To the Graduate Council:

I am submitting herewith a thesis written by Gregory S. Cole entitled "Constrained Dynamic Programming Inference of Markov Networks from Finite Sets of Sample Strings." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael G. Thomason, Major Professor

We have read this thesis

and recommend its acceptance:

Jean R. S. Blair

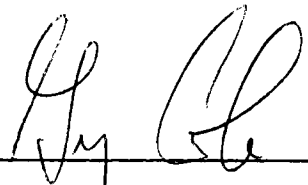
Accepted for the Council:

Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature 

Date 

CONSTRAINED DYNAMIC PROGRAMMING INFERENCE OF MARKOV NETWORKS FROM FINITE SETS OF SAMPLE STRINGS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Gregory S. Cole

June 1988

Copyright © Gregory S. Cole, 1988
All rights reserved

DEDICATION

To my grandmother, Alpha R. Wix, I dedicate this work -- the finished product of three years of effort which I would not have been able to complete without her. For providing me with a home, for feeding me, for making sure I always had something to wear, for keeping me organized, for always supporting me -- no matter how tired and irritable I was -- I am so grateful. And even more importantly, for continuing to instruct me, in both word and manner, in those issues of life that are truly important -- issues of character, of family, of relationship with others -- I am much the wiser and hopefully much better the person for the instruction and for the example given me. She will always remain my best teacher.

ACKNOWLEDGMENTS

There are a number of people -- scholars, family and friends who share credit for completion of my graduate work and of this thesis.

I am particularly indebted to my advisor and major professor, Dr. Michael Thomason. It was honor enough to have him sponsor my thesis work -- it has been one of the great privileges of my life to have worked with him these past three years. His careful scholarship, brilliant intellect, selfless attitude and ready willingness to guide and encourage my research has set my sights higher and made much more valuable my entire graduate experience. I do not know how to adequately express my appreciation for his sponsorship and for his friendship.

I would also like to thank Drs. David Straight and Jean Blair for their interest in this research and for their service on my thesis committee.

I am grateful to Dr. Erik Granum and his family for their interest in and encouragement of this work and for the chance to visit he, his family and his colleagues in Aalborg, Denmark. The experience in Denmark was a high point in my life for which I am very thankful. The experience not only sparked some of the ideas reported in this thesis but has given me enthusiasm for someday pursuing this research further.

I owe a very heavy debt of gratitude to Drs. C.W. Minkel, Tom Collins and Ken Heathington -- all of whom have served at different times as my employer and all of whom have supported and encouraged my

graduate work. Also, to the entire '404 gang' with whom I have worked these past five years, I thank each of you for your support and encouragement.

To my family, I am especially grateful -- for their patience with me, for tolerating my long absences, for their encouragement of my educational goals. To my parents, Lynn and Eleanor Cole, to my brother and sisters, Cindy, Susan and Richie, thank you. Also, to friends -- to Tim, Nancy, Bobby and Larry -- thank you for maintaining friendships which have been much too one-sided due to my time commitments to work and school.

Finally, to the one person who has made it possible for me to live through the pressure and stress imposed by work and school -- and to somehow enjoy it at the same time -- to my girlfriend and best friend, Monica, I thank you.

ABSTRACT

The research reported is an extension of work accomplished by others in automated machine inference of structural models from finite sets of sample strings. The extension has focused on modifying a key algorithm, the dynamic programming technique, to make it less costly and to thus make resulting applications more practical.

Two major modifications of the dynamic programming technique have been designed and implemented. In one, an algorithm for eliminating computation of major portions of the dynamic programming cost matrix was implemented. In the other, a set of redundant calculations was identified and eliminated from the cost matrix computation. The code was implemented on several different computer systems. Results reported here were gathered from code implemented in the C programming language under the VMS operating system on a VAX 8750 computer.

Elimination of redundant calculations yielded time speedups of 900% over a previous implementation. Eliminating computation of portions of the cost matrix yielded time improvements of almost 100% but actually cost more time with the code in which the redundant calculations had been eliminated. When using the application to recognize / classify strings drawn from human chromosome data, the partial matrix code yielded an average 96.5% correct classifications -- an increase over the 94.25% yielded by the full matrix computation code.

The research has provided a more time efficient technique for automated machine inference of structural models using dynamic programming and has shown that object classification using structural models can be improved upon by eliminating from computation portions of the dynamic programming cost matrix.

TABLE OF CONTENTS

CHAPTER

INTRODUCTION 1

1. OVERVIEW 3
   - 1.1. Discussion of Title 3
   - 1.2. String Generation 6
   - 1.3. Structural Models of Sample Strings 8
   - 1.4. Inference of Structural Models 8
     - 1.4.1. Dynamic Programming 10
     - 1.4.2. Constraining Computational Cost of Dynamic Programming 11
   - 1.5. Classification Using Structural Models 12

2. STRUCTURAL MODELS -- MARKOV NETWORKS 14
   - 2.1. The Markov Process 14
   - 2.2. The Markov Chain 16
   - 2.3. The Markov Network 17
   - 2.4. Relationship of Markov Networks to Inference Algorithm 18

3. DYNAMIC PROGRAMMING 20
   - 3.1. Introduction 20
   - 3.2. Application to Inference Algorithm 20
   - 3.3. Computation of the Dynamic Programming Cost Matrix 25
     - 3.3.1. Editing Operation Costs for 'Live' Nodes 27
     - 3.3.2. Editing Operation Costs for 'Bookkeeping' Nodes 32
     - 3.3.3. Computation Summary 34
   - 3.4. Computational Complexity 35

4. THE CHANNEL CONSTRAINT 36
   - 4.1. Introduction 36
   - 4.2. Restricting Computation to a Channel Through the Matrix 38
   - 4.3. The Channel Constraint 40
   - 4.4. Implementation of the Channel Constraint 42
     - 4.4.1. Absolute Value Computation 42
     - 4.4.2. Checking of Neighbor Cells 42

4.4.3. Restricting the Matrix View 44
4.4.4. Ensuring a Valid Solution 46
4.5. Channel Constraint Summary 48

5. THE IFC IMPLEMENTATION 49
5.1. Introduction 49
5.2. History of IFC Implementations 49
5.3. The Implementation 51
5.3.1. Data Structures 51
5.3.1.1. The Network 51
5.3.1.2. The Dynamic Programming Cost Matrix 54
5.3.1.3. The String Buffer 55
5.3.1.4. The Editing Trace Buffer 56
5.3.2. Code Structure / Algorithms 56
5.3.2.1. The Main Routine 58
5.3.2.2. The Inference Routine 59
5.3.2.3. The Classify Routine 61
5.3.2.4. The Initial Network Routine 62
5.3.2.5. The Ready Matrix Routines 62
5.3.2.6. The Compute Matrix Routine 64
5.3.2.7. The Compute Matrix B Routine 65
5.3.2.8. The Get Trace Routine 66
5.3.2.9. The Modify Network Routine 67
5.4. Implementation Summary 67

6. RESULTS 69
6.1. Timing Results 69
6.2. Classification Results 72
6.2.1. Speculation on Results 79
6.2.2. Problems 82

7. SUMMARY AND CONCLUSION 84
7.1. Summary 84
7.2. New Experiments with Existing Implementation 84
7.2.1. Use of IFC Code Using Longer Strings 84
7.2.2. Use of Channel Constraint in Inference of Markov Networks 85
7.3. Increased Efficiency of IFC Implementation 85
7.4. Study of Better Classification Results When Using Smaller Values of the Channel Constraint 86
7.5. Continuation of Work on Parallel Implementation 86

BIBLIOGRAPHY 88

APPENDIX: IFC USER'S GUIDE 90

1. INTRODUCTION 90

2. INFERENCE 90

3. CLASSIFICATION 94

4. COMPARING CLASSIFICATION RESULTS 95

5. SPECIAL NOTE FOR VMS USERS 96

VITA 97

LIST OF FIGURES

FIGURE

1.1. Constrained Dynamic Programming Inference of Markov Networks from Finite Sets of Sample Strings 4

1.2. String Generation 5

1.3. Landmarking 9

1.4. Structural Model Inference 9

1.5. Classification 13

2.1. Sample Markov Network 15

2.2. Transition Probability Matrix 15

2.3. Output Symbol Matrix 17

3.1. String to Network Alignments 22

3.2. Optimal Alignment Editing Sequence 24

3.3. Computation of the Cost Matrix 26

3.4. Ordering of Computation 27

3.5. Dynamic Programming Cost Matrix and Editing Operations 28

3.6. Multiple Predecessor Nodes of Bookkeeping Nodes 32

3.7. Computation for Rows Associated with Bookkeeping Nodes 33

4.1. Optimal Editing Trace Location Within Cost Matrix 37

4.2. The Channel 39

4.3. New Cost Matrix Cell Elements 41

4.4. Neighbor Cells 43

4.5. Restricting the 'View' of the Matrix 45

4.6. Problem of Channel 'Missing' the Solution 46

6.1. Timing Results from First Version of IFC Implementation 70

6.2. Timing Results from New Version of IFC Implementation 71

6.3. Timing Comparisons Between Two Recent IFC Implementations 72

6.4. Chromosome Groupings 74

6.5. Classification Experiment Results Summary 75

6.6. Correct Classification Percentages -- Learning Data Sets 76

6.7. Correct Classification Percentages -- Test Data Sets 76

6.8. Scaled Confidence Values -- Learning Data Sets 78

6.9. Scaled Confidence Values -- Test Data Sets 78

6.10. Scaled Confidence Values for Chromosome Type 6 Experiments 81

INTRODUCTION

The purpose of the research reported here is to produce a more efficient algorithm for dynamic programming inference of Markov networks.

The following is a brief outline of the material presented in this work. Chapter 1 introduces the concepts and objectives of the research project. Beginning with a definition of the paper's title, the chapter discusses object to string mappings, structural models defined over sets of strings, automated inference of structural models, and the practical utility of these structural models. The chapter briefly defines some of the important terminology used in the paper, including Markov networks, dynamic programming and the channel constraint.

Chapter 2 defines *Markov networks* and explains how they function as structural models. *Dynamic programming,* the basis of the inference process, is discussed in Chapter 3. Also discussed is the sizeable computational cost associated with dynamic programming.

Chapter 4 discusses how the computational intensity of the dynamic programming algorithm has been reduced via the use of the *channel constraint,* the implementation of which is shown to improve the speed of the algorithm by approximately 100%. Some discussion is also given to a curious and unexpected side effect in which the channel constraint generally improves the accuracy of object recognition and classification.

Chapter 5 discusses the computer code (referred to as IFC) produced for this research project. The code has been implemented on a VAX/VMS

machine using the C programming language. Although stopping short of a detailed explanation of the various algorithms, the chapter covers at a fairly high level some of the basic components of the IFC code. These include the basic algorithms and techniques for structural model inference and for dynamic programming using the channel constraint.

Finally, Chapter 6 discusses the results of the various pattern classification experiments. The results focus on timing considerations (speedup) and on the accuracy of classification experiments.

CHAPTER 1

OVERVIEW

This chapter provides a cursory overview of the research and begins with the paper's title, *Constrained Dynamic Programming Inference of Markov Networks from Finite Sets of Sample Strings*, a convenient means of introducing a number of important topics. A more detailed discussion of the key concepts follows.

1.1. Discussion of Title.

Inference is the process by which the computer is fed a set of strings (each of which represents a specific representation of some type of object) and automatically constructs a structural model representing key features of the set of strings. In Figure 1.1 the inference algorithm is illustrated as processing a set of strings as input to produce a structural model of the set of strings as output. The inference process discussed in this paper is completely automatic -- human supervision is neither necessary nor allowed.

"**Finite Sets of Sample Strings**" The data input to the inference machine is a finite set of finite length strings, each of which is comprised of discrete symbols. The set of strings depicted in Figure 1.1 is meant to represent a set of G-banded type-13 human chromosomes[1]. Figure 1.2 illustrates how such strings might be generated from a set of digitized representations of banded human chromosomes. Although human

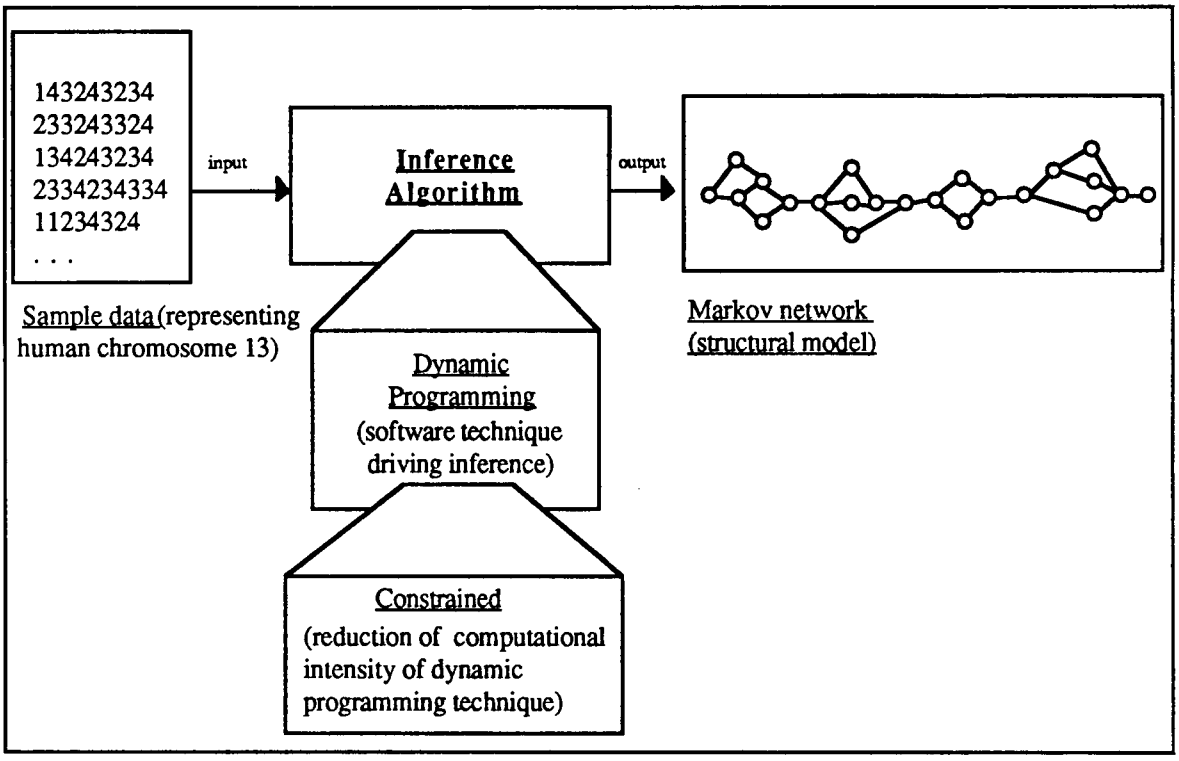


Figure 1.1. Constrained Dynamic Programming Inference of Markov Networks from Finite Sets of Sample Strings.

