total	5.1323	5.1323	5.1323

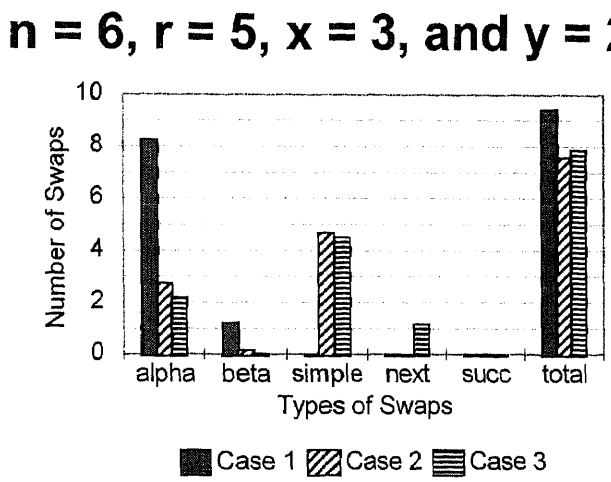

Figure 15 Number of Swaps in Each Case

Table 13
Number of Swaps in Each Case for n = 6, r = 5, x = 3, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	8.248	2.7441	2.19
beta	1.1982	.148	.022
simple	0	4.69	4.5124
next simple	0	0	1.1523
successive	0	0	0
total	9.4462	7.5821	7.8767

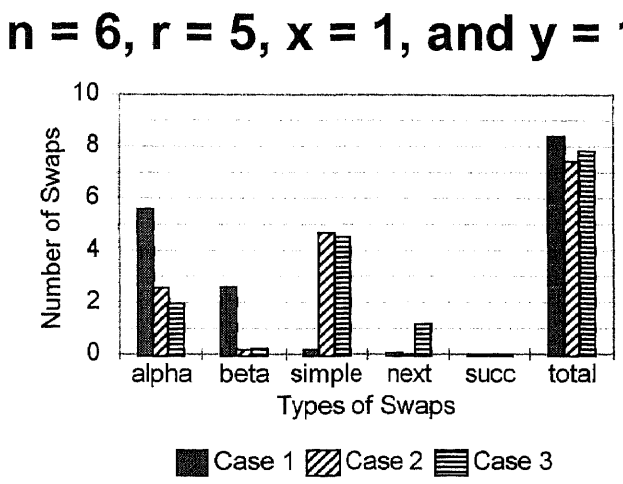

Figure 16 Number of Swaps in Each Case

Table 14
Number of Swaps in Each Case for n = 6, r = 5, x = 1, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	5.5963	2.5582	1.9542
beta	2.6046	.1819	.218
simple	.1615	4.694	4.5123
next simple	.0476	0	1.152
successive	0	0	0
total	8.4099	7.4341	7.865

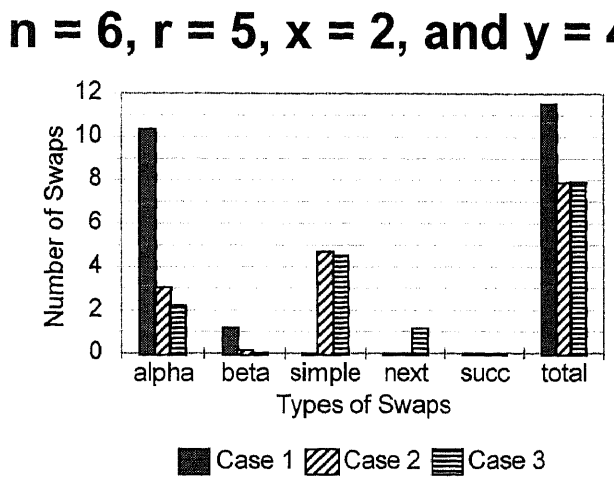

Figure 17 Number of Swaps in Each Case

Table 15
Number of Swaps in Each Case for n = 6, r = 5, x = 2, and y = 4

Type of Swap	Case 1	Case 2	Case 3
alpha	10.3324	3.04	2.2341
beta	1.198	.1486	.018
simple	0	4.69	4.512
next simple	0	0	1.1518
successive	0	0	0
total	11.534	7.8786	7.9159

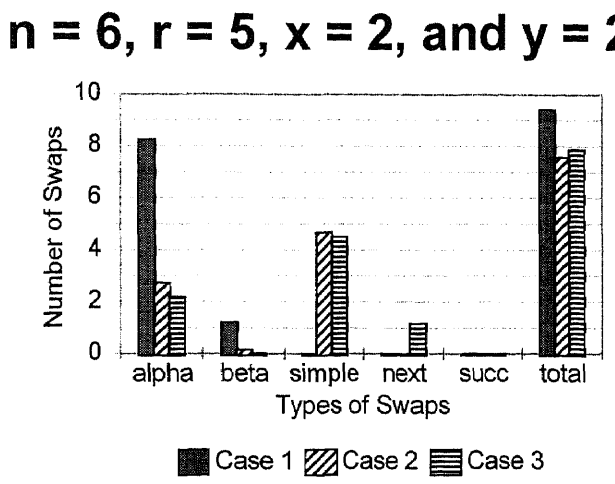

Figure 18 Number of Swaps in Each Case

Table 16
Number of Swaps in Each Case for n = 6, r = 5, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	8.2481	2.7436	2.19
beta	1.198	.1482	.0224
simple	0	4.69	4.512
next simple	0	0	1.1522
successive	0	0	0
total	9.4461	7.5818	7.8766

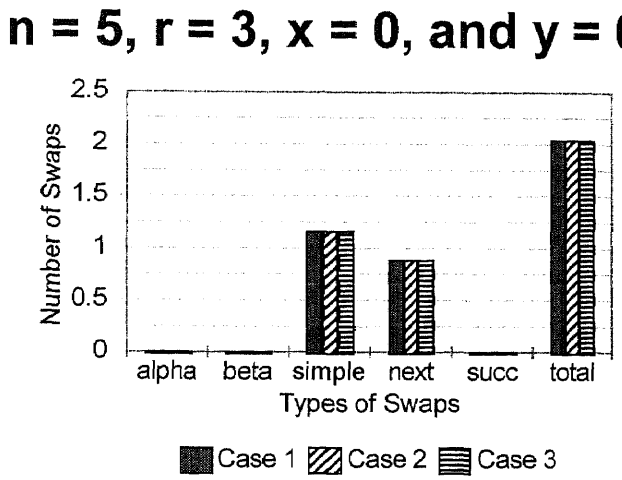

Figure 19 Number of Swaps in Each Case

Table 17
Number of Swaps in Each Case for n = 5, r = 3, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	1.1599	1.1599	1.1599
next simple	.8817	.8817	.8817
successive	0	0	0
total	2.0417	2.0417	2.0417

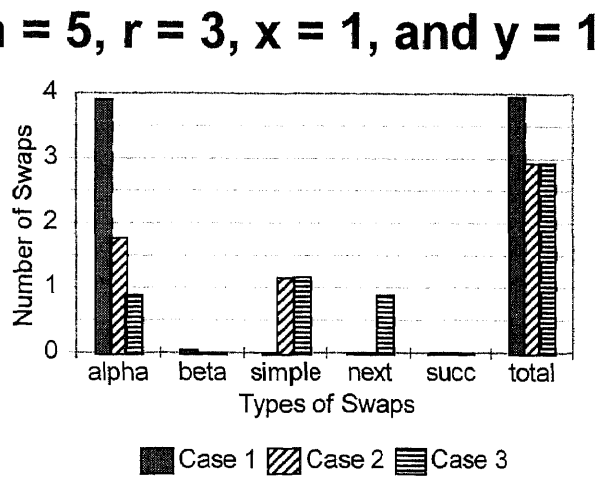

Figure 20 Number of Swaps in Each Case

Table 18
Number of Swaps in Each Case for n = 5, r = 3, x = 1, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	3.9054	1.7677	.882
beta	.041	0	0
simple	.0023	1.1579	1.1596
next simple	.001	0	.882
successive	0	0	0
total	3.9497	2.9257	2.9237