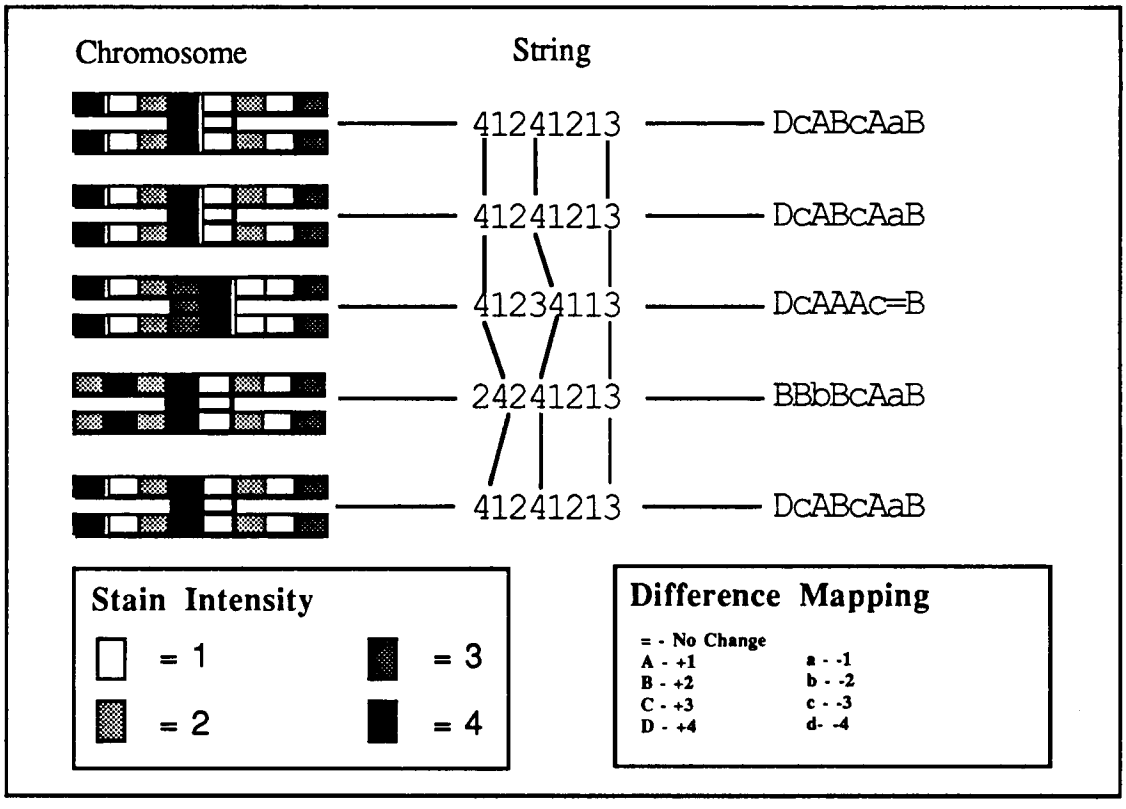


Figure 1.2. String Generation.

chromosome data was used as a source of experimental data for this research, the method of inferring structural models described in this paper will work with any class of objects which can be successfully mapped into finite length strings.

"**Markov Networks**" are the structural models inferred from the input strings. The Markov network is depicted in Figure 1.1 as the output of the inference process. It will be shown that this network structure is defined such that several very important properties of the set of input strings are preserved.

"**Dynamic Programming**" is an optimization technique used within the inference algorithm to find an optimal alignment of a sample string with an existing Markov network. Although it does find an optimal alignment, it does so at a fairly high computational cost.

The term '**constrained**' indicates that the computational cost of the basic dynamic programming algorithm is 'constrained' to be more time efficient and less computer resource intensive. A large part of the discussion in this paper explains **how** the cost of the dynamic programming is constrained in the computer code.

1.2. String Generation.

Figure 1.2 illustrates the process by which strings of symbols can be produced from representations of physical objects. In the figure, images of stained human type-13 chromosomes are represented on the left hand side. The chromosome images have been partitioned into segments of equal length and the stain intensity of each segment (which would normally

span some continuous range of values) has been mapped into one of four discrete values. It is easily noted how this discrete mapping can be transformed into strings of discrete symbols. This research effort used strings which had been transformed through one more step -- into 'difference' strings. Instead of maintaining symbols representing the segment intensities, differences between the intensity symbols are maintained. The resulting strings are found on the right hand side of the figure. The intensity difference mapping is found in the lower right hand corner of the figure.

It should be noted that although all of the banded chromosome images are from the same class of chromosomes (human G-banded, type-13), there are differences in the images which might be due to natural variations, to noise in the original data, or to error in the mapping process. This is a typical characteristic of real-world samples of data which must be dealt with by any process producing structural models. At the same time, there are similar characteristics in each of the five strings. The success of any structural model inference process can be measured by how well it extracts these similar characteristics and represents them in its inferred models.

The inference described in this paper deals with variation by attempting to locate *landmark substrings*, substrings which occur in many, most or all of the input strings. This concept of landmark substrings is illustrated in the figure as lines connecting certain symbols or substrings of symbols in each of the five strings. These connected symbols or substrings appear in every string and thus represent important structural

characteristics of the objects from which the strings were generated. It is shown in the next section that locating and representing these landmark substrings in the structural models is an important characteristic of the inference process.

1.3. Structural Models of Sample Strings.

Figure 1.3 illustrates landmark substrings in a set of five sample strings. A graphic representation of a Markov network is shown below the set of strings. This network, inferred from the set of five strings, is a structural model representing important features of the set of strings (and thus of the set of objects from which the strings were initially generated). The Markov network may be viewed as a 'merged' version of the set of strings, i.e., five strings merged into a single network. Each node of the directed graph (denoted by a circle) has an associated output symbol, a symbol from the set {e, a, A, b, B, c, C, d, D, =}. The nodes with an e-symbol are termed 'bookkeeping' nodes. The other nodes are labeled with symbols in strings from which the network was inferred. Note that just as the strings are viewed as a left-to-right ordering of symbols, the network is viewed as a left-to-right ordering of nodes.

1.4 Inference of Structural Models.

Figure 1.4 summarizes the discussion thus far. In the figure, the inference algorithm receives as input a set of strings and produces as output a structural model of the set of strings. The construction of this Markov network proceeds by first constructing a trivial network from the

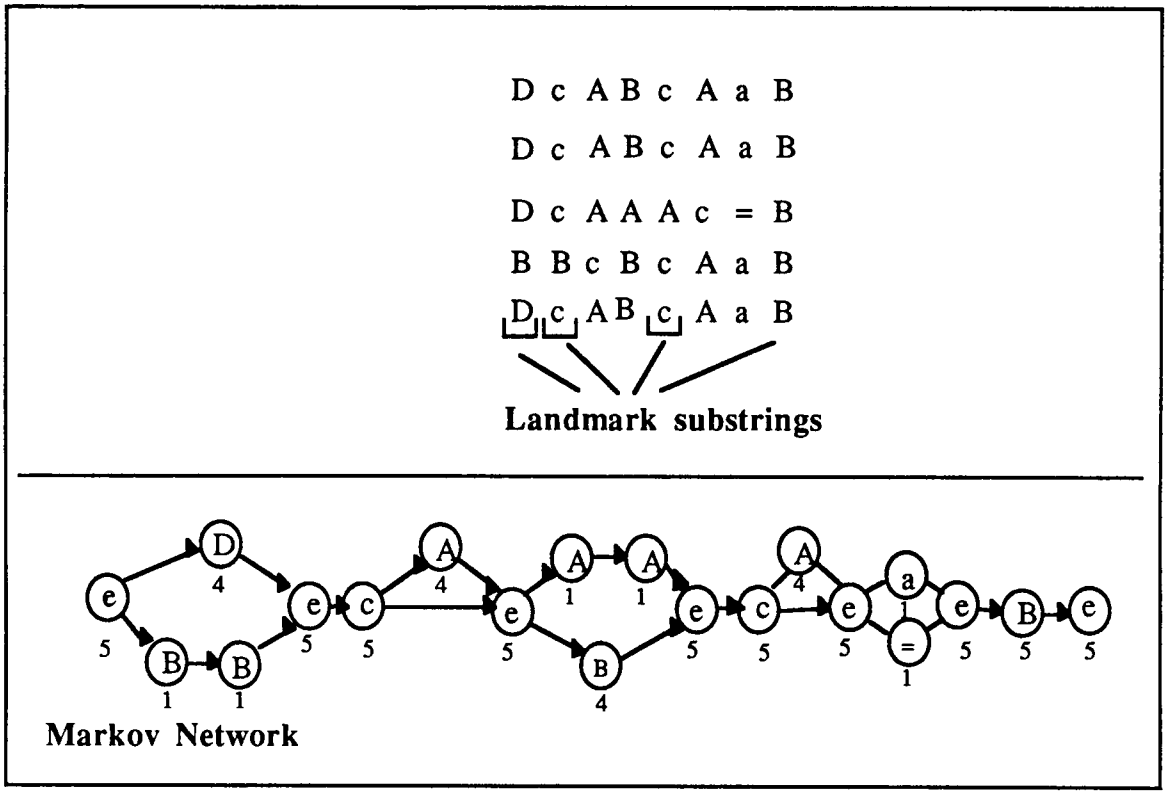


Figure 1.3. Landmarking.

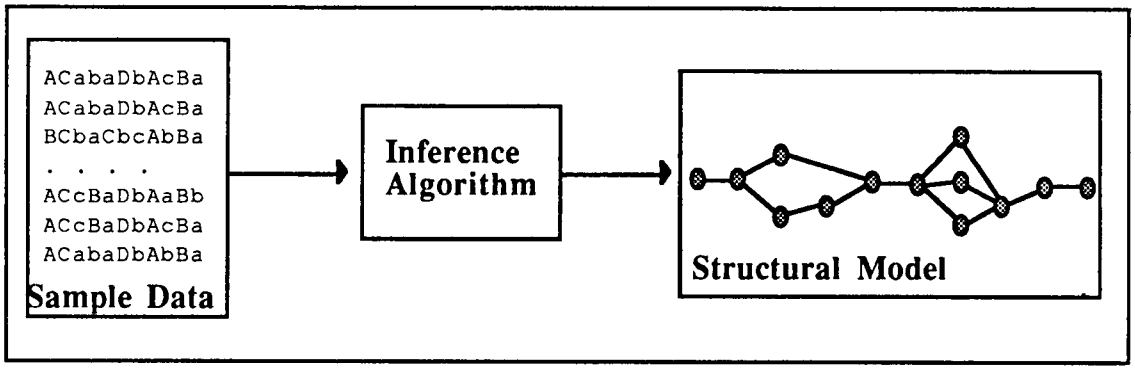


Figure 1.4. Structural Model Inference.

first string in the data -- this initial network is simply a representation of the first string in the sample set . An *optimal alignment* with this initial network is then found for the second string. The network is modified to incorporate the optimal alignment (so that both strings are represented in the network). The process continues for each string in the sample set. Each string is optimally aligned with the current network and the network is then modified to incorporate the newly aligned string. The process ends when all strings have been incorporated into the network. The resulting network represents the entire sample set of strings.

This inference process is order-dependent; i.e., the order in which the strings are incorporated into the network makes a qualitative difference in the composition of the structural model. Experience and many experiments indicate that providing a *sufficiently high number of quality strings* minimizes the effect of order dependence on the construction of the structural model.

1.4.1. Dynamic Programming.

The expensive and difficult portion of the inference process is finding the optimal alignment of a string to a Markov network. This involves the use of a technique called dynamic programming in which a cost matrix is computed. Every cell in this matrix represents the calculation of an optimal alignment of a prefix substring (one beginning with the first symbol in the string) with an initial portion of the network (one beginning with the first node in the network). When the entire matrix has been

computed, information is contained within the matrix on the optimal alignment of the entire string to the entire network. This process maximizes use of the transition probabilities contained within the Markov network and attempts to reinforce *landmark nodes* within the network by aligning symbols in the string with landmark nodes which have identical output symbols.

Unfortunately, dynamic programming is expensive. Calculating the matrix with a string of 80 symbols and a network of 500 nodes (typical sizes for the human chromosome experiments conducted for this research) requires computation of 40,000 cells. The computation of each of these cells requires several calculations, comparisons and data transfers. Thus, although the dynamic programming technique does guarantee an optimal alignment of the string to the network, it does so at a high computational cost.

1.4.2. Constraining Computational Cost of Dynamic Programming.

For reasons which will be explained later, information required for determining the optimal alignment can almost always be found along or near the main diagonal of the matrix (upper left hand corner to the lower right hand corner). This means that calculation of much of the matrix is generally not needed. Much of the effort of this research has gone into developing an algorithm which will guide the computation of the cost matrix such that it follows near the main diagonal of the matrix. A "**channel**" is thus computed down the center diagonal of the matrix and thus many of the cell computations are omitted. It should be noted that

although eliminating computation of parts of the matrix (in this case, the upper right hand and lower left hand corners) generally does not greatly affect the optimal alignment solution, it does eliminate the *assurance* of optimality.

1.5. Classification Using Structural Models.

A fairly high-level view of a technique to infer structural models from sets of sample strings has been shown. This section briefly explains one of the main applications of these structural models -- automatic object classification.

In Figure 1.5 a string is represented which is known *a priori* to represent a GROUP D human chromosome (a type 13, 14 or 15). The problem is to identify to which of the three classes this chromosome belongs. The solution to the problem is initiated by aligning the string with each of the three networks. The alignment operation yields a value which represents the relative probability of alignment of the string to the network. Thus, by aligning the string with each of the three networks, three values are obtained, the highest of which should indicate the desired classification. In this case, the alignment with the type-14 network yields the highest value. Thus, the unknown string is classified as a type-14. Experiments have proven this method to correctly classify strings an average of 96 - 98% of the time with G-banded human chromosome data.

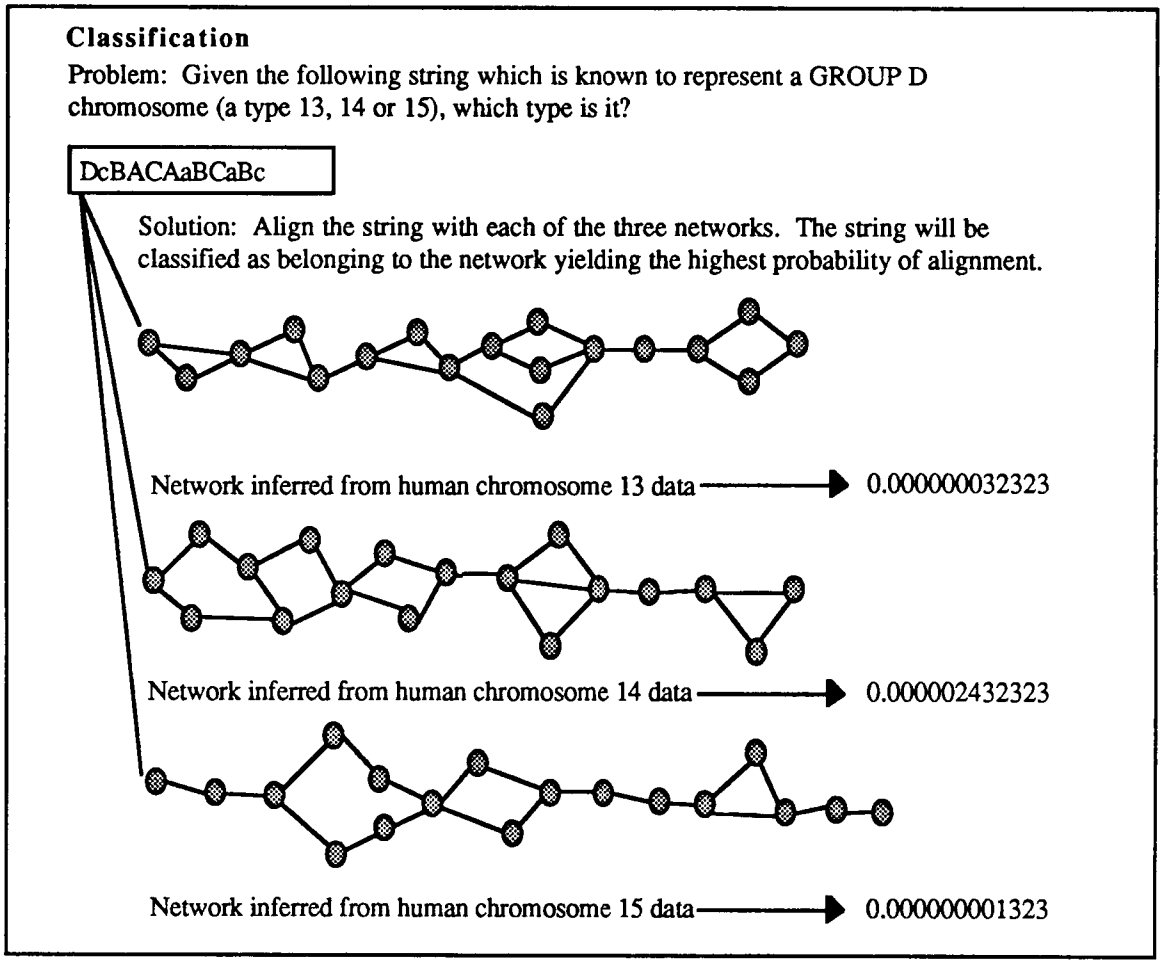


Figure 1.5. Classification.

CHAPTER 2

STRUCTURAL MODELS -- MARKOV NETWORKS

To define the Markov network, this chapter begins by defining the concepts of Markov processes and Markov chains. The Markov network, a specific type of Markov chain, is then defined and discussed regarding its role in this research effort.