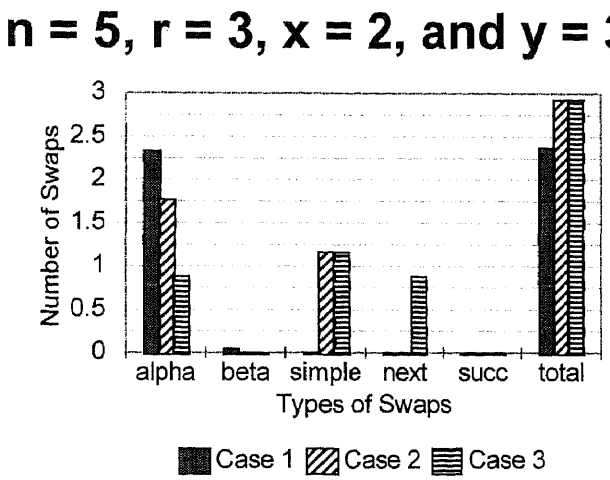

Figure 21 Number of Swaps in Each Case

Table 19
Number of Swaps in Each Case for n = 5, r = 3, x = 2, and y = 3

Type of Swap	Case 1	Case 2	Case 3
alpha	2.3322	1.7677	.882
beta	.043	0	0
simple	0	1.1579	1.1596
next simple	0	0	.882
successive	0	0	0
total	2.3752	2.9257	2.9237

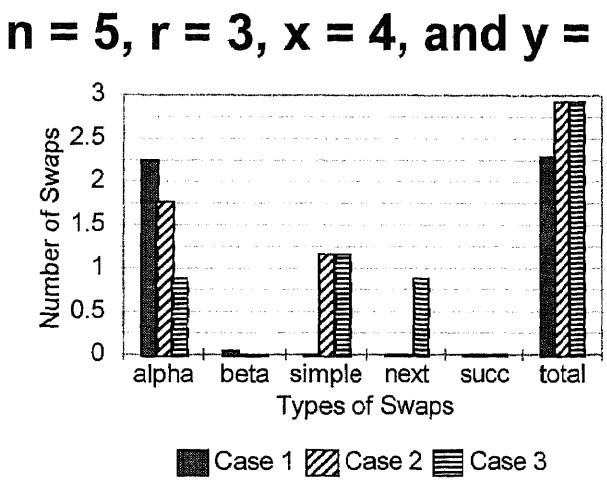

Figure 22 Number of Swaps in Each Case

Table 20
Number of Swaps in Each Case for n = 5, r = 3, x = 4, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	2.2463	1.7677	.882
beta	.043	0	0
simple	0	1.1579	1.1596
next simple	0	0	.882
successive	0	0	0
total	2.2892	2.9257	2.9237

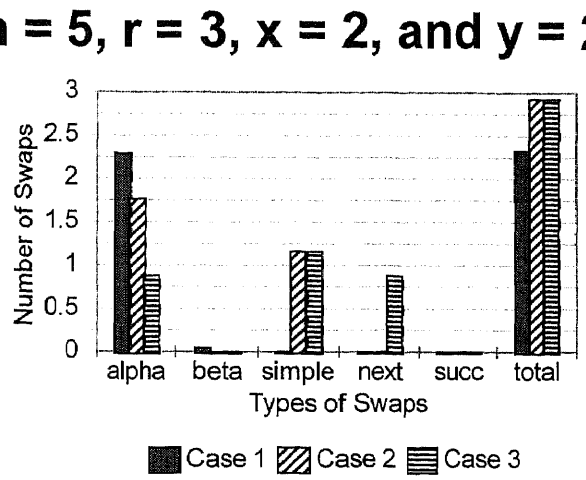

Figure 23 Number of Swaps in Each Case

Table 21
Number of Swaps in Each Case for n = 5, r = 3, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	2.2892	1.7677	.882
beta	.043	0	0
simple	0	1.1579	1.1596
next simple	0	0	.882
successive	0	0	0
total	2.3322	2.9257	2.9237

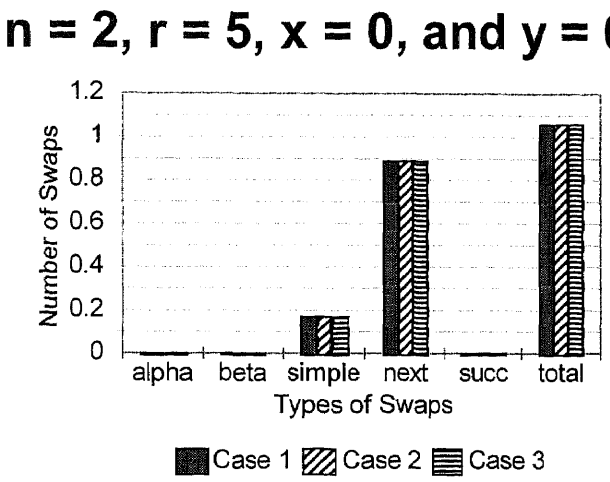

Figure 24 Number of Swaps in Each Case

Table 22
Number of Swaps in Each Case for n = 2, r = 5, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	.1699	.1699	.1699
next simple	.887	.887	.887
successive	0	0	0
total	1.057	1.057	1.057

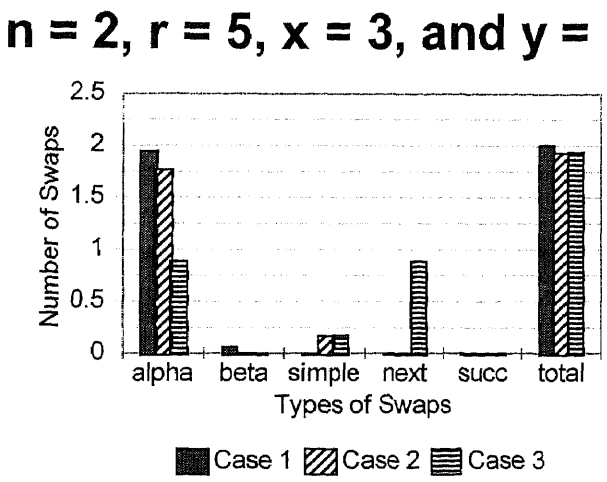

Figure 25 Number of Swaps in Each Case

Table 23
Number of Swaps In Each Case For n = 2, r = 5, x = 3, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	1.946	1.7661	47
beta	.0613	.0007	.0023
simple	0	.1693	.1699
next simple	0	0	.887
successive	0	0	0
total	2.0073	1.936	1.944

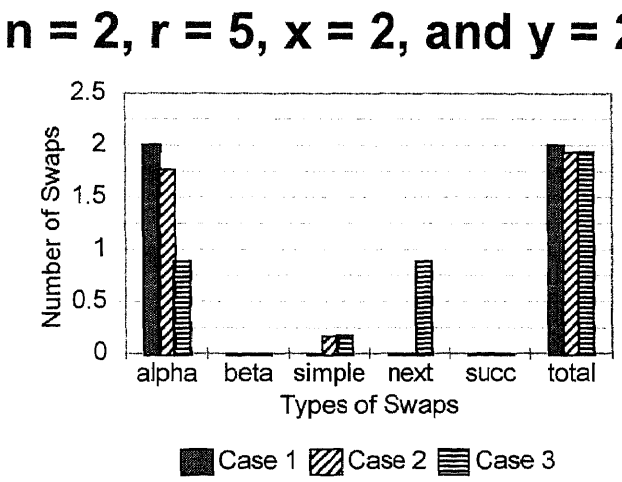

Figure 26 Number of Swaps in Each Case

Table 24
Number of Swaps In Each Case For n = 2, r = 5, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	2.0073	1.7667	.887
beta	0	0	0
simple	0	.1693	.1699
next simple	0	0	.887
successive	0	0	0
total	2.0073	1.936	1.944

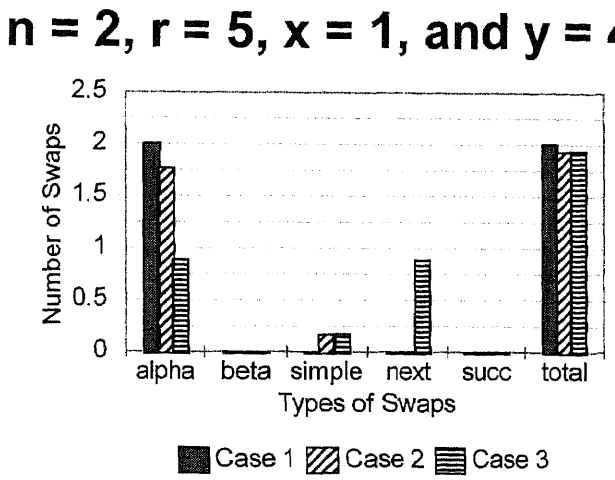

Figure 27 Number of Swaps in Each Case

Table 25
Number of Swaps In Each Case For n = 2, r = 5, x = 1, and y = 4

Type of Swap	Case 1	Case 2	Case 3
alpha	2.0073	1.7667	.887
beta	0	0	0
simple	0	.1693	.1699
next simple	0	0	.887
successive	0	0	0
total	2.0073	1.936	1.944

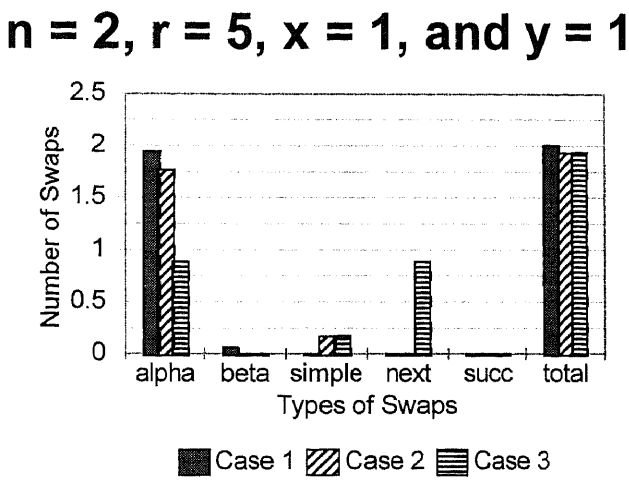

Figure 28 Number of Swaps in Each Case

Table 26
Number of Swaps In Each Case For n = 2, r = 5, x = 1, and y =1

Type of Swap	Case 1	Case 2	Case 3
alpha	1.946	1.7661	.8847
beta	.0613	.0007	.0023
simple	0	.1693	.1699
next simple	0	0	.887
successive	0	0	0
total	2.0073	1.936	1.944

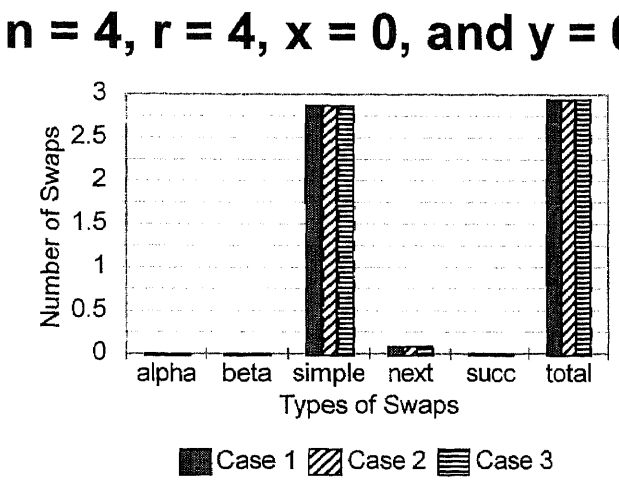

Figure 29 Number of Swaps in Each Case

Table 27
Number of Swaps In Each Case For n = 4, r = 4, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	2.8654	2.8654	2.8654
next simple	.0796	.0796	.0796
successive	0	0	0
total	2.945	2.945	2.945

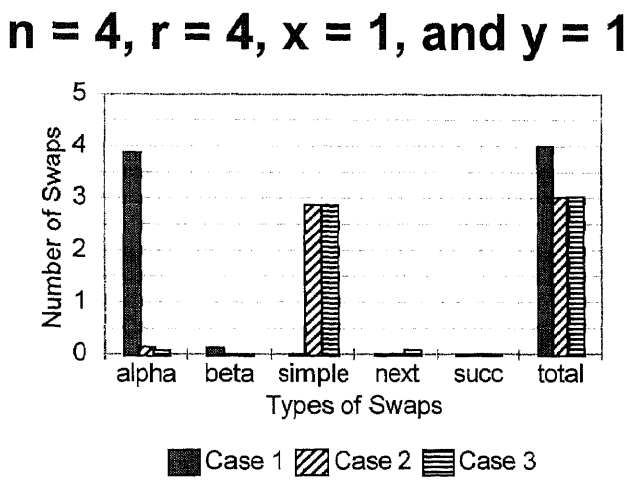

Figure 30 Number of Swaps in Each Case

Table 28
Number of Swaps In Each Case For n = 4, r = 4, x = 1, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	3.8807	.1436	.078
beta	.12	.0033	.003
simple	.001	2.871	2.865
next simple	.0003	0	.0796
successive	0	0	0
total	4.002	3.018	3.0257

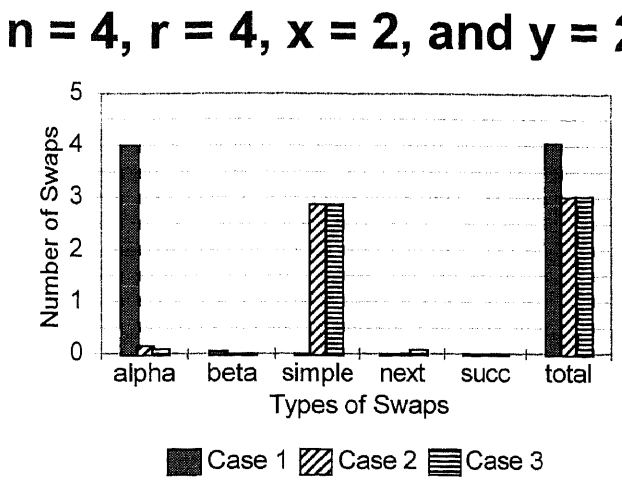

Figure 31 Number of Swaps in Each Case

Table 29
Number of Swaps In Each Case For n = 4, r = 4, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	4.0017	.147	.081
beta	.0526	.0033	0
simple	0	2.871	2.865
next simple	0	0	.0796
successive	0	0	0
total	4.0543	3.0213	3.0257

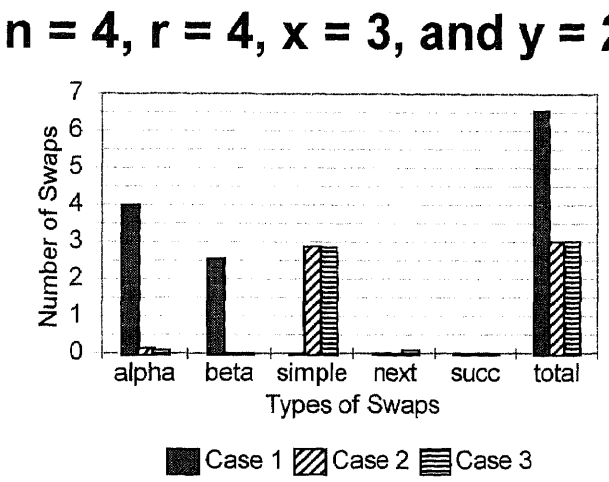

Figure 32 Number of Swaps in Each Case

Table 30
Number of Swaps In Each Case For n = 4, r = 4, x = 3, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	4.0017	.147	.081
beta	2.5528	.0033	0
simple	0	2.871	2.865
next simple	0	0	.0796
successive	0	0	0
total	6.5545	3.0213	3.0257