2.1. The Markov Process.

The definition of a simple Markov process will refer to Figures 2.1. and 2.2 The definition is as follows:

(i) The Markov process is a system which at any given time is in one of a finite number of states.

(ii) The system undergoes transitions from one state to another at certain discrete intervals of time.

(iii) The probability of a transition from state s_i to state s_j during the next time interval is dependent only upon s_i and s_j, **not** on the history of the system before it reached state s_i.

(iv) There exists a set of conditional probabilities, p_{ij} which specify that a system which now occupies s_i will after the next time interval, occupy s_j. This set of transition probabilities is a matrix $P = [p_{ij}]$.

(v) Since the p_{ij} are probabilities, it must be true that $0 \leq p_{ij} \leq 1$ for all i, j.

(vi) Since the system must be in some state after the next transition, we have:

$$\sum_{j=1}^{N} p_{ij} = 1, \text{ for all i, where N equals the number of states.}$$

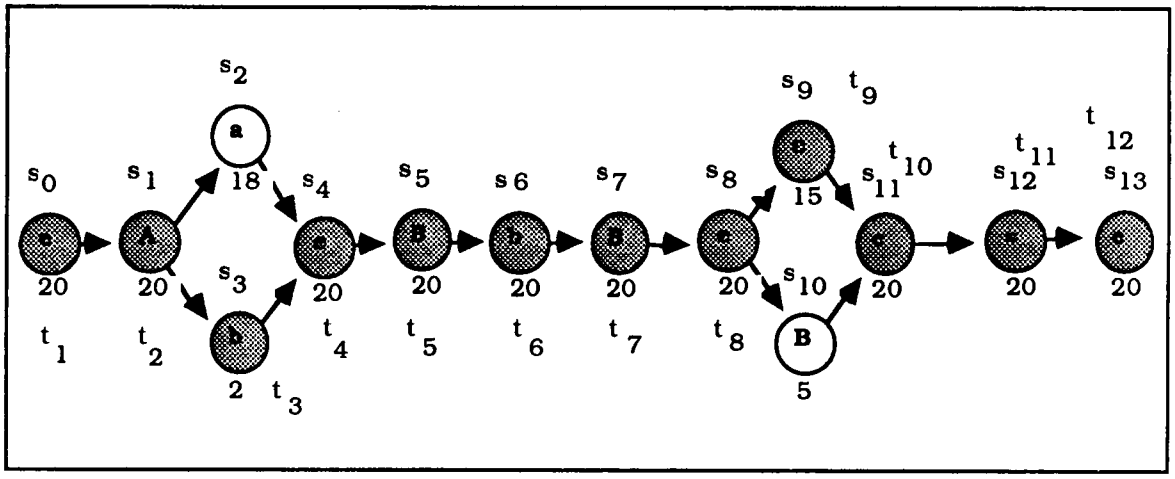


Figure 2.1. Sample Markov Network.

time = t

time = t + 1	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}
s0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s1	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s2	0.00	0.90	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s3	0.00	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s4	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s5	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s6	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s7	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
s8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
s9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.75	0.00	0.00	0.00	0.00	0.00
s10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00
s11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00
s12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
s13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00

Figure 2.2. Transition Probability Matrix.

Figure 2.1 is a graphic representation of a simple Markov process (which also happens to be a Markov network). Each of the 14 states is represented by a circle and is labeled by a state identifier s_n. The shaded states indicate states through which the system might pass during discrete intervals of time. The time identifiers are labeled as t_n. The system in this example begins in state s_0 at time t_1 and then proceeds state by state through the process until it reaches state s_{13} at time t_{12}.

Figure 2.2 represents the transition probability matrix P. The states of the process at time t are shown along the top of the matrix. The states of the process at time t+1 are shown along the left hand side of the matrix. The entries of the matrix are the probabilities that the system which is in state s_i at time t will be in state s_j at time t+1. Looking at the entries of the matrix, it is easily noted that parts (v) and (vi) of the definition are satisfied.

2.2. The Markov Chain.

Next, a Markov chain is a simple Markov process which is characterized by five items[2]:

(i) a set of states, $S = \{s_0, s_1, \ldots, s_n, s_f\}$, also called **nodes**.

(ii) a set of outputs assignable to states, $V = \{v_0, v_1, \ldots, v_T\}$ which includes the unobservable *empty string* $v_0 = e$, and T observable symbols, $v_1, \ldots, v_T$.

(iii) an initial state distribution $\{p^0_0, p^0_1, \ldots, p^0_f\}$.

(iv) the state transition matrix P with entries $p_{ij} = \text{prob}(s_j^{t+1} \mid s_i^t)$.

(v) the output matrix B with entries $b_{ij} = prob(v_k^t \mid s_i^t)$.

Referring again to Figure 2.1, the outputs assignable to states (part ii of the definition) are indicated inside the circles. In the diagram, the vector V = {e, a, A, b, B, c, C, d, D, =}. The transition matrix (Figure 2.2) indicates that the initial state distribution is (1, 0, . . . , 0) -- i.e., the system begins in a single state, s_0. Figure 2.3 illustrates the output symbol matrix B. For this diagram, there is a single output associated with each state. This is evident in the output symbol matrix. Each state outputs with probability 1 a single output symbol.

time = t

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}
v0	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	1.00
v1	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
v2	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
v3	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
v4	0.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00
v5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
v6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
v7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
v8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
v9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00

Figure 2.3. -- Output Symbol Matrix.

2.3. The Markov Network.

Finally, a Markov network is a Markov chain for which[3]:

(i) the initial state distribution is (1, 0, . . ., 0) -- there is a single starting state s_0.

(ii) $p_{ii} = 0$ for $i \neq f$, and $p_{ff} = 1$, i.e., there is a single absorbing state s_f.

(iii) the states are ordered such that $i < j$ implies $p_{ji} = 0$ -- a realization of the process moves "left to right" from s_0 without loops or cycles until absorption in s_f.

(iv) each state deterministically outputs one specific symbol, but different states may output the same symbol.

2.4. Relationship of Markov Networks to Inference Algorithm.

Again referring to Figures 2.1-2.3, the network graph, probability matrix and output matrix illustrate an example Markov network -- one which might be inferred from the application described in this paper.

An important element of the network is the 'frequency of occurrence' number found below each node. This is the number of times the symbol associated with that node has been encountered in the string samples. For example, the first **A**-node (located at the left hand side of the graph) has an associated occurrence number of 20. This is the number of times the **A**-symbol is encountered in the sample strings at or near the beginning of each string.

Probabilities in the Markov network are computed as relative frequencies of occurrence. For example, the first A-node in Figure 2.2 is followed by two nodes, an **a**-node and a **b**-node. The occurrence numbers on these nodes indicate that the arc from the **A**-node to the **a**-node has been travelled 18 times (out of 20) and the arc to the **b**-node has been travelled 2 times. This means that the transition probabilities from the **A**-

node is 0.9 to the **a**-node and 0.1 to the **b**-node. This information plays an important role in the inference process.

Noting the information given in Chapter 1 on how sample strings are 'merged' into the network structure, Figure 2.1 illustrates that the sample set used in inferring this network consisted of 20 strings, all of which begin with a **A**-symbol, 18 of which are then followed by an **a**-symbol (the other 2 being followed by a **b**-symbol), all of which then contained the substring **BbB**, etc. All of the strings ended with an =-symbol. The diagram illustrates how the Markov network structure represents important landmark features of the sample set of data.

In the implementation produced by this research effort, there are two general classes of nodes, *live* nodes and *bookkeeping* nodes. *Live* nodes output an observable symbol (drawn from the symbols in the sample set of strings) and represent the structural and probabilistic information in the network. *Bookkeeping* nodes output the unobservable e-symbol and serve a strictly structural function in the network. For example, in this implementation there is a restriction that requires only one predecessor node for any live node. If there is a need for a live node to have two or more predecessor nodes, a bookkeeping node is placed before the node. Arcs are then placed from the predecessor nodes to the bookkeeping node. This situation is illustrated in Figure 2.1 with the set of nodes {s_2, s_3, s_4, s_5}.

CHAPTER 3

DYNAMIC PROGRAMMING

3.1. Introduction.

The technique used in the inference algorithm for string-to-network alignment is **dynamic programming**, "an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic [being] the multistage nature of the optimization procedure."[4] Dynamic programming is a general framework for finding optimal solutions to a variety of problems. An essential characteristic is the structuring of the given optimization problem into multiple **stages**. Each stage, an optimization problem in itself, is solved sequentially and contributes to the solution of later stages. When all of the stages for a given problem are solved, the optimal solution to the original problem is given. A detailed and formal treatment of dynamic programming will not be given here -- the reader is instead referred to the material in the Bradley, Hax, Magnanti text[5].

3.2. Application to Inference Algorithm.

The optimization problem to be solved in the inference algorithm is described as follows. Given a string, o, and a Markov network, M, an optimal alignment of o with modified network M' (given by incorporating the alignment of o with M) is sought which will give: (1) a series of network modification operations (matches, substitutions, insertions and

deletions) which if applied to M will give M', and (2) a conditional probability value, p, which is the maximum probability with which the modified network M' can generate the string o.

Dynamic programming is thus used to find a minimal cost editing sequence for mapping a string into a path in a given network. Costs representing modification of the network structure are assigned to the four atomic operations on symbols in a string -- deletions, insertions, substitutions, and matches. The desired string to network mapping is the one which minimizes the total cost of the network modifications.

Figure 3.1 illustrates this mapping problem. The optimal mapping operation takes advantage of the probabilistic information contained within the network structure and thus attempts to reinforce landmark substrings. In Figure 3.1, two different alignments of a string with a network are shown. The first alignment is optimal -- it successfully reinforces landmark nodes by matching symbols of the string with nodes of the network which occur with relatively high probability. The second alignment, although a valid alignment of the string to the network, is not optimal.

Each stage of the inference problem corresponds to finding the optimal alignment of a prefix substring of the sample string (one which begins with the first symbol of the string) with an initial sequence of nodes within the network (a sequence which begins with the first node of the network). Given a string of 100 symbols and a network of 700 nodes, the optimal alignment of the string with the network requires the solution of 70,000 stages. The solution of the 70,000th stage (alignment of the initial

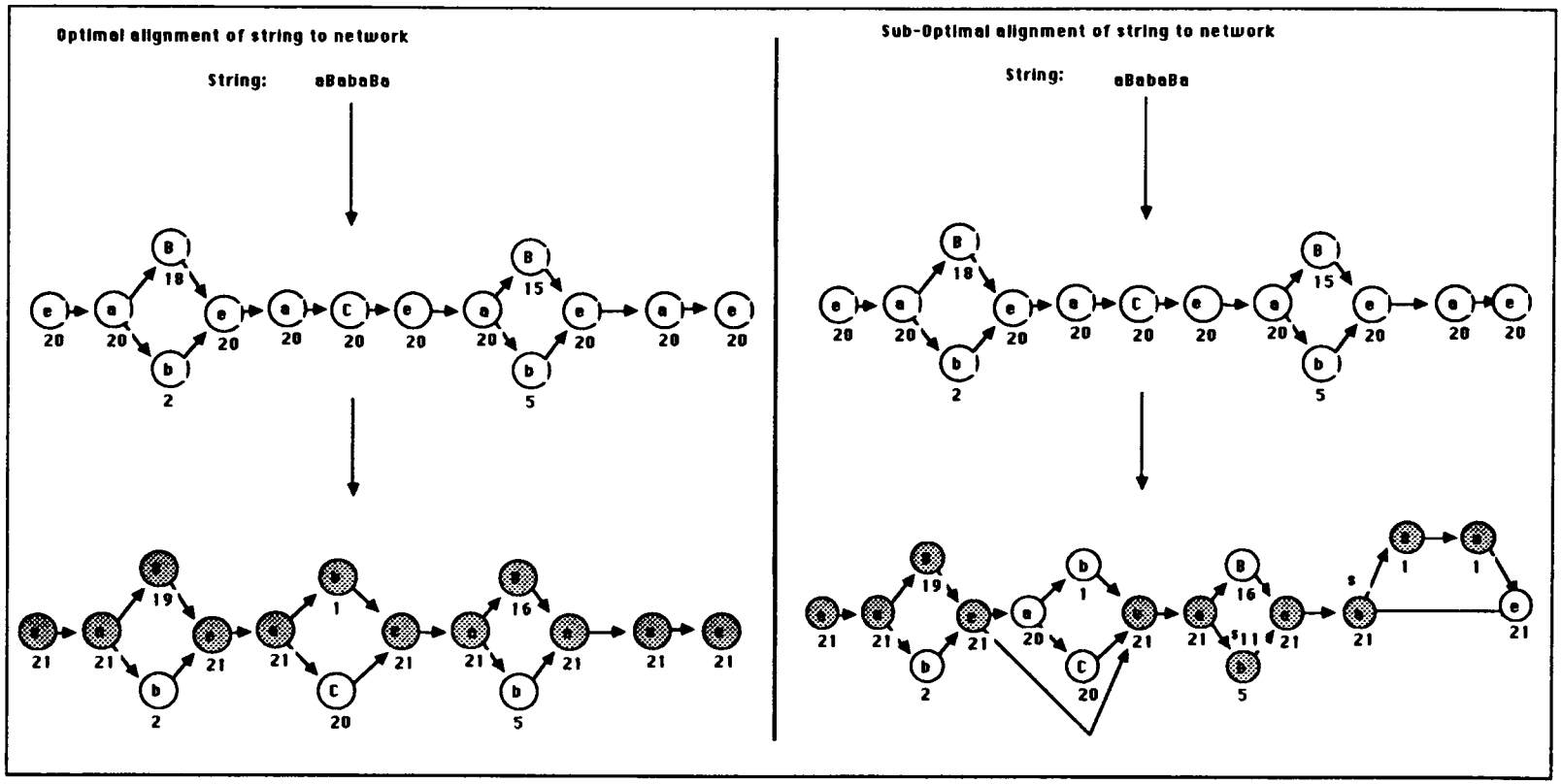


Figure 3.1. String to Network Alignments.

substring of 100 symbols (the entire string) with the initial sequence of 700 nodes (the entire network)) yields the optimal solution to the original problem.

Dynamic programming proceeds by computing a cost matrix (with each cell of the matrix corresponding to one stage of the dynamic programming problem) resulting in a trace from the upper left cell to the lower right cell of the matrix which gives an optimal alignment of the string with the network. Figure 3.2 illustrates the resulting editing sequence drawn from a computed cost matrix. The editing sequence implies the following: The first 'M' (in the upper left hand corner) implies a match operation of the first e-symbol in the string with the initial e-node in the network. The next 'M' implies matching the a-symbol with the first a-node. Three more match operations are indicated and then a substitution of the 'b' symbol for the 'C'-node is indicated by the letter 'S'. "Insertions" and "Deletions" are other operations indicated in editing traces. Also illustrated in Figure 3.2 is the incorporation of this optimal alignment into the network. The resulting network has reinforced many of the nodes (for example, the prefix substring 'aBa' has an even higher probability of occurrence than it did before the new string was incorporated) and has added one new node (the 'b' node).

Given that we obtain the optimal editing sequence (and thus the optimal alignment) from information contained in the cost matrix, the natural issue to deal with next is the computation of the cost matrix. The following section describes the computation and illustrates the practical implementation of dynamic programming.