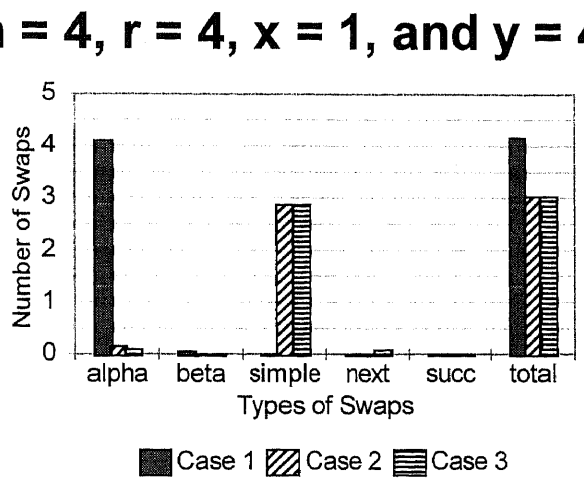

Figure 33 Number of Swaps in Each Case

Table 31
Number of Swaps In Each Case For n = 4, r = 4, x = 1, and y = 4

Type of Swap	Case 1	Case 2	Case 3
alpha	4.1063	.1536	.081
beta	.0523	.0033	0
simple	.0003	2.871	2.865
next simple	0	0	.0796
successive	0	0	0
total	4.1589	3.028	3.0257

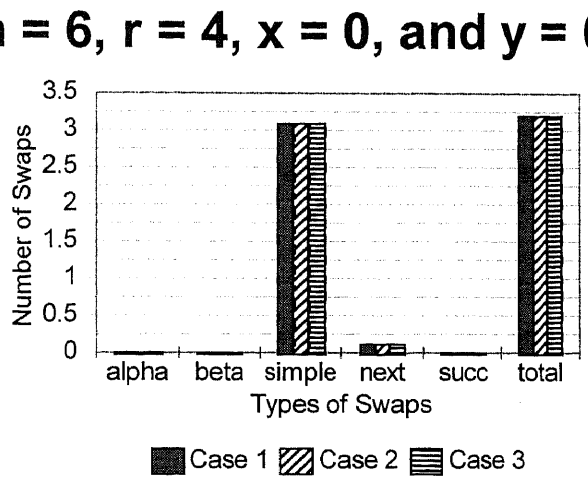

Figure 34 Number of Swaps in Each Case

Table 32
Number of Swaps In Each Case For n = 6, r = 4, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	3.081	3.081	3.081
next simple	.1176	.1176	.1176
successive	0	0	0
total	3.1986	3.1986	3.1986

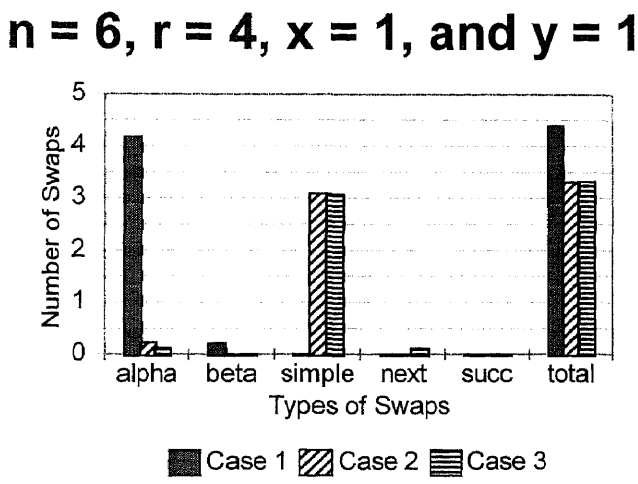

Figure 35 Number of Swaps in Each Case

Table 33
Number of Swaps In Each Case For n = 6, r = 4, x = 1, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	4.1769	.2183	.1203
beta	.2129	.0027	.0047
simple	.0067	3.088	3.08
next simple	.001	0	.118
successive	.0013	0	0
total	4.3989	3.3089	3.3229

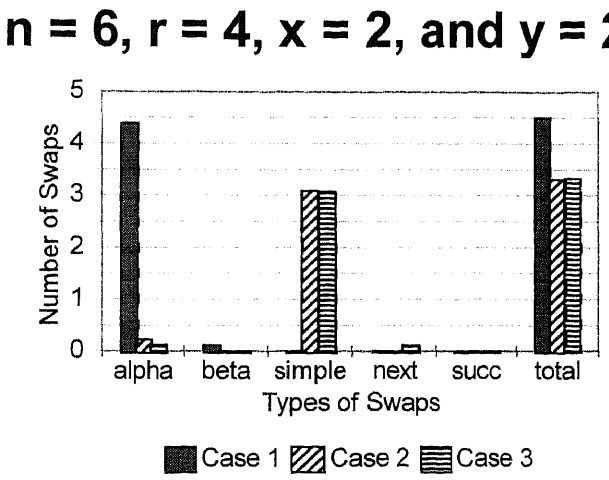

Figure 36 Number of Swaps in Each Case

Table 34
Number of Swaps In Each Case For n = 6, r = 4, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	4.3945	.2209	.1253
beta	.107	.0027	0
simple	0	3.088	3.08
next simple	0	0	.118
successive	0	0	0
total	4.5015	3.3116	3.3232

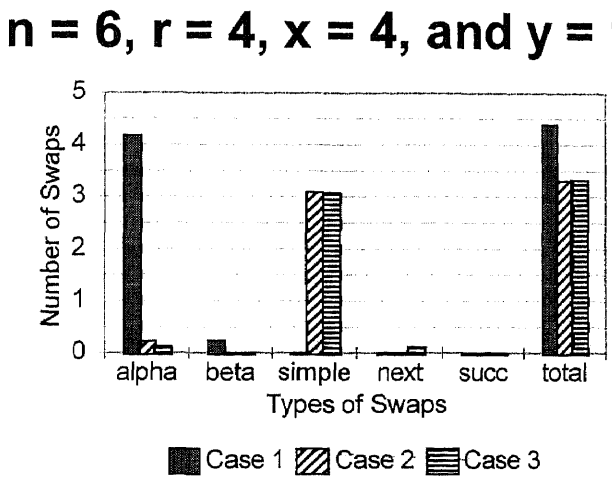

Figure 37 Number of Swaps in Each Case

Table 35
Number of Swaps In Each Case For n = 6, r = 4, x = 4, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	4.1769	.2183	.1203
beta	.2219	.0027	.0047
simple	0	3.088	3.08
next simple	0	0	.118
successive	0	0	0
total	4.3989	3.3089	3.3229

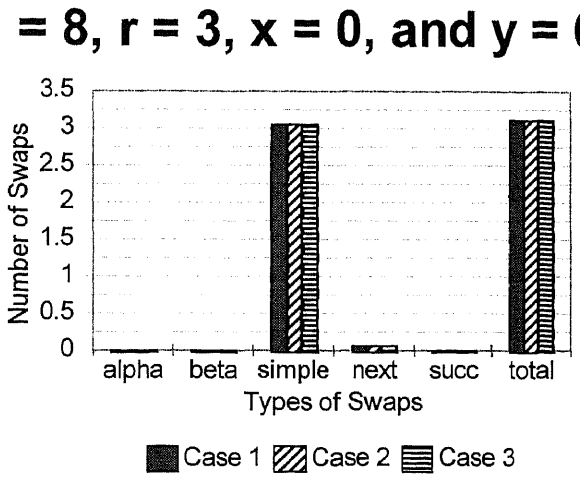

Figure 38 Number of Swaps in Each Case

Table 36
Number of Swaps In Each Case For n = 8, r = 3, x = 0, and y = 0

Type of Swap	Case 1	Case 2	Case 3
alpha	0	0	0
beta	0	0	0
simple	3.0506	3.0506	3.0506
next simple	.061	.061	.061
successive	0	0	0
total	3.1116	3.1116	3.1116

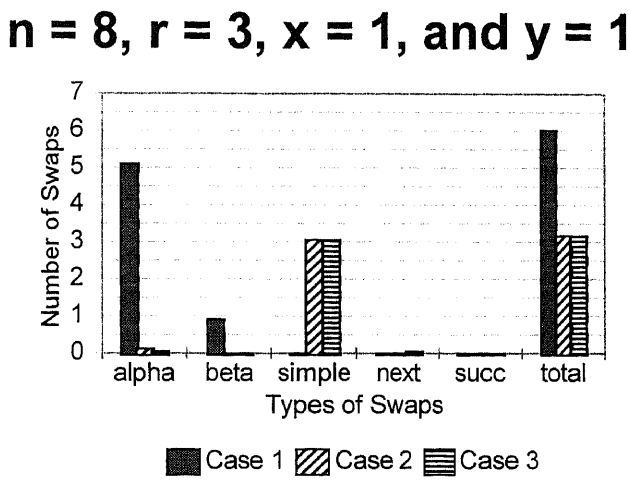

Figure 39 Number of Swaps in Each Case

Table 37
Number of Swaps In Each Case For n = 8, r = 3, x = 1, and y = 1

Type of Swap	Case 1	Case 2	Case 3
alpha	5.102	.1236	.0616
beta	.911	0	0
simple	.001	3.0546	3.05
next simple	.0003	0	.0616
successive	.0003	0	0
total	6.0147	3.1783	3.1733