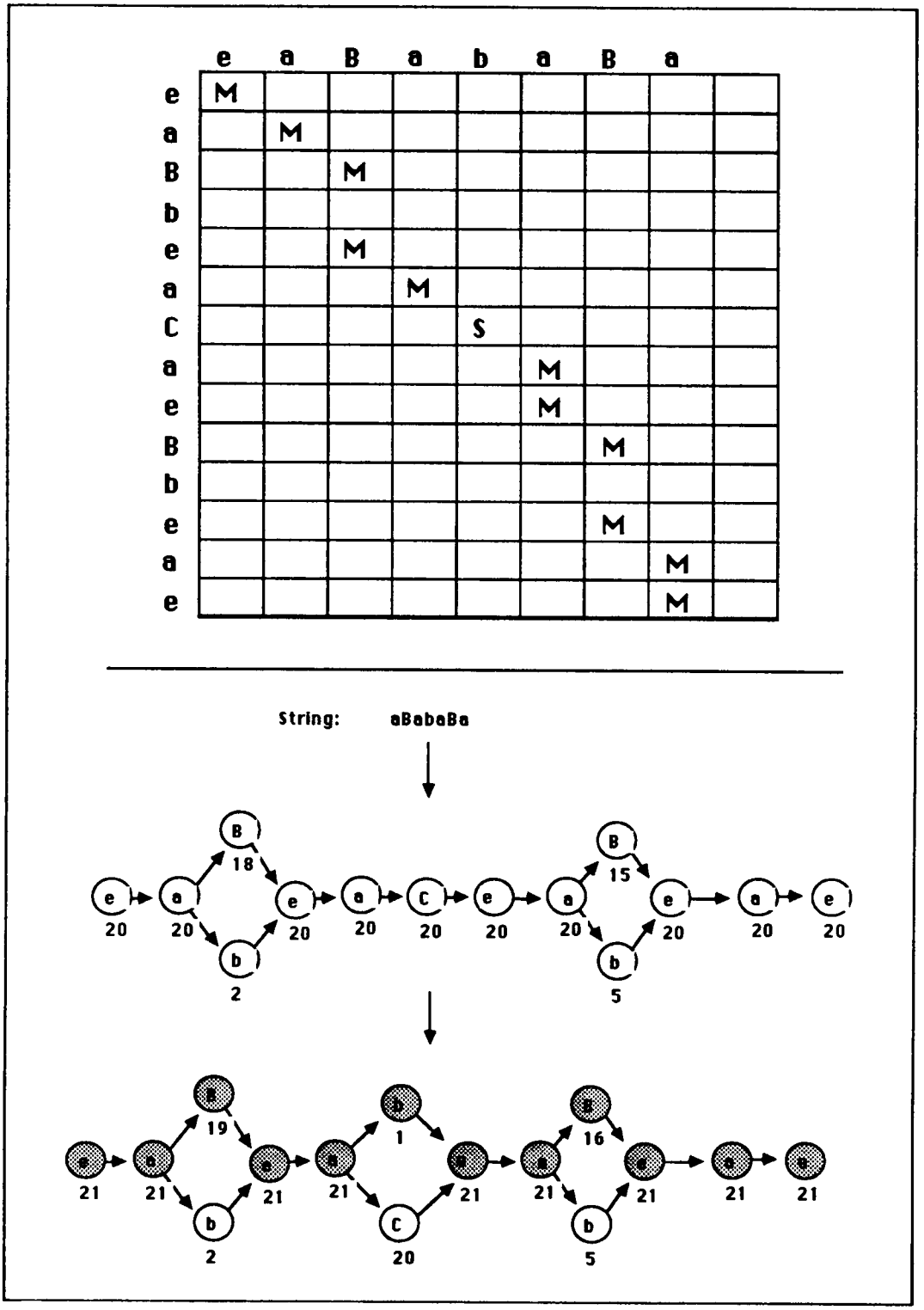


Figure 3.2. -- Optimal Alignment Editing Sequence.

3.3. Computation of the Dynamic Programming Cost Matrix.

The cost matrix has dimensions m x n where m is the number of symbols in the string and n is the number of nodes in the network. It is helpful to consider each symbol of the string associated with one column of the matrix. Likewise, each node of the network is associated with one row of the matrix. Thus, each cell of the matrix is at the intersection of a network node and a string symbol and thus corresponds to a **stage** in the dynamic programming problem. The completion of each stage implies the optimal alignment of a prefix substring with an initial portion of the network.

Figure 3.3 illustrates this with a short string and network. The computation of cell A gives the optimal solution to the mapping of substring $z_0..z_2$ to the portion of the network comprised of nodes $s_0..s_2$. Cell B yields the optimal solution to the alignment of $z_0..z_4$ to $s_0..s_4$; cell C, $z_0..z_2$ to $s_0..s_8$. When computation of the matrix completes with cell E, the optimal alignment of the substring $z_0..z_7$ to the network $s_0..s_{14}$ is given -- this is the solution to the original problem.

As Figure 3.4 indicates, computation of the matrix begins with the upper left corner cell and proceeds on a row-wise basis until it finishes with the lower right corner cell of the matrix. The procedure calculates for each cell the maximum probability with which a modified portion MP' of the partial network MP could generate the current prefix substring. Thus, the probability value is the product of the probabilities associated with

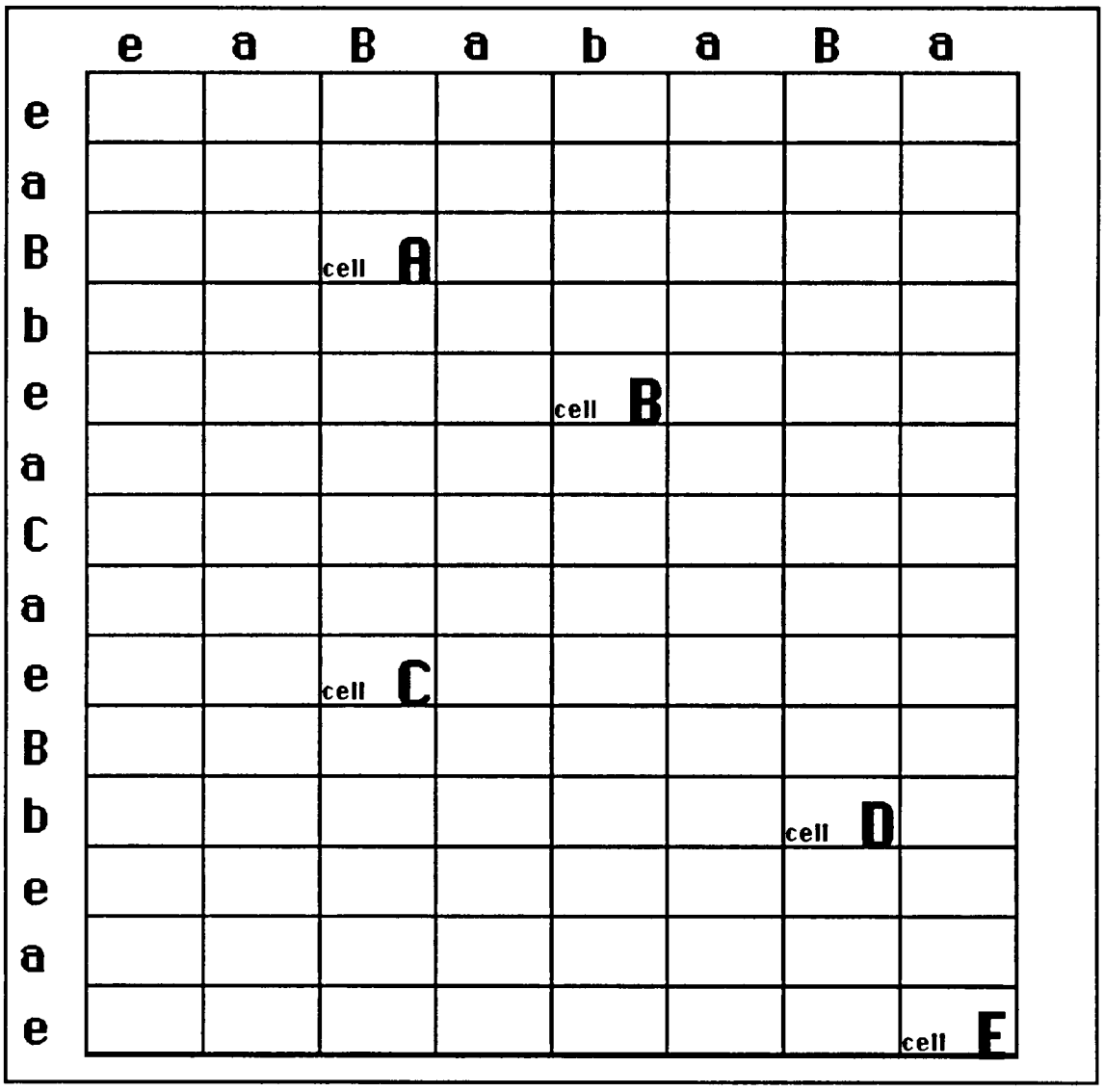


Figure 3.3. Computation of the Cost Matrix.

nodes on the optimal network path the output symbols of which (concatenated together) would yield the current substring.

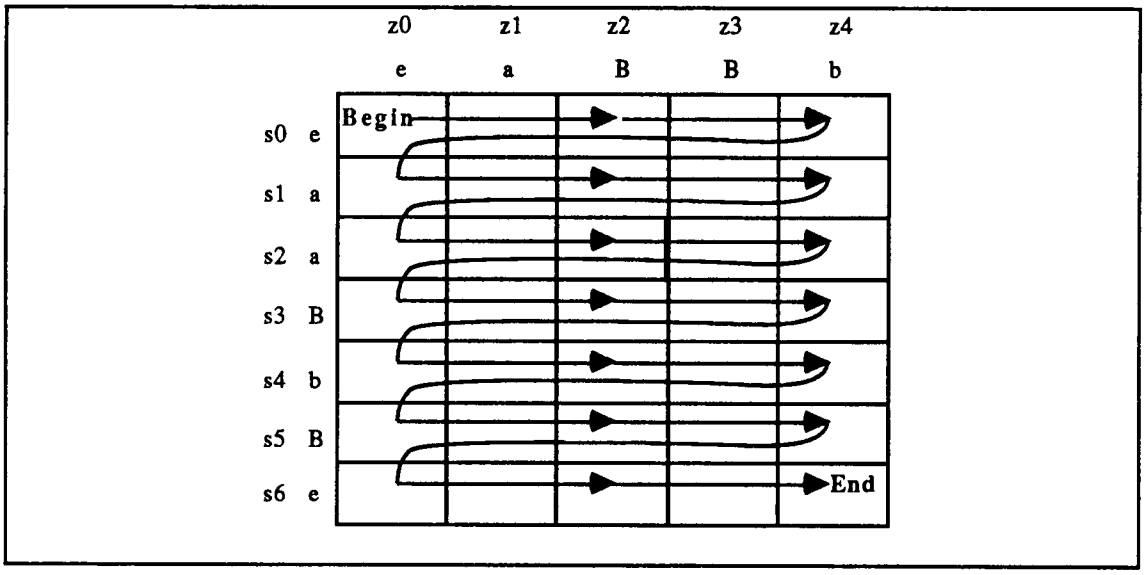


Figure 3.4. Ordering of Computation.

As mentioned at the end of Chapter 2, there are two classes of nodes in the network, 'live' nodes and 'bookkeeping' nodes. The computation of cells contained in a row associated with a 'live' node is different from the computation of cells associated with a 'bookkeeping' node. The following sections discuss the two algorithms.

3.3.1. Editing Operation Costs for 'Live' Nodes.

The following discussion of costs refers to the two diagrams in Figure 3.5. The top diagram represents the cost matrix. The other represents the four network modification operations. The matrix is represented with the

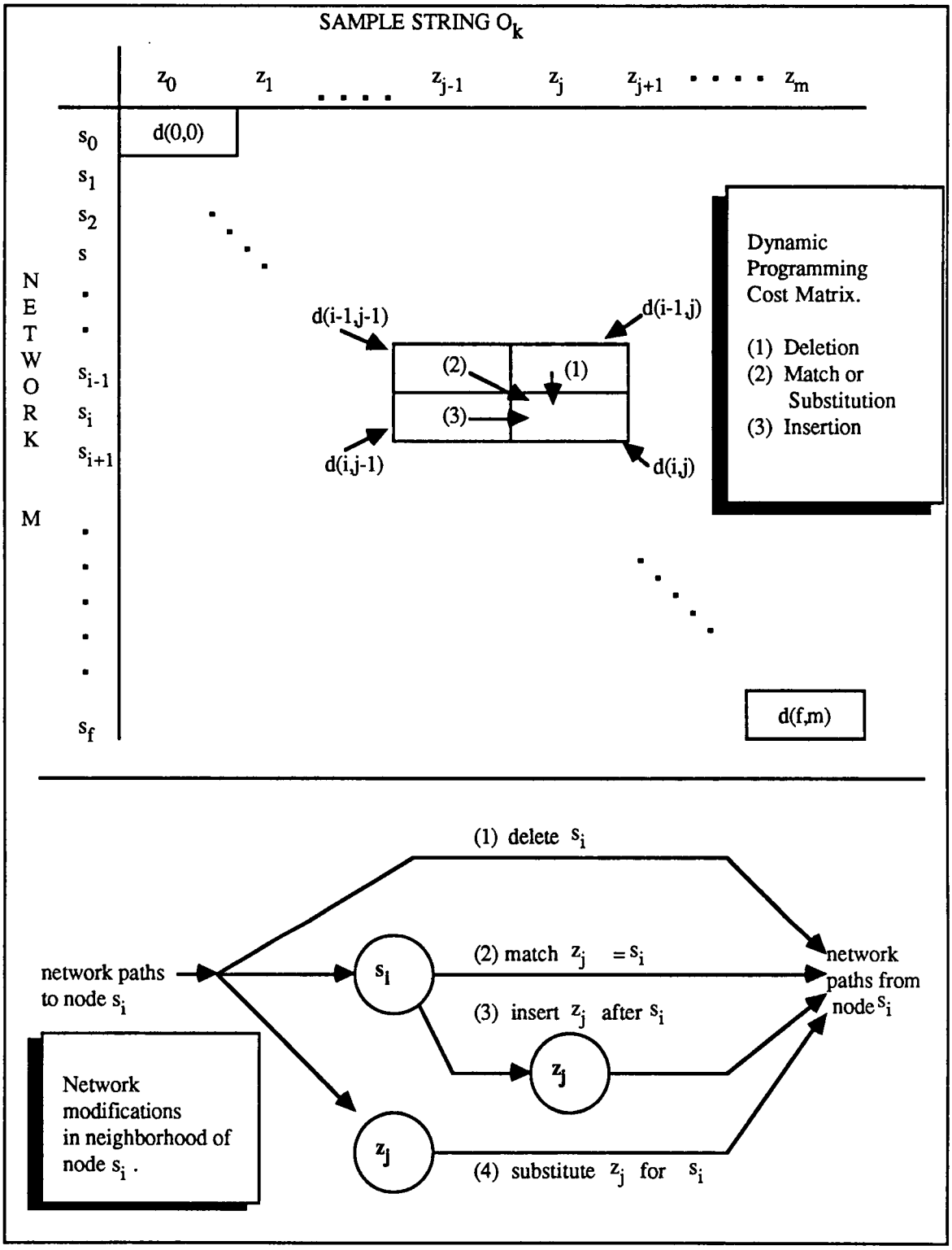


Figure 3.5. Dynamic Programming Cost Matrix and Editing Operations.

symbols of a sample string defining columns of the matrix and nodes of a network defining the rows. Thus, the cell indicated by d(i,j) represents the optimal alignment of the substring $z_0..z_j$ with the initial portion of the network $s_0..s_j$. Because of the order in which the matrix is computed, it is given that the three neighboring cells of d(i,j) (indicated by d(i-1,j-1), d(i-1,j), and d(i,j-1)) have already been computed and thus the maximum probability values representing the optimal alignment of $z_0..z_{j-1}$ to $s_0..s_i$, of $z_0..z_{j-1}$ to $s_0..s_{i-1}$, and of $z_0..z_j$ to $s_0..s_{i-1}$ are already known.

The **match** operation uses the optimal alignment of the $z_0..z_{j-1}$ substring to the network portion $s_0..s_{i-1}$ as a base for matching symbol z_j to node s_i. The cost of modifying the network structure is simply the probability value from cell (2) multiplied by the cost of updating the network structure for a match operation. This operation involves incrementing by one the frequency count of node s_{i-1} and the frequency count of the arc connecting nodes s_{i-1} and s_i. Thus, the following formula is used for computing the cost of the match operation.

$$d(i - 1, j - 1) * \frac{f_{i-1,i} + 1}{f_{i-1} + 1}$$

Because these probability values quickly become so small as to cause problems with underflow on the computer, the natural logarithm of the probability value is instead maintained. Thus, the actual value computed for the match operation is represented by the following formula.

$$d(i - 1, j - 1) + \log(\frac{f_{i-1,i} + 1}{f_{i-1} + 1})$$

It should be noted that the match operation can only be computed if symbol z_j is equal to the symbol output by node s_i -- i.e., the string symbol must *match* the output symbol of the node.

The **substitution** operation uses the optimal alignment of the $z_0..z_{j-1}$ substring to the network portion $s_0..s_{i-1}$ as a base for substituting symbol z_j for the symbol output by node s_i. The cost of modifying the network structure is the probability value from cell (2) added to the cost of updating the network structure for the substitution operation. The operation involves incrementing the frequency count by one of node s_{i-1} and the frequency count of the arc connecting nodes s_{i-1} and s_i. Since the substitution operation also involves the addition of a new node (with output symbol z_j) to the network (insertion is after the node which precedes node s_i), the cost of traversing the new arc to the new node must also be considered. Thus, the following formula is used for computing the cost of the substitution operation.

$$d(i - 1, j - 1) + \log(\frac{f_{i-1,i} + 1}{f_{i-1} + 1}) + \log(\frac{1}{f_i + 1})$$

The **insertion** operation uses the optimal alignment of the $z_0..z_{j-1}$ substring to the network portion $s_0..s_i$ as a base for inserting a node with output symbol z_j into the network. The cost of modifying the network structure is the probability value from cell (3) added to the cost of updating the network structure for the insertion operation. The operation involves computing the cost of traversing the new arc to the new node. Thus, the following formula is used for computing the cost of the insertion operation.

$$d(i, j - 1) + \log(\frac{1}{f_i + 1})$$

The final operation to be considered is the **deletion** operation. This operation uses the optimal alignment of the $z_0..z_j$ substring to the network portion $s_0..s_{i-1}$ as a base for deleting (or bypassing) node s_i. The cost of modifying the network structure is the probability value from cell (1) added to the cost of updating the network structure for the deletion operation. The operation involves computing the cost of travelling around node s_i (and thus adding a new arc to the network structure). Thus, the following formula is used for computing the cost of the deletion operation.

$$d(i - 1, j) + \log(\frac{f_{i-1,i} + 1}{f_{i-1} + 1}) + \log(\frac{1}{f_i + 1})$$

To find the operation which yields the highest probability value, the insertion, deletion and substitution values must all be calculated. If z_j is equal to the output symbol of s_{i-1}, the match operation must also be calculated. A code representing the operation yielding the highest probability value is stored for the cell along with the probability value. Also maintained is the node from which the operation was calculated. If the insertion operation yielded the highest value then this 'previous-node' value is set equal to node s_i. Otherwise, it is set equal to s_{i-1}. The operation code and the previous node values comprise the information needed for the optimal editing trace. The probability value will be used in the later stages of the dynamic programming problem.

3.3.2. Editing Operation Costs for 'Bookkeeping' Nodes.

Bookkeeping nodes play a special structural role in the networks and must be handled differently in the dynamic programming computation. First, it is important to note that the single-predecessor restriction placed on 'live' nodes does not apply to bookkeeping nodes. This is illustrated in Figure 3.6 in which the bookkeeping node s_8 has three predecessor nodes s_5, s_6 and s_7. Since the bookkeeping node outputs an unobservable

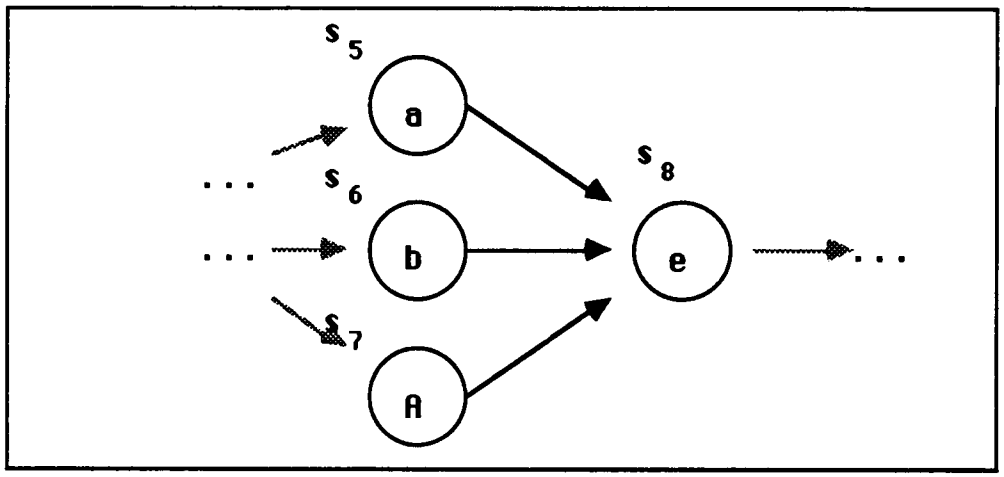


Figure 3.6. Multiple Predecessor Nodes of Bookkeeping Nodes.

symbol, the standard editing operations are not applicable. Instead, in computing cells in the row associated with a bookkeeping node, the equivalent to a 'match' operation is computed from the predecessor node yielding the maximum value. Figure 3.7 illustrates a cost matrix corresponding to the partial network of Figure 3.6. Cell D is in the row corresponding to the bookkeeping node. To compute the probability value

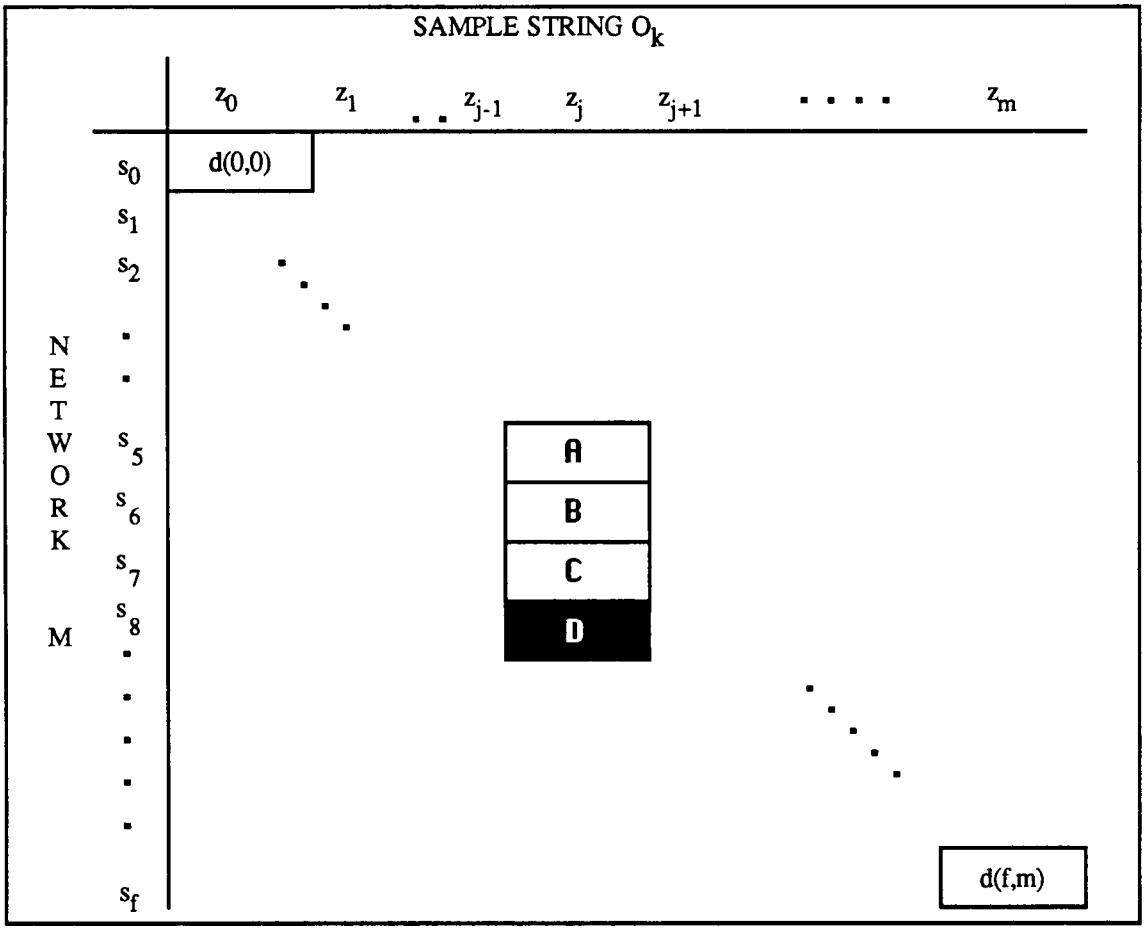


Figure 3.7. Computation for Rows Associated with Bookkeeping Nodes.

for this cell, the maximum probability value must be calculated from the three cells A, B and C. The calculation for each of these is

$$d(pn, j) + \log(\frac{f_{pn,i} + 1}{f_{pn} + 1})$$

where pn indicates "predecessor node".

Thus, to compute the probability value for a cell in a row corresponding to a bookkeeping node, the above calculation is applied to every predecessor node. The probability value is set equal to the discovered maximum value. The previous-node element of the cell is set to indicate the predecessor node yielding the maximum value. A special operation code is maintained for the cell indicating that the bookkeeping node operation was used.

3.3.3. Computation Summary.

The computation of the dynamic programming cost matrix is as follows: Beginning with the first cell in the cost matrix, the algorithm proceeds on a row-wise basis, calculating for each cell a probability value, an operation code and a previous node value. The algorithm used for a given row depends upon whether the corresponding network node is a live node or a bookkeeping node. When calculation of the matrix is complete, two sources of information are available: An optimal editing trace is embedded in the cost matrix; and the maximum probability with which a modified version of the network can generate the string is located in the bottom right hand corner cell of the matrix.

3.4. Computational Complexity.

The complexity of the described computation is proportional to m*n where m is the length of the string and n is the number of nodes in the network. Thus, the complexity is proportional to the number of stages to be solved in the dynamic programming problem. For most practical applications, this cost of the matrix computation is prohibitive because the size of the sample strings and of the network is usually large. It is shown in the next chapter how efforts to lessen this prohibitive cost have been implemented.

CHAPTER 4

THE CHANNEL CONSTRAINT

4.1 Introduction.

The primary motivation behind this research was to develop an algorithm which would allow the matrix computation step to ignore those portions of the matrix which rarely contribute to the string-to-network alignment problem. Figure 4.1 illustrates an important fact concerning the solution of a dynamic programming problem. Even though the solution of the problem (the optimal editing trace) is found only in the black-shaded cells, the previously described algorithm requires computation of **all** the cells in the matrix.

Although computation of the entire matrix does **guarantee** the optimal alignment of string to network, it would be very unlikely for the outer corner regions (upper right, lower left) of the matrix to be needed for the solution. For one of the two outer regions to contain a portion of the optimal trace, one of two unlikely situations would have to occur: (1) a high number of insertions would have to be followed by an approximately equally high number of deletions, or (2) a high number of deletions would have to be followed by an approximately equally high number of insertions. Experience and many experiments have shown that since the match and substitution operations are such commonly encountered operations in optimal editing traces and since they tend to guide the trace

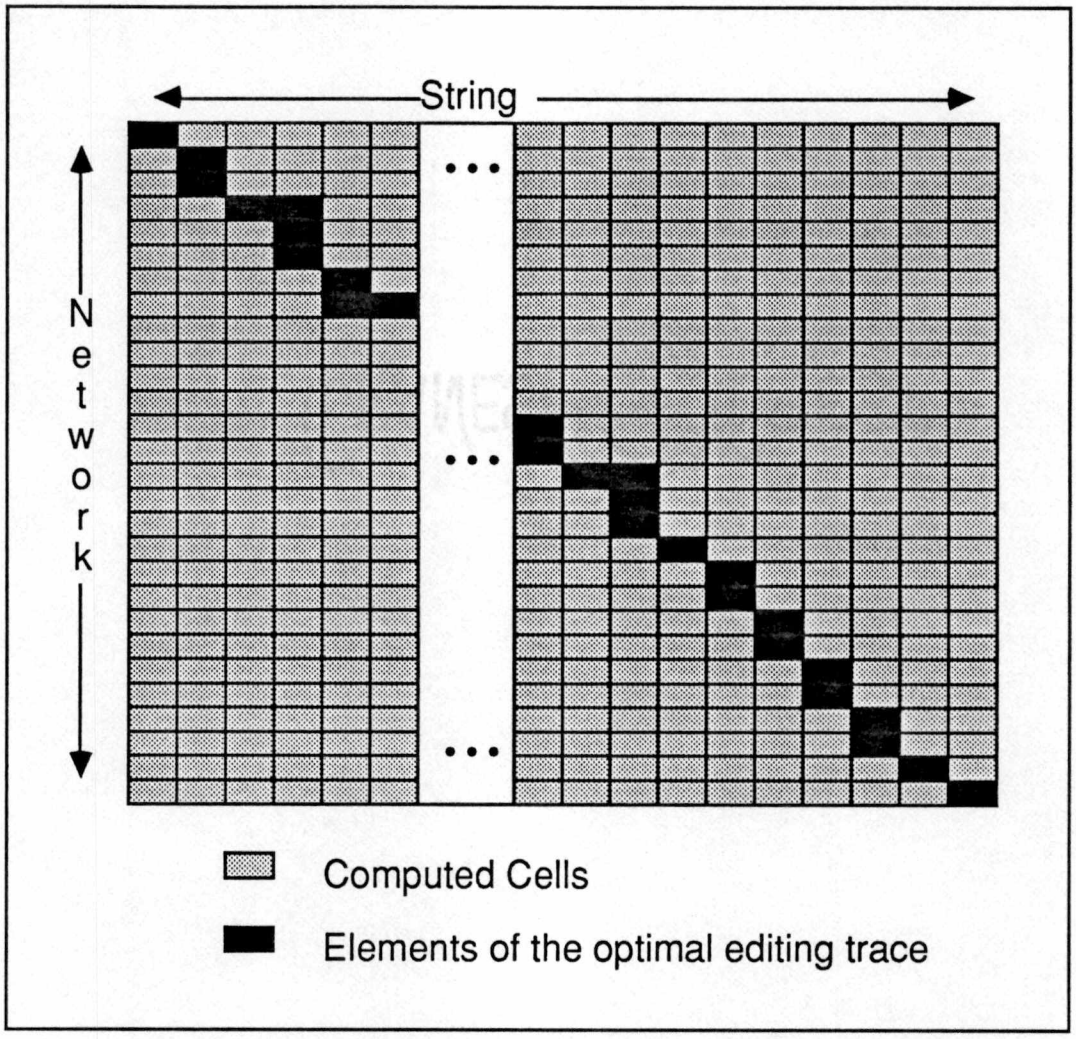


Figure 4.1. Optimal Editing Trace Location Within Cost Matrix.

down a diagonal of the matrix, most computations tend to follow along or near the main diagonal.

4.2. Restricting Computation to a Channel Through the Matrix.

The desired computational set-up is to restrict computation of the matrix to that portion surrounding the main diagonal. This portion of cells surrounding the matrix is referred to as the *channel*. This concept is illustrated in Figure 4.2. There are two problems with this solution.

The first problem is that ignoring any part of the matrix eliminates the assurance of an **optimal** mapping of string to network. Unfortunately, there is no solution to this problem -- it must be hoped that an optimal or near-optimal trace will fall within the computed channel. Of course, the width of the channel can always be increased -- but only with corresponding increase in computational effort.

The second problem is in the increased complexity of the matrix computation. It is a trivial matter when computing the entire matrix to select the cells to be computed -- they are all computed. It is not trivial to guide the selection of cells to be computed so that the computation proceeds along the main diagonal and finishes with the last cell in the matrix. The next section will describe how this has been accomplished.