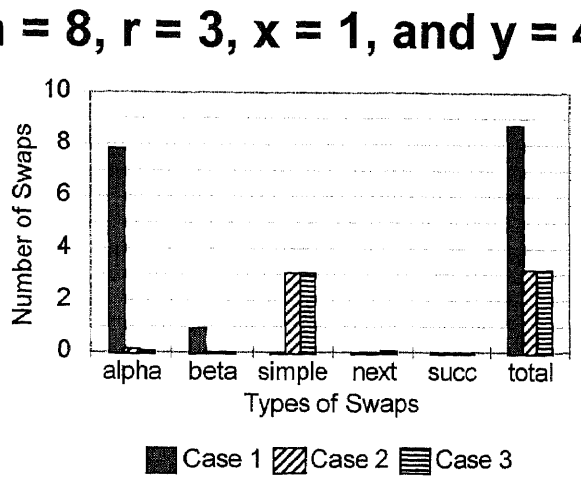

Figure 40 Number of Swaps in Each Case

Table 38
Number of Swaps In Each Case For n = 8, r = 3, x = 1, and y = 4

Type of Swap	Case 1	Case 2	Case 3
alpha	7.8401	.1236	.0616
beta	.911	0	0
simple	.001	3.0546	3.05
next simple	.0003	0	.0616
successive	.0003	0	0
total	8.7527	3.1783	3.1733

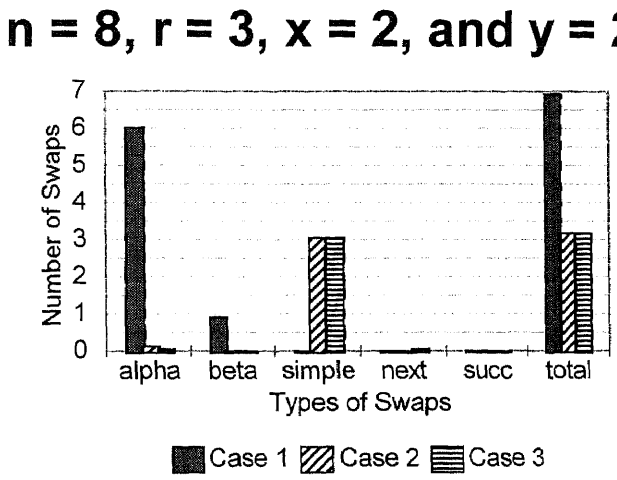

Figure 41 Number of Swaps in Each Case

Table 39
Number of Swaps In Each Case For n = 8, r = 3, x = 2, and y = 2

Type of Swap	Case 1	Case 2	Case 3
alpha	6.0147	.1236	.0616
beta	.9127	0	0
simple	0	3.0546	3.05
next simple	0	0	.0616
successive	0	0	0
total	6.9274	3.1783	3.1733

CHAPTER 5

CONCLUSION

Previous research by Lee and Carpinelli has shown that using the extra switches in a fault tolerant Clos (FTC) network under no-fault conditions can significantly speed up the network's performance. They also introduced an algorithm to be used when extending the network to utilize these switches (Lee, Hwang, and Carpinelli 1996, 1572-3).

However, by using the C language to program the algorithm and run a network simulation, the order of the swaps in this algorithm, given as wild swap, simple swap, next simple swap, and successive swap, was shown to be the least efficient of three possible cases. Re-arranging these swaps with the simple swap first, followed by the next simple and successive swaps, regardless of the placement of the wild swap, will noticeably decrease the number of swaps needed to decompose the matrix which assigns the switch settings in the FTC network.

APPENDIX

matrix.c

```
/* This program will input n, r, x, and y and then generate 500 matrices with
   that data set, to be used in the Clos swapping program. */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define NUMBER 500

FILE *matrices;

int rand_range (int, int);

int
main (void)
{
  int n, r, x, y, main1, main2, maintemp, maincount;
  matrices = fopen ("matrices", "a");
  printf ("What is n\?\n");
  scanf (" %d", &n);
  fprintf (matrices, "%d ", n);
  printf ("What is r\?\n");
  scanf (" %d", &r);
  fprintf (matrices, "%d ", r);
  printf ("What is x\?\n");
  scanf (" %d", &x);
  fprintf (matrices, "%d ", x);
  printf ("What is y\?\n");
  scanf (" %d", &y);
  fprintf (matrices, "%d\n", y);

  for (maincount = 0; maincount < NUMBER; maincount++)
    {
      for (main1 = 1; main1 <= r; main1++)
          {
            for (main2 = 1; main2 <= n; main2++)
              {
                /*
                 * Pick a random number between 0 and n for each
                 * element
```

```
            */
            maintemp = rand_range (0, n);
            fprintf (matrices, "%d ", maintemp);
          }
        fprintf (matrices, "\n");
      }
    fprintf (matrices, "\n");
  }
}

/* choose a random integer between min and max inclusive */
int
rand_range (int min, int max)
{
 const int range = max - min + 1;
 int my_max_rand;
 int r;

 my_max_rand = (RAND_MAX / range) * range;

 /* ignore numbers past my_max_rand; the remaining numbers
    are then evenly divisible into range subsets */
 while ((r = rand ()) >= my_max_rand)
  continue;

 return (r % range) + min;
```