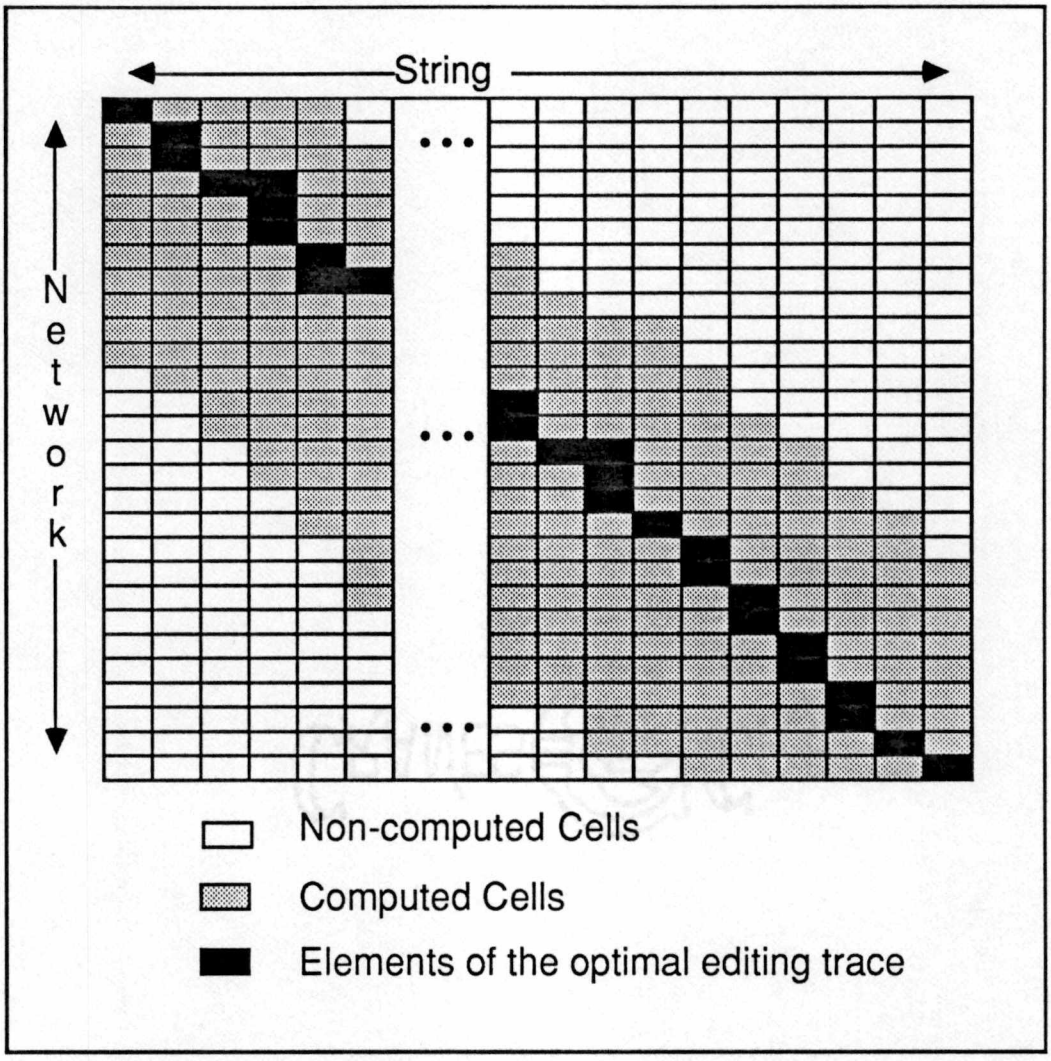


Figure 4.2. The Channel.

4.3. The Channel Constraint.

To guide the computation of the matrix along the main diagonal, there are several important factors to be considered. First, it must be ensured that the last cell in the matrix is included among the cells that are computed (it is the trace from the last cell to the first cell which gives the editing alignment solution). Second, the selection of cells should be such as to proceed near the main diagonal. Third, there should be a way of describing how large a portion of the matrix surrounding the main diagonal should be computed -- different string to network mapping problems may require different 'widths' of the computed channel.

In this implementation, the width of the channel is constrained by computing cells for which the absolute value

$$| \text{cumulative number of insertions} - \text{cumulative number of deletions} |$$

falls within a supplied parameter value which must be ≥ 1 and $\leq m$ (where m is the length of the string). This parameter is called the **channel constraint.**

In the implementation, it is necessary to associate two additional values with every cell in the matrix. This is illustrated in Figure 4.3. The full matrix computation implementation requires a probability value, an operation code value and a previous node value. The constrained matrix computation requires two additional elements -- number of insertion operations and number of deletion operations. These two elements contain the cumulative number of the described operation encountered in the trace from the first matrix cell to the current cell. By constraining these two elements to remain within a set range of each other, the computation

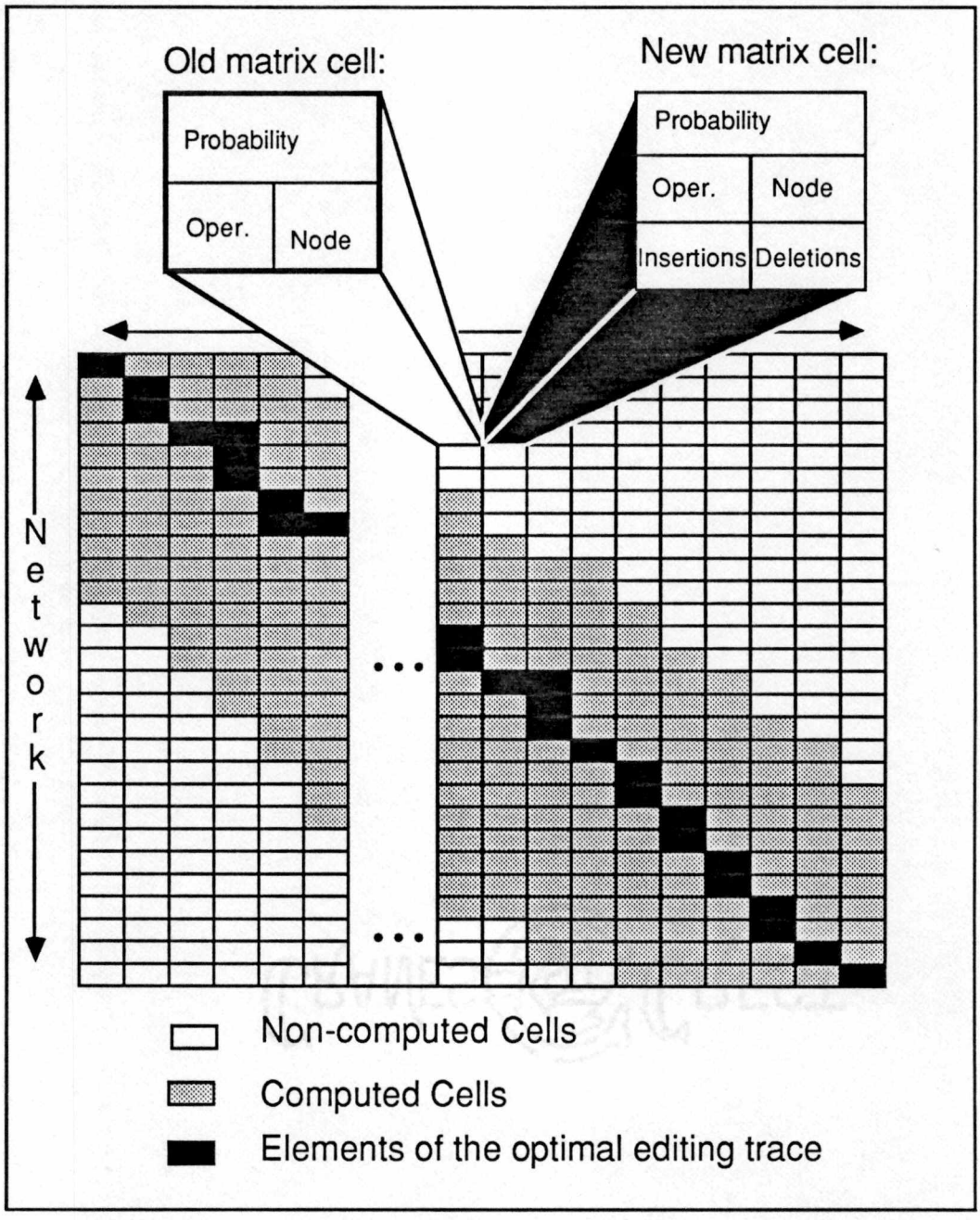


Figure 4.3. New Cost Matrix Cell Elements.

is prohibited from extending too far into both the upper right and lower left hand regions.

4.4 Implementation of the Channel Constraint.

The basic algorithm for using the channel constraint differs in many respects from the earlier described algorithm. This section will discuss the algorithm and will also discuss the problems and the encountered overhead. It should be noted that the overhead involved in managing the channel is quite large and does reduce the improved performance of the technique.

4.4.1. Absolute Value Computation.

One of the prices to be paid for use of the channel constraint is that extra work is required when computing every cell of the channel. Performing the absolute value calculation before deciding to compute a cell and maintaining the number of insertions and number of deletions elements requires additional computations for each cell in the channel.

4.4.2. Checking of Neighbor Cells.

There is also an additional step when computing the probability value for a cell as it must first be ensured that the cells from which the computation is based (i.e., all neighboring cells) are themselves computed cells. When the matrix is initialized for use with a given string, all probability values are set equal to a special value. If the computation of a

cell in the matrix encounters that special value in a neighboring cell, then it is known that that neighboring cell has not been computed. Figure 4.4 illustrates a cell to be computed and its three neighboring cells. Before the probability value for an insertion operation can be computed, it must first

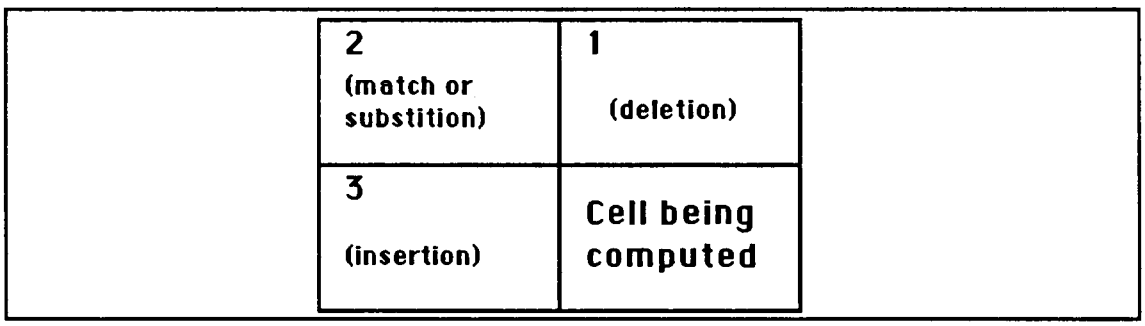


Figure 4.4. 'Neighbor' Cells.

be ensured that cell #3 was computed. If cell #3 was not computed then the insertion operation is not considered for the cell. This also holds for the deletion operation with cell #1 and for the match/substitution operations with cell #2. For bookkeeping nodes, the appropriate matrix cell associated with each predecessor node must be checked. Obviously, this checking of neighbor cells adds to the overhead of the channel constraint solution.

4.4.3. Restricting the Matrix View.

Another problem with the proposed channel constraint is that every cell in the matrix must still be dealt with even if the probability value is not computed for the cell -- it was discovered in an earlier implementation that this overhead completely eliminated the computational savings of the channel constraint. Thus, the following solution, illustrated in Figure 4.5, was implemented.

For each row in the matrix, two new data elements were added. The channel_begin element indicates the column of the matrix in which computations are to begin for that row. The channel_end element indicates the column in which computations are to end for that row. Whenever processing of a row in the matrix has concluded, the channel_begin and channel_end parameters are computed for all successor nodes (those which follow the associated node via output arcs) . This is easily accomplished by looking at the insertions and deletions elements in the first and last cells computed in the current row and computing, for example, how much wider (if any) the channel for the successor node row should be. By the time the successor node row is encountered in the matrix, its channel_begin and channel_end parameters are already known.

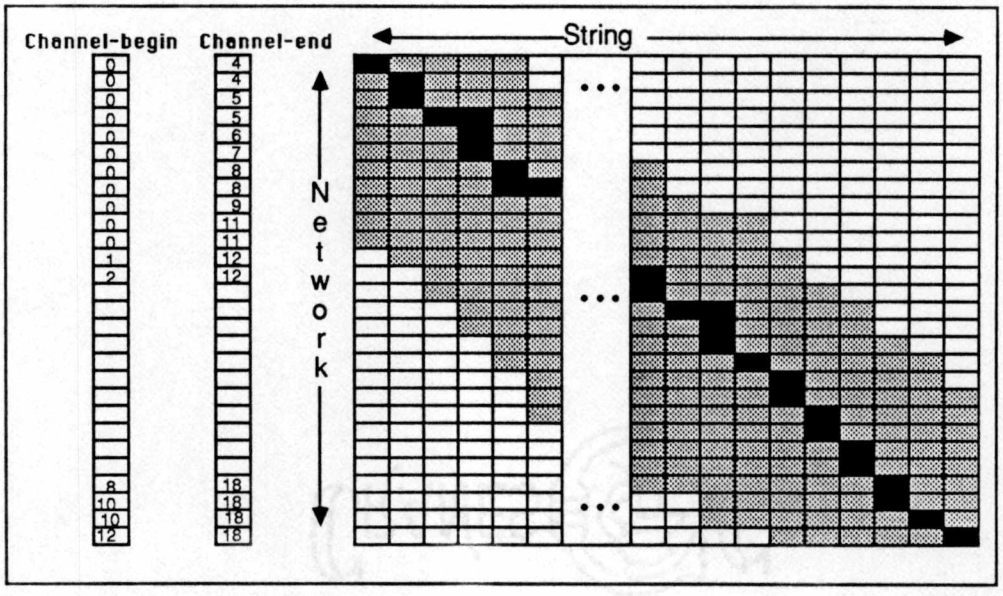


Figure 4.5. Restricting the 'View' of the Matrix.

4.4.4. Ensuring a Valid Solution.

It is possible when using the channel constraint to constrain processing so that the last cell of the matrix is not computed. This is illustrated in Figure 4.6. Computation of the last cell of the matrix concludes the problem of finding the near-optimal mapping of the entire string to the entire network. If this last cell is not reached during the computation, then the algorithm has failed to find a solution to the problem. The solution to this problem is fairly simple but differs based on whether the dynamic programming is being used for inference or classification.

Because the inference algorithm requires the editing trace information (to incorporate the string into the network), the last cell must be reached. If this cell is not reached, then the channel constraint parameter is incremented by some integer and the matrix re-computed. This is repeated until the last cell is reached during the computation.

As the classification algorithm does not modify the network with the editing trace information but rather merely reports the probability value (found in the lower right corner cell), one solution is to just report a probability value of 0 and assume that the string so poorly mapped into the network that the value of 0 is reasonable. The other solution is to continue incrementing the channel constraint parameter and recomputing the matrix until the last cell and thus a solution is reached.

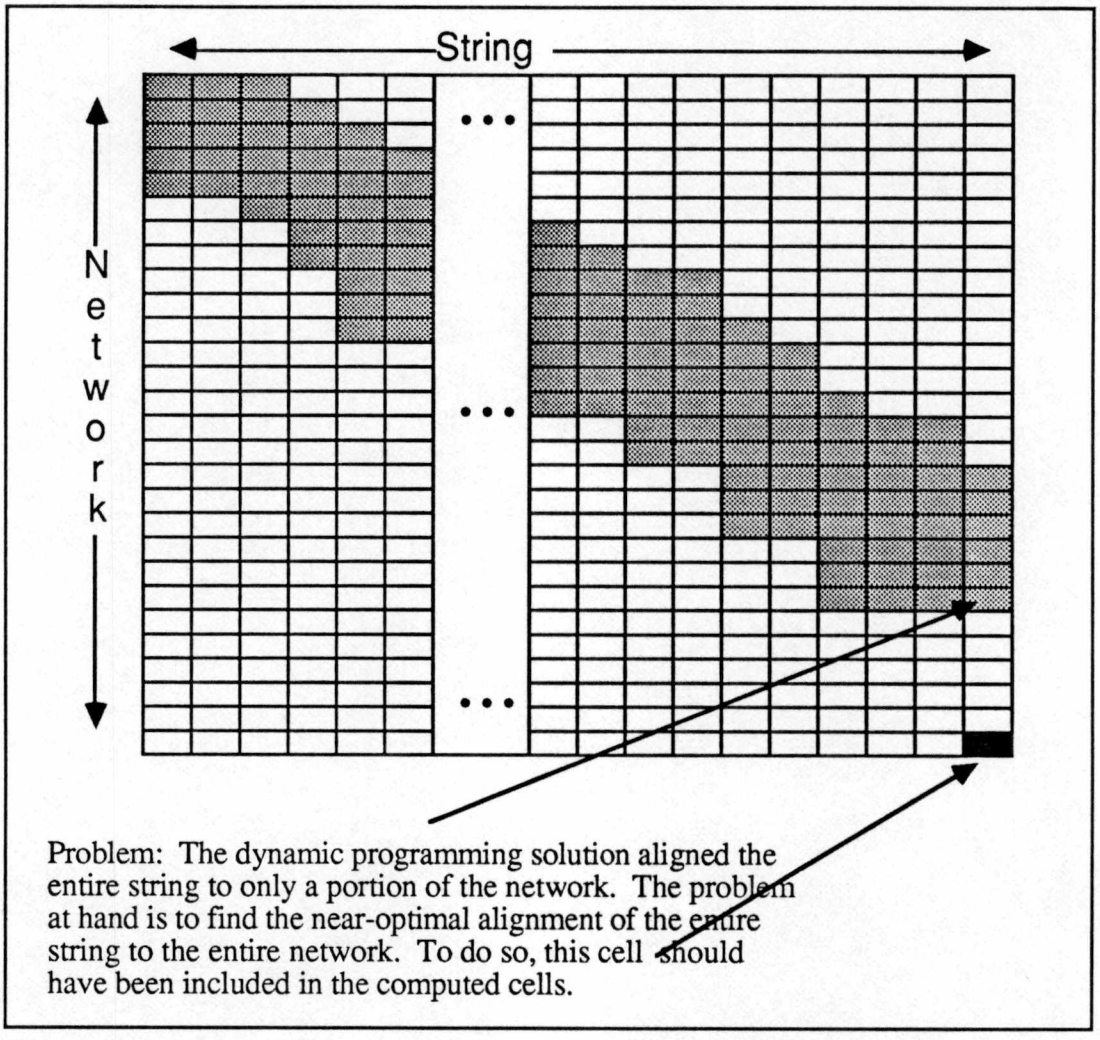


Figure 4.6. Problem of Channel 'Missing' the Solution.

4.5. Channel Constraint Summary.

The channel constraint algorithm is a heuristic which enables the matrix computation step to ignore large portions of the matrix and thus lessen the computational cost of the dynamic programming algorithm. It functions by accepting a supplied parameter value called the channel constraint and by then forcing the difference of cumulative number of insertions and deletions in every editing trace to be less than or equal to the parameter value. Although the overhead of guiding the channel near the main diagonal of the matrix is great, it will be shown in Chapter 6 that use of the channel constraint can subtract almost 50% from the amount of time required by the dynamic programming algorithm.

CHAPTER 5

THE IFC IMPLEMENTATION

5.1. Introduction.

Several versions of the inference algorithm have been implemented during the course of this research effort. This chapter will begin with a brief history of the implementations and will then describe at a fairly high level the data structures and algorithms of the IFC code.

5.2. History of IFC Implementations.

The research reported here began during September, 1985. Although a version of IFC (produced by Drs. Mike Thomason and Erik Granum) had already been implemented under the UNIX operating system on a VAX 8200, a version was desired that would run efficiently on an IBM PC AT using the Turbo Pascal compiler and programming environment. Initially, this work was to be done in conjunction with researchers at Johns Hopkins University who were interested in building structural models of EKG wave patterns for analysis and monitoring of various human heart conditions. Although the PC version was completed during the winter of 1986, the strict memory constraints imposed by the DOS operating system, the relatively slow speed of the 16-bit, 6 Mhz PC AT and problems mapping EKG wave patterns into strings caused the research to take a different focus.

It was decided that the basic algorithm of the dynamic programming computation should be altered so that a 'channel' would be computed in

the cost matrix and thus large portions of the matrix eliminated from the computation. During the summer of 1986, a PC version was finished in which the channel constraint was in place. Unfortunately, the notion of restricting the view of the matrix using the channel_begin and channel_end elements (described in Chapter 4) was not incorporated and thus the time savings realized were rather minimal -- at best, savings of 10 - 30% were noted. A similar version of this code, without a graphics interface, was later implemented on a Texas Instruments PC XT compatible. Another version was ported to an Apple MacIntosh Plus.

These PC versions of the IFC code were very limited because of the memory constraints and speed of the machines. Problems of reasonable size could not be solved. Thus, it was decided in late 1986 to take the implementation back to the VAX series of machines. In early 1987, work began on a VAX 8200 UNIX machine using the C programming language. For various reasons this work was abandoned and the effort moved to a much more powerful VAX 8700 using the VMS operating system and the C programming language. This version was completed during the late summer of 1987 and is the version described by this effort.

The results of this work were presented at a conference on pattern recognition in Aalborg, Denmark in mid-December, 1987. While in Aalborg and during the month following the trip, some important modifications to the code were made which produced rather dramatic speedups in execution time. This revised implementation was also ported to the UNIX environment. Some preliminary results are reported in Chapter 6.

Work is currently underway towards producing a version of the IFC code which will run on a parallel processing machine under UNIX. Early speedup results have been highly encouraging but the implementation requires a great deal more work and thus the results are not reported in Chapter 6.

5.3. The Implementation.

It was noted earlier that the version of the IFC code on which most of the reported results are based has been revised. The description of the data structures and the algorithms which follows is based on the older version but wherever there are important differences between the two versions, they will be reported. It should be noted that the major revision made in the new code is the elimination of redundant calculations in the matrix computation. The elimination required additional data elements in some of the data structures and subtle changes in some of the routines. These will be noted as encountered.

5.3.1. Data Structures.

There are four primary data structures used in the IFC implementation--the network, the dynamic programming matrix, a string buffer and an editing trace buffer. These are described as follows.

5.3.1.1. The Network.

The network data structure is the representation of the Markov network. It is represented in the IFC implementation as a data structure

with three elements. One of the elements is a large array of pointer elements each of which points to a node structure. Also contained is an integer which points to the final node in the network and another integer which points to the next available element in the array of pointer elements. Each node structure is dynamically allocated and is comprised of the following data elements:

symbol This is the output symbol for the given node. For a live node, this value is drawn from the set of observable symbols. For a bookkeeping node, it contains the value of the unobservable symbol.

visits This is the frequency count on the given node -- i.e., the number of times this node has been visited in traversals through the network during the inference process.

input_arcs This is a pointer to a chain of input arcs. Each of the elements of the chain consists of:

- **node** the node from which the input arc travelled.
- **freq** the number of times the arc has been traversed.
- **next_arc** points to the next arc on the chain.

output_arcs This is a pointer to a chain of output arcs. Each of the elements of the chain consists of:

- **node** the node to which the output arc travels.
- **freq** the number of times the arc has been traversed.
- **next_arc** points to the next arc on the chain.

next Points to the next node structure to be processed during the dynamic programming step.

previous Points to the previous node structure to be processed during the dynamic programming step.

The network node data structure was modified for the new IFC version. A quick review of the five operations used in calculating the probability value for each cell of the matrix shows that each of the calculations requires an addition of a calculated number to a probability value from a neighboring cell. It was not realized until recently that these calculated numbers, different for each of the five operations, only had to be computed once per node of the network and thus per row of the matrix. Earlier versions of the code required these values to be computed for every cell of the matrix. By calculating these numbers once and storing them with the network structure, a large number of calculations were eliminated. The following describes the additional data elements associated with each node structure to accommodate the storage of these calculated values.

match_num The result of the match calculation between the current node, i, and its predecessor node, i-1. It is set equal to the value of the following calculation:

$$\log\left(\frac{f_{i-1,i} + 1}{f_{i-1} + 1}\right)$$

subst_num The result of the substitution calculation between the current node, i, and its predecessor node, i-1. It is set equal to the value of the following calculation:

$$\log\left(\frac{f_{i-1,i} + 1}{f_{i-1} + 1}\right) + \log\left(\frac{1}{f_i + 1}\right)$$

delete_num The result of the deletion calculation between the current node, i, and its predecessor node, i-1. It is set equal to the value of the following calculation:

$$\log\left(\frac{f_{i-1,i} + 1}{f_{i-1} + 1}\right) + \log\left(\frac{1}{f_i + 1}\right)$$

insert_num The result of the insertion calculation. It is set equal to the value of the following calculation:

$$\log\left(\frac{1}{f_i + 1}\right)$$

Because the bookkeeping nodes do not necessarily have a single predecessor node, the calculated value associated with the bookkeeping node 'operation' must be attached to the input arcs of the bookkeeping node. Thus, the following data element is added to the input arcs data type.

empty_op_num The result of the deletion calculation between the current node, i, and its predecessor node, i-1. It is set equal to the value of the following calculation:

$$\log\left(\frac{f_{i-1,i} + 1}{f_{i-1} + 1}\right)$$

5.3.1.2. The Dynamic Programming Cost Matrix.

The cost matrix structure is an array of pointers (the same size as the array of pointers to node structures) each of which points to a structure consisting of: (1) an array of pointers each of which points to a cell in the

matrix, (2) the channel_begin element, and (3) the channel_end element. Each of the cell structures is comprised of the following data elements.

prob	The probability value representing the maximum probability with which the modified network can generate the prefix substring.
operation	The operation which produced the probability value.
pnode	The predecessor node from which the cell's calculation is based.
inserts	The cumulative number of insertions on the editing trace which ends with the cell.
deletes	The cumulative number of deletions on the editing trace which ends with the cell.

It should be noted that the current implementation requires two levels of addressing indirection (to the row and then to the cell). This allows the dynamic allocation of only as many rows and cell elements as are absolutely required. It requires, however, more time to resolve the addresses. If enough storage is available to allow static allocation, the modification of the code to address a statically allocated matrix might prove beneficial.

5.3.1.3. The String Buffer.

The string buffer is used for holding the sample strings used in the inference and in the classification algorithms. It is a data structure consisting of three elements: (1) a large buffer of characters (a contiguous

array of characters), (2) an array of pointers pointing to strings in the buffer, and (3) an integer indicating the number of strings in the buffer.

5.3.1.4. The Editing Trace Buffer.

After the inference algorithm completes computation of the cost matrix, it must obtain the editing trace information from the matrix. This trace information is read into the trace buffer which is simply an array of elements each of which consists of: (1) an operation code (indicating the operation to apply to the given symbol in the string) and (2) an identifier of the node with which the editing operation is to be applied.

5.3.2. Code Structure/Algorithms.

Before discussing the routines and algorithms in the code, it may be helpful to look at a couple of command level invocations of the IFC application. There are two basic applications which can be run with IFC: (1) inference -- a new network is inferred from a sample set of strings, and (2) classification -- an existing network structure is used to assign probability of generation values to a set of strings. These values can then be compared to values generated by other network structures (for the same set of strings) to obtain classification of each string.

A typical invocation of the inference application is:

```
ifc i/samples=ld1/output=ld1.network/channel=3
```

where the 'i' indicates inference mode, 'ld1' is the name of the file containing the sample strings, 'ld1.network' is the name of the file in

which the network will be stored and '3' is the size of the channel_constraint parameter.

An invocation of the classification application is:

```
ifc c/samples=td1/input=ld1.network/results=td1.ld1/channel=2
```

where the 'c' indicates classification mode, 'td1' is the name of the file containing the sample strings, 'ld1.network' is the name of the file from which the network will be loaded, 'td1.ld1' is the name of the file in which the probability values will be stored and '2' is the size of the channel_constraint parameter.

The remainder of this chapter will focus on the important routines and algorithms of the IFC code.

5.3.2.1. The Main Routine.

The main routine is, of course, the driver for the IFC application. The following pseudocode describes the structure of this routine:

```
Main()
    perform necessary initializations (set flags, open files, etc.)

    process the user-supplied command line options
        if errors found in command line options  exit program

    fill string buffer from the string samples file
        if no strings found exit program

    if the mode_indicator = 'I'
        allocate and initialize the cost matrix structure
        perform the inference() routine

    else if the mode_indicator = 'C'
        load the network from the network input file
        allocate and initialize the cost matrix structure
        perform the classify() routine

    if user supplied a file name in which to store the network
        perform the save_network() routine

    if user indicated a desire for network analysis
        perform the network_analysis() routines
        print the analysis information to the analysis file

    close files and exit program
```

The main routine does little other than call routines to perform necessary work. The non-trivial routines in this implementation are those called by the Inference() and Classify() routines.

5.3.2.2. The Inference Routine.

This routine performs all the work needed to infer a network from a sample set of strings. The following pseudocode outlines the structure of the routine:

```
inference()
    Call initial_network() to build the initial linear Markov network from the first
string
    in the sample set

    While more strings in the sample set do
    |
    |   set the constraint equal to the user-supplied channel_constraint
    |
    |   Until a valid solution is found do
    |   |   if constraint = 0    { indicating that channel is not to be used -- entire
    |   |   |                       matrix is to be computed }
    |   |   |   call the ready_matrix_b() routine
    |   |   |                   { compute the first row and first column of matrix }
    |   |   |   call the compute_matrix_b() routine
    |   |   |                   { computes the entire matrix }
    |   |   |   indicate that a valid solution was found
    |   |
    |   |   else { constraint is > 0 }
    |   |   |   call the ready_matrix() routine
    |   |   |                   { compute the first row and first column of matrix }
    |   |   |   call the compute_matrix() routine
    |   |   |                   { computes the appropriate channel of the matrix }
    |   |   |   if last cell in matrix was not reached by the computation
    |   |   |           indicate that a valid solution was not found
    |   |   |           zero out the matrix
    |   |   |           increment the constraint value by 1
    |   |   |   else
    |   |   |           indicate that a valid solution was found
    |
    |   call get_trace() to obtain the editing trace information from the matrix
    |
    |   call modify_network() to apply the editing trace to the network
    |
    |   call zero_matrix() to zero out the cost matrix
```

The inference routine first builds a trivial network from the first string in the sample set. It then sequentially incorporates each string into the current network structure until there are no more strings. Since the

cells in the first row and first column of the matrix have no neighboring cells and thus cannot be handled by the normal cell computation, they are computed by a special routine before the compute_matrix routine is called. If the user indicated that the channel constraint was not to be used, then the entire matrix is computed. Otherwise, the channel is computed in the matrix. If the last cell is not reached in the computation, the constraint value is incremented and the matrix is computed again.

Once a valid solution to the problem is found, the editing trace information is gathered from the matrix and is then applied to the current network. The matrix is then re-initialized and the process begins again with the next string.

5.3.2.3. The Classify Routine.

This routine performs all the work needed to produce probability values for each string in a sample set of strings by aligning those strings with a pre-existing network structure. The following pseudocode outlines the structure of the routine:

```
classify()
    While more strings in the sample set do
    |
    |   set the constraint equal to the user-supplied channel_constraint
    |
    |   Until a valid solution is found do
    |   |   if constraint = 0     { indicating that channel is not to be used -- entire
    |   |                           matrix is to be computed }
    |   |   |   call the ready_matrix_b() routine
    |   |   |                     { compute the first row and first column of matrix }
    |   |   |   call the compute_matrix_b() routine
    |   |   |                     { computes the entire matrix }
    |   |   |   indicate that a valid solution was found
    |   |
    |   |   else  { constraint is > 0 }
    |   |   |   call the ready_matrix() routine
    |   |   |                     { compute the first row and first column of matrix }
    |   |   |   call the compute_matrix() routine
    |   |   |                     { computes the appropriate channel of the matrix }
    |   |   |   if last cell in matrix was not reached by the computation
    |   |   |           indicate that a valid solution was not found
    |   |   |           zero out the matrix
    |   |   |           increment the constraint value by 1
    |   |   |   else
    |   |   |           indicate that a valid solution was found
    |
    |   call zero_matrix() to zero out the cost matrix
```

The classify routine aligns each string with the network structure. The computation is identical to the inference() routine except that there is no need to gather the trace information and no need to update/modify the network.