clos.c

```
/* This program inputs sample specification matrices and performs the original order of
swaps (Case 1 of the Matrix Decomposition Algorithm) on them. */

#include <stdio.h>
#include <stdlib.h>

FILE *output;

int S[50][50];
int Zero[70][70];
int Two[70][70];
int JE[70][70][70];
int r, n, x, y;
int alpha = 0, beta = 0, simp = 0, nextsimp = 0, succ = 0;

void createS (void);
void showS (void);
void findZeroTwo (void);
void zeroZeroTwo (void);
void printZero (void);
void printTwo (void);
void zeroJE (void);
void findJE (void);
void printJE (void);
void swapping (void);
int wildswap (int, int);
void simple (int, int, int, int);
void successive (int, int, int, int);
void successb (int, int, int, int, int);
void removefrom (int, int, int);
void addto (int, int, int);
void addtwo (int, int);
void addzero (int, int);
void removetwo (int, int);
void removezero (int, int);

int
main (void)
{
  output = fopen ("output", "a");
  printf ("What is n\?\n");
```

```
 scanf (" %d", &n);
 printf ("What is r\?\n");
 scanf (" %d", &r);
 printf ("What is x\?\n");
 scanf (" %d", &x);
 printf ("What is y\?\n");
 scanf (" %d", &y);

/* n is the number of columns in S (without the extra switches), while r is
   the number of rows. x is the number of extra switches in the outer stages,
   and y is the number of extra center switches. */

 createS ();
 showS ();
 zeroZeroTwo ();
 findZeroTwo ();
 zeroJE ();
 findJE ();
 printZero ();
 printf ("\n");
 printTwo ();
 printf ("\n");
 printJE ();
 printf ("Zero[2][1] is %d, Zero[2][2] is %d, Zero[2][3] is %d\n", Zero[2][1],
Zero[2][2], Zero[2][3]);
 swapping ();
 fprintf (output, "%d alpha swaps made\n", alpha);
 fprintf (output, "%d beta swaps made\n", beta);
 fprintf (output, "%d simple swaps made\n", simp);
 fprintf (output, "%d next simple swaps made\n", nextsimp);
 fprintf (output, "%d successive swaps made\n", succ);
 fprintf (output, "\n");
 return (0);
}

void
createS (void)
{
 int i, j, input;
 printf ("Enter the values of the matrix, beginning with the upper right corner and
continuing across each row.\n");
 for (i = 0; i < r; i++)
```

```
    for (j = 0; j < n; j++)
        {
         scanf (" %d", &input);
         S[i][j] = input;
        }
   }
/* Now need to add the rows of alphas and columns of betas. */
 for (i = r; i < r + y; i++)
   {
    for (j = 0; j < n + x; j++)
        S[i][j] = 'A';
   }
 for (j = n; j < n + x; j++)
   {
    for (i = 0; i < r; i++)
        S[i][j] = 'B';
    while (i < r + y)              /* adds 'x' to intersection of A's and B's */
        {
         S[i][j] = 'x';
         i++;
        }
   }
}

void
showS (void)
{
 int i, j;
 printf ("This is S:\n");
 for (i = 0; i < r + y; i++)
   {
    for (j = 0; j < n + x; j++)
        printf ("%4d ", S[i][j]);
    printf ("\n");
   }
}

void
findZeroTwo (void)
{
 int e, i, j, counter, k, m;
 for (e = 0; e < r; e++)
```

```
    k = 0;                          /* index in Zero[e][k] array */
    m = 0;                          /* index in Two[e][m] array */
    for (j = 0; j < n + x; j++)     /* which column you're in */
        {
         counter = 0;               /* keeping track of how many e's in column j */
         for (i = 0; i < r; i++)    /* which row you're in */
          {
           if (S[i][j] == e)
               counter++;           /* Found an e in column j! Increment! */
          }
         if (counter == 0)
          {
           Zero[e][k] = j;          /* Add e to Zero[][], as no e found. */
           k++;
          }
         if (counter >= 2)
          {
           Two[e][m] = j;           /* Add e to Two[][]; at least two e's found. */
           m++;
          }
        }
  }
* Doing again for e = A and e = B. */
 e = 'A';
 k = 0;                             /* index in Zero[e][k] array */
 m = 0;                             /* index in Two[e][m] array */
 for (j = 0; j < n + x; j++)        /* which column you're in */
  {
    counter = 0;                    /* keeping track of how many e's in column j */
    for (i = 0; i < r + y - 1; i++)         /* which row you're in */
        {
         if (S[i][j] == e)
           counter++;               /* Found an e in column j! Increment! */
        }
    if (counter == 0)
        {
         Zero[e][k] = j;            /* Add e to Zero[][], as no e found. */
         k++;
        }
    if (counter >= 2)
        {
         Two[e][m] = j;             /* Add e to Two[][]; at least two e's found. */
         m++;
```

```
        }
    }
  e = 'B';
  k = 0;                        /* index in Zero[e][k] array */
  m = 0;                        /* index in Two[e][m] array */
  for (j = 0; j < n + x; j++)   /* which column you're in */
    {
     counter = 0;               /* keeping track of how many e's in column j */
     for (i = 0; i < r + y - 1; i++)      /* which row you're in */
          {
           if (S[i][j] == e)
             counter++;         /* Found an e in column j! Increment! */
          }
     if (counter == 0)
          {
           Zero[e][k] = j;      /* Add e to Zero[][], as no e found. */
           k++;
          }
     if (counter >= 2)
          {
           Two[e][m] = j;       /* Add e to Two[][]; at least two e's found. */
           m++;
          }
    }
}

void
findJE (void)
{
 int j = 0, e = 0, i = 0, counter, counter2;
 for (j = 0; j < n + x; j++)
   {
    for (e = 0; e < r; e++)
         {
          counter = 0;
          for (i = 0; i < r; i++)  /* going across rows looking for a match */
            {
             if (S[i][j] == e)
                  {
                   JE[j][e][counter] = i;
                   counter++;
```

```
        }
/* Now check extra rows for A and B... */
    counter = 0;
    counter2 = 0;
    for (i = 0; i < r + y; i++)
        {
         if (S[i][j] == 'A')
           {
            JE[j]['A'][counter] = i;
            counter++;
           }
         if (S[i][j] == 'B')
           {
            JE[j]['B'][counter2] = i;
            counter2++;

           }
        }
   }
}

/* This function initializes the Zero[][] and Two[] arrays to all -1's. */

void
zeroZeroTwo (void)
{
 int i, j;
 for (i = 0; i < 70; i++)
   {
    for (j = 0; j < 50; j++)
        {
         Zero[i][j] = -1;
         Two[i][j] = -1;
        }
   }
}

/* This function does the same for the JE[][][] array. */

void
zeroJE (void)
{
 int i, j, k;
```

```
 for (i = 0; i < 70; i++)
   {
    for (j = 0; j < 70; j++)
        {
         for (k = 0; k < 50; k++)
           JE[i][j][k] = -1;
        }
   }
}

void
printZero (void)
{
 int e, k;
 for (e = 0; e < r; e++)
   {
    printf ("0(%d) is: ", e);
    for (k = 0; k < 50; k++)
        {
         if (Zero[e][k] != -1)  /* Don't want to print the -1 entries. */
           printf ("%d ", Zero[e][k]);
        }
    printf ("\n");
   }
/* Specially print the entries for A and B: */
 printf ("0(A) is: ");
 for (k = 0; k < 50; k++)
   {
    if (Zero['A'][k] != -1)       /* Don't want to print the -1 entries. */
        printf ("%d ", Zero['A'][k]);
   }
 printf ("\n");
 printf ("0(B) is: ");
 for (k = 0; k < 50; k++)
   {
    if (Zero['B'][k] != -1)       /* Don't want to print the -1 entries. */
        printf ("%d ", Zero['B'][k]);
   }
 printf ("\n");
}

void
printTwo (void)
```

```
{
 int e, m;
 for (e = 0; e < r; e++)
   {
    printf ("2(%d) is: ", e);
    for (m = 0; m < 50; m++)
        {
         if (Two[e][m] != -1) /* Don't want to print the -1 entries. */
           printf ("%d ", Two[e][m]);
        }
    printf ("\n");
   }
/* Specially print the entries for A and B: */
 printf ("2(A) is: ");
 e = 'A';
 for (m = 0; m < 50; m++)
   {
    if (Two[e][m] != -1)        /* Don't want to print the -1 entries. */
        printf ("%d", Two[e][m]);
   }
 printf ("\n");
 printf ("2(B) is: ");
 e = 'B';
 for (m = 0; m < 50; m++)
   {
    if (Two[e][m] != -1)        /* Don't want to print the -1 entries. */
        printf ("%d", Two[e][m]);
   }
 printf ("\n");
}

void
printJE (void)
{
 int j, e, c;
 for (j = 0; j < n + x; j++)
   {
    for (e = 0; e < r; e++)
        {
         printf ("(%d,%d) is: ", j, e);
         for (c = 0; c < r; c++)
           {
            if (JE[j][e][c] != -1)        /* Don't print -1 entries. */
```

```
                printf ("%d ", JE[j][e][c]);
            }
          printf ("\n");
        }
/* Specially print the cases where e = A or B. */
    e = 'A';
    printf ("(%d,A) is: ", j);
    for (c = 0; c < r; c++)
        {
         if (JE[j][e][c] != -1)
           printf ("%d ", JE[j][e][c]);
        }
    printf ("\n");
    e = 'B';
    printf ("(%d,B) is: ", j);
    for (c = 0; c < r; c++)
        {
         if (JE[j][e][c] != -1)
           printf ("%d ", JE[j][e][c]);
        }
    printf ("\n");
   }
}

void
removefrom (int a, int b, int c)
{
 int counter, keep;
 for (counter = 0; counter < 50; counter++)
   {
    if (JE[b][c][counter] == a)
        {
         keep = counter;
         counter = 50;
        }
   }
 for (counter = keep; counter < 49; counter++)
  JE[b][c][counter] = JE[b][c][counter + 1];
 JE[b][c][49] = -1;
}

void
addto (int a, int b, int c)
```

```
{
 int counter, counter2, temp;
 counter = 0;
 for (counter2 = 0; counter2 < 50; counter2++)
   {
    while (JE[b][c][counter] < a && JE[b][c][counter] != -1)
        counter++;
   }
 temp = counter;
 for (counter = 49; counter > temp; counter--)
  JE[b][c][counter] = JE[b][c][counter - 1];
 JE[b][c][temp] = a;
}

void
addtwo (int a, int b)
{
 int counter, temp;
 for (counter = 0; counter < 50; counter++)
   {
    if (Two[b][counter] > a)
        {
         temp = counter;
         counter = 50;                  /* end loop */
        }
   }
 for (counter = 49; counter > temp; counter--)
  Two[b][counter] = Two[b][counter - 1];
 Two[b][temp] = a;
}

void
addzero (int a, int b)
{
 int counter, temp;
 for (counter = 0; counter < 50; counter++)
   {
    if (Zero[b][counter] > a)
        {
         temp = counter;
         counter = 50;                  /* end loop */
```

```
 for (counter = 49; counter > temp; counter--)
   Zero[b][counter] = Zero[b][counter - 1];
 Zero[b][temp] = a;
}

void
removetwo (int a, int b)
{
 int counter, temp;
 for (counter = 0; counter < 50; counter++)
   {
    if (Two[b][counter] == a)
        {
         temp = counter;
         counter = 50;
        }
   }
 for (counter = temp; counter < 50; counter++)
   Two[b][counter] = Two[b][counter + 1];
 Two[b][49] = -1;
}

void
removezero (int a, int b)
{
 int counter, temp;
 for (counter = 0; counter < 50; counter++)
   {
    if (Zero[b][counter] == a)
        {
         temp = counter;
         counter = 50;
        }
   }
 for (counter = temp; counter < 50; counter++)
   Zero[b][counter] = Zero[b][counter + 1];
 Zero[b][49] = -1;
}

void
swapping (void)
{
 int i, iprime, j, k, e = 0, flag = 0;
```

```
while (e < r)
 {
  if (Two[e][0] != -1)          /* 2(e) isn't empty */
      {
       j = Two[e][0];
       k = Zero[e][0];
       flag = wildswap (e, j);
       printf ("Back in swapping and flag is %d\n", flag);
       if (flag == 0)
         {
          i = JE[j][e][0];      /* i is first element of (j.e) */
          if ((e < S[i][k] || S[i][k] == 'B') && i != -1)
              {
               simp++;
               simple (e, i, j, k);       /* Do simple swap */
              }
          else if (e > S[i][k])
              {
               iprime = JE[j][e][1];
               if (e < S[iprime][k] || S[iprime][k] == 'B')
                 {
                  /* Making sure iprime actually exists */
                  if (iprime != -1)
                      {
                       nextsimp++;
                       /* Do next simple swap */
                       simple (e, iprime, j, k);
                      }
                 }
               else if (e > S[iprime][k] || S[iprime][k] == 'A')
                 successive (e, i, j, k);          /* Do successive swap */
              }
         }
        e = 0;
       }
    else
       e++;
  }
}

int
wildswap (int e, int j)
```

```
int i, u, temp, counter, counter2, v, temp2, flag = 0;
printf ("At beginning of wild loop, e is %d\n", e);
i = JE[j][e][0];
if (i != -1)
  {
   for (counter = 0; counter < 50; counter++)
       {
        u = JE[j]['A'][counter];
        if (u == -1)
          counter = 50;          /* end loop, as have reached the end of (j,A) */
        else if (u < r)
          u = u;                 /* Do nothing. Want to keep going through (j,A)
                                    to find a u >= r. */
        else if (u >= r)
          {
           flag = 1;             /* doing a wild swap */
           alpha++;
           printf ("Doing an alpha swap\n");
           temp = S[u][j];
           S[u][j] = S[i][j];    /* swap */
           S[i][j] = temp;
           /* removing i from (j, e) */
           removefrom (i, j, e);
           /* removing u from (j, A) */
           removefrom (u, j, 'A');
           /* Adding i to (j, A). */
           addto (i, j, 'A');
           /* Now adding u to (j,e). */
           addto (u, j, e);
        /* |(j,e)|* = 1 */
           if ((JE[j][e][0] < r && JE[j][e][1] >= r) || JE[j][e][1] == -1)
               {
                printf ("Removing in alpha\n");
                removetwo (j, e);
                showS ();
                printZero ();
                printTwo ();
                printJE ();
                return (flag);
               }
           showS ();
           printZero ();
           printTwo ();
```

```
            printJE ();
          }
        }
      }

* Now want to see if any element i in (j,e) matches an element in (v,B). */
 for (counter = 0; counter < 50; counter++)
   {
    if (JE[j][e][counter] < r)
        {
         i = JE[j][e][counter];
         for (v = n; v < n + x; v++)
           {
            if (JE[v]['B'][0] != -1)
                {
                 for (counter2 = 0; counter2 < 50; counter2++)
                   {
                    if (JE[v]['B'][counter2] == i && i != -1)
                        {
                         flag = 1;       /* doing a wild swap */
                         beta++;
                         printf ("Doing a beta swap with i = %d, j = %d\n", i, j);
                         temp2 = S[i][v];
                         S[i][v] = S[i][j];
                         S[i][j] = temp2;
                         /* removing i from (j,e) */
                         removefrom (i, j, e);
                         /* removing i from (v,B) */
                         removefrom (i, v, 'B');
                         /* Adding i to (j,B) */
                         addto (i, j, 'B');
                         /* Adding i to (v,e) */
                         addto (i, v, e);
                         if ((JE[j][e][0] < r && JE[j][e][1] >= r) || JE[j][e][1] == -1)
        /* |(j,e)|* = 1 */
                           {
                            printf ("Removing in beta\n");
                            removetwo (j, e);
                           }

                         if (JE[j]['B'][x - 1] != -1 && JE[j]['B'][x] == -1)
                          removezero (j, 'B');
                         if (JE[j]['B'][x] != -1 && JE[j]['B'][x + 1] == -1)
```

```
                    addtwo (j, 'B');
                  if (JE[v]['B'][x - 2] != -1 && JE[v]['B'][x - 1] == -1)
                    addzero (v, 'B');
                  if (JE[v]['B'][x - 1] != -1 && JE[v]['B'][x] == -1)
                    removetwo (v, 'B');
                  removezero (v, e);
                  showS ();
                  printZero ();
                  printTwo ();
                  printJE ();
                  return (flag);
                }
              }
            }
          }
        }
      }
     return (flag);
}

void
simple (int e, int i, int j, int k)
{
  int temp, eprime;
/***** Simple Swap *****/
/* Performing the swap */
printf("In simple swap with e = %d, eprime = %d, i = %d, j = %d, and k =
%d\n",e,S[i][k],i,j,k);
  eprime = S[i][k];
  temp = S[i][j];
  S[i][j] = S[i][k];
  S[i][k] = temp;
/* Removing i from (j,e) */
  removefrom (i, j, e);
/* Removing i from (k,e') */
  removefrom (i, k, eprime);
  addto (i, j, eprime);
  addto (i, k, e);
  if ((JE[j][e][0] < r && JE[j][e][1] >= r) || JE[j][e][1] == -1)  /* |(j,e)|* = 1 */
    removetwo (j, e);
  if (eprime == 66)
    {
printf("In eprime == B\n");
```

```
    if (JE[j]['B'][x - 1] != -1 && JE[j]['B'][x] == -1)
        removezero (j, 'B');
    if (JE[j]['B'][x] != -1 && JE[j]['B'][x + 1] == -1)
        addtwo (j, 'B');
    if (JE[k]['B'][x - 2] != -1 && JE[k]['B'][x - 1] == -1)
        addzero (k, 'B');
    if (JE[k]['B'][x - 1] != -1 && JE[k]['B'][x] == -1)
        removetwo (k, 'B');
  }
 else
  {
printf("In else\n");
    if ((JE[j][eprime][0] < r && JE[j][eprime][1] >= r) || JE[j][eprime][1] == -1)  /*
|(j,e')|* = 1 */
        removezero (j, eprime);
    if ((JE[j][eprime][1] < r && JE[j][eprime][2] >= r) || JE[j][eprime][2] == -1)  /*
|(j,e')|* = 2 */
        addtwo (j, eprime);
    if (JE[k][eprime][0] >= r || JE[k][eprime][0] == -1)  /* |(k,e')|* = 0 */
        addzero (k, eprime);
    if ((JE[k][eprime][0] < r && JE[k][eprime][1] >= r) || JE[k][eprime][1] == -1) /*
|(k,e')| = 1 */
        removetwo (k, eprime);
  }
 removezero (k, e);
 showS ();
 printZero ();
 printTwo ();
 printJE ();
 swapping ();
}

void
successive (int e, int i, int j, int k)
{
 int u;
 if (i != -1)
  {
   succ++;
   u = e;
   removezero (k, u);          /* Remove k from 0(u) */
   printf ("In succ swap with e = %d, j = %d, k = %d, i = %d, u = %d, and v = %d\n",
e, j, k, i, u, S[i][k]);
```

```
    /* |(j,u)|* = 2 */
    if ((JE[j][u][1] < r && JE[j][u][2] >= r) || JE[j][u][2] == -1)
        removetwo (j, u);
    successb (e, i, j, k, u);
   }
}

void
successb (int e, int i, int j, int k, int u)
{
 int v, temp;
 v = S[i][k];
 temp = S[i][j];
 S[i][j] = S[i][k];
 S[i][k] = temp;
/* Remove i from (j, u) */
 removefrom (i, j, u);
/* Remove i from (k,v) */
 removefrom (i, k, v);
/* Add i to (j,v) */
 addto (i, j, v);
/* Add i to (k,u) */
 addto (i, k, u);
 if (v == 'B' || e < v || v == 'B' || (JE[j][v][0] < r && JE[j][v][1] >= r) || JE[j][v][1] == -1)
   {
    if (v == 'B')
        {
         /* |(k,B)| = x-1  */
         if (JE[k]['B'][x - 2] != -1 && JE[k]['B'][x - 1] == -1)
           addzero (k, 'B');
         if (JE[j]['B'][x - 1] != -1 && JE[j]['B'][x] == -1)     /* |(j,B) = x */
           {
            removetwo (k, 'B');
            removezero (j, 'B');
           }
         if (JE[j]['B'][x] != -1 && JE[j]['B'][x + 1] == -1)    /* |(j,B) = x+1 */
           addtwo (j, 'B');
        }
    else
        {
         if (JE[k][v][0] >= r || JE[k][v][0] == -1)
           addzero (k, v);
         if ((JE[k][v][0] < r && JE[k][v][1] >= r) || JE[k][v][1] == -1)
```

```
            removetwo (k, v);
          if ((JE[j][v][0] < r && JE[j][v][1] >= r) || JE[j][v][1] == -1)
            removezero (j, v);
          if ((JE[j][v][1] < r && JE[j][v][2] >= r) || JE[j][v][2] == -1)
            addtwo (j, v);
        }
      showS ();
      printZero ();
      printTwo ();
      printJE ();
      swapping ();                /* Go to Step 1 */
    }
  showS ();
  printZero ();
  printTwo ();
  printJE ();

  if (e > v || v == 'A')
    {
      u = v;
/* Take i from (j,v) */
      i = JE[j][v][0];
      successb (e, i, j, k, u);
```

average.c

```
/* This program will take the list of outputs and find the average number
   of each type of swap, as well as the average number of total swaps for
   each data set. */

#include <stdio.h>

int main (void)
{
double alpha = 0.0, beta = 0.0, simple = 0.0, next = 0.0, succ = 0.0, total = 0.0, counter =
0.0;
int i;
double alphav, betav, simpav, nextav, succav, totav;
int temp1, temp2, temp3, temp4, temp5, temp6;

for(i = 0; i <= 3000; i++)
{
scanf(" %d", &temp1);
scanf(" %d", &temp2);
scanf(" %d", &temp3);
scanf(" %d", &temp4);
scanf(" %d", &temp5);
if(temp1 != 1000)
{
counter++;
alpha = alpha + temp1;
beta = beta + temp2;
simple = simple + temp3;
next = next + temp4;
succ = succ + temp5;
temp6 = temp1 + temp2 + temp3 + temp4 + temp5;
total = total + temp6;
}
else
i = 3000;
}

alphav = alpha/counter;
betav = beta/counter;
simpav = simple/counter;
nextav = next/counter;
succav = succ/counter;
```

```
totav = total/counter;

printf("Av. number of alpha swaps: %.4lf\n", alphav);
printf("Av. number of beta swaps: %.4lf\n", betav);
printf("Av. number of simple swaps: %.4lf\n", simpav);
printf("Av. number of next simple swaps: %.4lf\n", nextav);
printf("Av. number of successive swaps: %.4lf\n", succav);
printf("Av. number of total swaps: %.4lf\n", totav);
```

Swapping Function for Case 2

```
void swapping (void)
{
 int i, iprime, j, k, e = 0, flag = 0, flag2 = 0;
 while (e < r)
  {
   if (Two[e][0] != -1 && Zero[e][0] != -1)          /* 2(e), 0(e) not empty */
       {
        j = Two[e][0];
        k = Zero[e][0];
        i = JE[j][e][0];         /* i is first element of (j.e) */
      flag2 = 0;
      flag = 0;
    if ((e < S[i][k] || S[i][k] == 'B') && i != -1)
          {
           simp++;
           simple (e, i, j, k);  /* Do simple swap */
         flag2 = 1;
          }
   printf("Flag2 is %d\n", flag2);
   if (flag2 == 0)
          {
           flag = wildswap (e, j);
           if (e > S[i][k] && flag == 0)
                {
                 iprime = JE[j][e][1];
                 if ((e < S[iprime][k] || S[iprime][k] == 'B') && iprime != -1)
                   {
                          nextsimp++;
                          /* Do next simple swap */
                          simple (e, iprime, j, k);
                   }
                 else if (e > S[iprime][k] || S[iprime][k] == 'A')
                   successive (e, i, j, k);         /* Do successive swap */
                }
          }
        e = 0;
       }
   else
      e++;
```

Swapping Function for Case 3

```
void swapping (void)
{
 int i, iprime, j, k, e = 0, flag = 0, flag2 = 0;
 while (e < r)
   {
    if (Two[e][0] != -1 && Zero[e][0] != -1)          /* 2(e), 0(e) not empty */
        {
         j = Two[e][0];
         k = Zero[e][0];
         i = JE[j][e][0];          /* i is first element of (j.e) */
       flag = 0;
       flag2 = 0;
         if ((e < S[i][k] || S[i][k] == 'B') && i != -1)
          {
           simp++;
           simple (e, i, j, k);  /* Do simple swap */
           flag2 = 1;
          }
         if (flag2 == 0)
          {
           if (e > S[i][k])
                {
                 iprime = JE[j][e][1];
                 if ((e < S[iprime][k] || S[iprime][k] == 'B') && iprime != -1)
                   {
                          nextsimp++;
                          /* Do next simple swap */
                          simple (e, iprime, j, k);
                   }
                }
           flag = wildswap (e, j);
           if (flag == 0)
                {
           iprime = JE[j][e][1];
                 if ((e > S[iprime][k] || S[iprime][k] == 'A') && iprime != -1)
                   successive (e, i, j, k);          /* Do successive swap */
                }
          }
     e = 0;
         }
    else
```

```
            e++;
        }
    }
```

REFERENCES

1. J.D. Carpinelli and C.B. Wang, "Performance of a New Decomposition Algorithm for Rearrangeable Fault-tolerant Clos Interconnection Networks under Sub-maximal and No-fault Conditions," DIMACS paper, Princeton, NJ, July 1997.

2. C. Clos, "A Study of Non-Blocking Switching Networks," *Bell Systems Technical Journal*, vol. 32, no. 2, pp. 406-424, March 1953.

3. J. Gordon and S. Srikanthan, "Novel Algorithm for Clos-Type Networks," *Electronic Letters*, vol. 26, no. 21, pp. 1772-1774, October 1990.

4. H.Y. Lee, F.K. Hwang, and J.D. Carpinelli, "A New Decomposition Algorithm for Rearrangeable Clos Interconnection Networks," *IEEE Trans. Commun.*, vol. COM-44, no. 11, pp. 1572-1578, November 1996.

5. H. Nassar and J. Carpinelli, "Design and Performance of a Fault Tolerant Clos Network," *Proceedings of the 1995 Conference on Information Sciences and Systems*, Baltimore, MD, pp. 810-815, March 1995.

6. C.B. Wang, "The Simulation of A New Decomposition Algorithm for Rearrangeable Clos Interconnection Networks," Master Project, New Jersey Institute of Technology, Newark, NJ, May 1997.