5.3.2.4. The Initial Network Routine.

This routine builds the initial Markov network from the first string in the sample set. The following pseudocode describes the algorithm:

```
initial_network()
    allocate a new node in the network (the first node)
    initialize the new node as a bookkeeping node

    while more symbols in the string do
    |   allocate a new node
    |   initialize the new node as outputting the current symbol

    allocate a new node in the network (the final node)
    initialize the new node as a bookkeeping node
```

5.3.2.5. The Ready Matrix Routines.

The purpose of these two routines is to compute the first column and the first row of the dynamic programming matrix. Computing the first column of the matrix solves the problem of mapping the first symbol of the string (always assumed to be the empty string symbol) into the network. Computing the first row of the matrix corresponds to mapping the string into the first node of the network (a bookkeeping node). With the first row and first column computed, the matrix has 'neighbors' ready for the normal dynamic programming computation. Ready_matrix is called if the user is utilizing the channel constraint. Otherwise, Ready_matrix_b is called.

```
ready_matrix()
    Calculate the first cell (upper left hand corner) of the matrix

    { compute the first column of the dynamic programming matrix }

    while more nodes in the network do
    |   if current node is a bookkeeping node
    |       look at all input arcs and compute 'best' bookkeeping operation
    |   else { if current node is a 'live' node }
    |       compute 'deletion' operation from predecessor node

    { compute the first row of the dynamic programming matrix }

    while number_of_consecutive_insertions <= channel_constraint do
        compute insertion operation

    compute the channel_begin and channel_end values for all successor nodes of
    (the starting node)
```

```
ready_matrix_b()
    Calculate the first cell (upper left hand corner) of the matrix

    { compute the first column of the dynamic programming matrix }

    while more nodes in the network do
    |   if current node is a bookkeeping node
    |       look at all input arcs and compute 'best' bookkeeping operation
    |   else { if current node is a 'live' node }
    |       compute 'deletion' operation from predecessor node

    { compute the first row of the dynamic programming matrix }

    for (i = 1 to length_of_current_string) do
        compute insertion operation
```

5.3.2.6. The Compute Matrix Routine.

Most of the computational time spent by the IFC application is spent in this routine (or in the corresponding compute_matrix_b routine). The compute_matrix routine is called if the user supplied a valid channel constraint value. The compute_matrix_b routine is used if the user does not wish to use the channel constraint. The following pseudocode describes the algorithm:

```
while more nodes to process in network do
    if current node is a bookkeeping node then do

        for i = channel_begin to channel_end do

            look at appropriate cell of each node immediately preceding
                    current node (via an input arc) to find the one with
                    highest probability of travel to current node
            assign values to cell[i] of current node

            for all nodes connected to current node via an output arc
                    calculate the channel begin and channel_end values

    else do
        for i = channel_begin to channel_end do
            compute insertion operation for cell[i] (if possible)
                    assign results to best_operation
            compute deletion operation for cell[i] (if possible)
                    if better results than best_operation assign results to
                            best_operation
            compute match/substitution operations for cell[i] (if possible)
                    if better results than best_operation assign results to
                            best_operation

            assign best_operation results to cell[i]

            if best_operation was an INSERTION or DELETION
                    increment inserts/deletes count appropriately

            if last cell computed was not on the boundary of the channel,
                    (i.e., if more insertions / deletions allowed)
                    compute more cells on the current row of the matrix

            for all nodes connected to current node via an output arc,
                    calculate the channel begin and channel_end values
```

5.3.2.7. The Compute Matrix B Routine.

This routine is called if the user did not request use of the channel constraint. The following pseudocode describes the algorithm:

```
while more nodes to process in network do
    if current node is a bookkeeping node then do

        for i = 1 to length_of_current_string do

            look at all nodes immediately preceding current node (via an
                input arc) to find the one with highest probability of
                travel to current node
            assign values to cell[i] of current node

            for all nodes connected to current node via an output arc
                calculate the channel begin and channel_end values

    else do
        for i = 1 to length_of_current_string do

            compute insertion operation for cell[i]
                assign results to best_operation
            compute deletion operation for cell[i]
                if better results than best_operation assign results to
                    best_operation
            compute match/substitution operations for cell[i]
                if better results than best_operation assign results to
                    best_operation

            assign best_operation results to cell[i]
```

5.3.2.8. The Get_Trace Routine.

The get_trace routine gathers the editing trace information from the cost matrix and places the information in the trace buffer. It is described by the following pseudocode:

```
get_trace()
    Set current node equal to the very last node in the network (and thus the last
        row in the matrix)
    Set i = 0

    until first node in the network (and first row in matrix) is reached do
    |   trace[i].node = current node
    |   trace[i].operation = cell[column].operation
    |
    |   current_node = cell[column].previous_node
    |
    |   if trace[i].operation = MATCH, INSERTION, or SUBSTITUTION
    |       column = column - 1
    |
    |   i = i + 1
```

5.3.2.9. The Modify Network Routine.

The modify_network routine is responsible for applying the optimal editing trace information to the network, thus incorporating the current string into the network structure. It moves through the trace buffer element by element applying editing operations with the current string symbol to the network. The following pseudocode describes the algorithm:

```
modify_network()
    Set curr_node equal to the first node in the network
    Set string_index equal to 0 { to point to first symbol in string }

    while more elements in the trace_buffer do
        case current trace_element.operation of

            bookkeeping node operation:
                perform match operation on current node from the predecessor node

            match operation:
                perform match operation on current node from the predecessor node
                increment the string_index by 1

            substitution operation:
                perform substitution operation from the predecessor node with
                        current string symbol
                increment the string_index by 1

            insertion operation:
                perform insertion operation with current string symbol
                increment the string_index by 1

            deletion operation:
                perform the deletion operation from the predecessor node
```

5.4. Implementation Summary.

The IFC implementation is comprised of approximately 40 routines which are in turn comprised of a total of 5,000 lines of C code (including some documentation). Although the code is written in as generic a C dialect as possible, there are some calls made to VMS library clock routines

which make the porting to other environments (such as UNIX) a small amount of trouble. Although the code is generally robust, there is the potential for trouble if insufficient memory is available or if the I/O routines attempt to write to a full device. The code has been executed in literally thousands of experiments and has run satisfactorily. One potential weakness of the code is its use of arrays of pointers to address dynamically allocated data structures. Although this does allow the code to consume only as much memory as needed, it introduces addressing indirection which costs time in this very memory-busy application.

CHAPTER 6

RESULTS

The most important result of this project is the time savings realized by using the channel constraint. These time savings are discussed in section 6.1. Also important are the results from classification experiments in which the Markov networks are used to classify/recognize strings. These results are of particular interest since the channel constraint is a heuristic which would be expected to give worse classification results than earlier IFC implementations which guaranteed optimal string-to-network alignments. Not only has this proven **not** to be the case, but use of the channel constraint almost always gives better classification results. These results are discussed in section 6.2. Some thoughts on why the better classification results have been realized are presented in section 6.2.1. Some problems are presented in 6.2.2.

6.1. Timing Results

Figure 6.1 gives the average number of CPU seconds required by the **compute matrix** routines for different values of the channel constraint parameter. To obtain these values, 6 experiments were run for each of the 22 G-banded human chromosomes. Each experiment involved computing the alignment of 100 sample strings of a chromosome data set with a network which had earlier been inferred from the same set of strings. Each of the 6 experiments used a different channel constraint parameter -- from

0 (indicating that the channel constraint was not to be used) to 5 (meaning the absolute difference between insertions and deletions was kept less than or equal to 5). The amount of CPU time required by the compute_matrix or compute_matrix_b routine was reported for each string-to-network alignment. These values were averaged for the set of 100 strings. These values were then averaged over the 22 different chromosome types. The result is the average amount of CPU time required for different values of the channel constraint parameter. The version of the IFC code used in these experiments was the earlier one in which the editing operation values were re-computed at each cell of the matrix. The cost of computing each cell was relatively expensive. Use of the channel constraint, by eliminating computation of many of these 'expensive' cells, gives fairly good time savings.

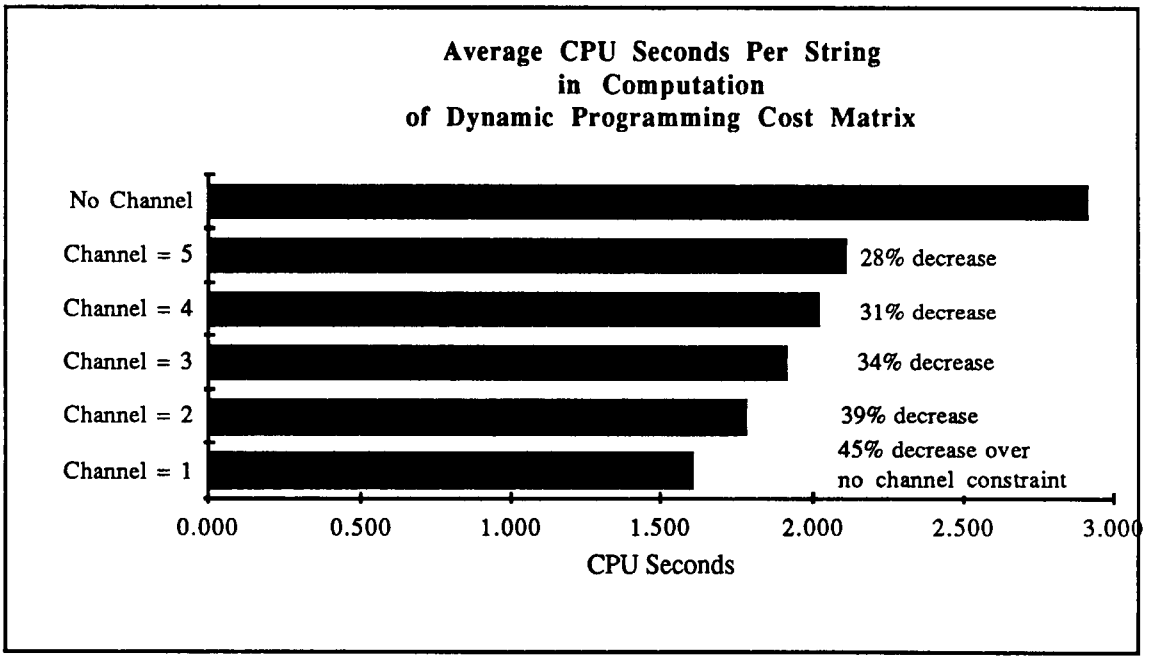


Figure 6.1 -- Timing Results from First Version of IFC Implementation.

The results in Figure 6.2 were obtained by running experiments similar to those run for Figure 6.1. However, the newer version of the IFC code (with the redundant cell computations eliminated) was used to obtain the timing information. Because the computation of each cell of the matrix was far less expensive, the overhead involved in guiding the channel plays a much greater role. The result is that even for the smallest values of the channel constraint, it actually requires more time to compute a channel through the matrix than it does to compute the entire matrix. The implication is that the greater the cost of computing each stage of the dynamic programming problem, the more likely the channel constraint is to give beneficial time savings.

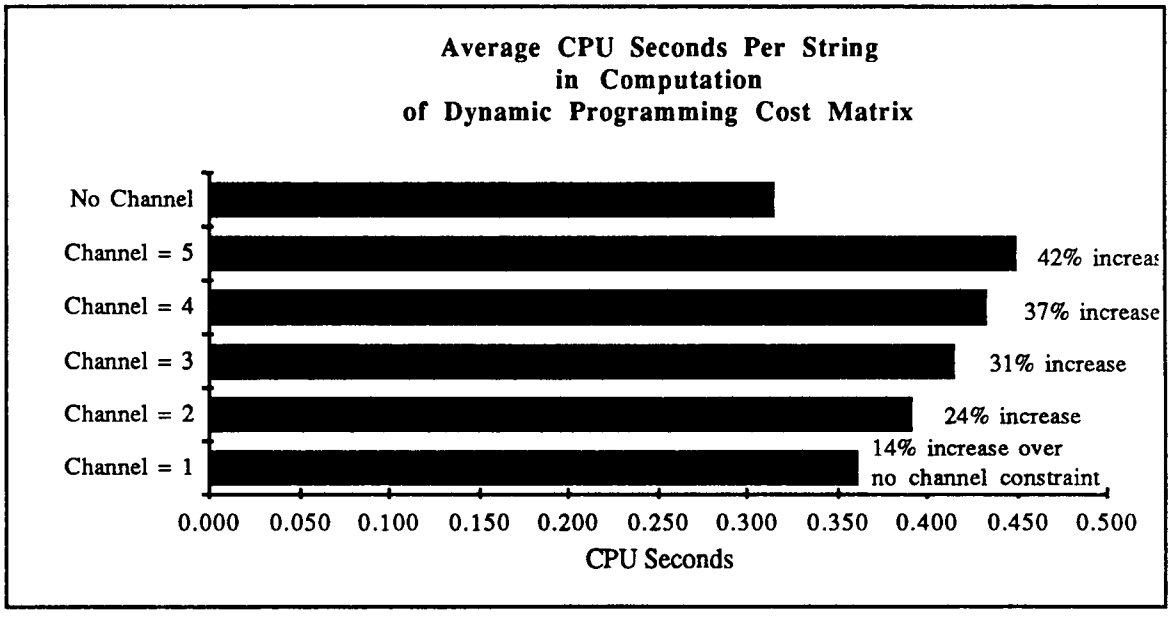


Figure 6.2. Timing Results from New Version of IFC Implementation.

Figure 6.3 combines information from Figures 6.1 and 6.2 to present the time savings realized by eliminating the redundant calculations from the dynamic programming problem. When the channel constraint is not used (parameter value = 0), the speedup is 924%.

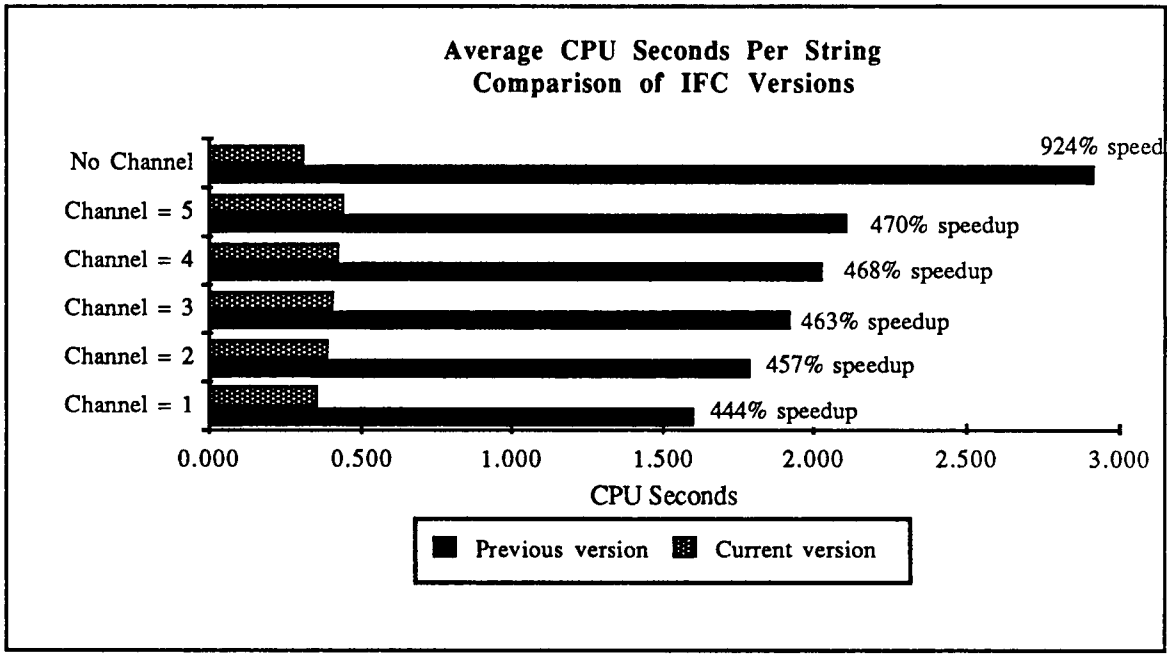


Figure 6.3. Timing Comparisons Between Two Recent IFC Implementations.

6.2. Classification Results.

Classification experiments were run with all 22 sets of G-banded human chromosome data. For each type of chromosome, two sets of 100 strings were used -- a learning data set (used to infer the Markov network) and a test data set. The following describes how the classification experiments were conducted.

Before running the experiments, networks were inferred from each of the 22 learning data sets. The classification experiments were run in groups corresponding to the grouping of human chromosome types shown in Figure 6.4. Thus, for the Group A experiments, every string in the type 1 data set was aligned with the type 1, type 2 and type 3 network. The result of each alignment was a probability value indicating the probability with which the given network could generate the string. Thus, for every Group A string, three probability values were generated. Classification of the string as a type 1, 2 or 3 chromosome was based on which network yielded the highest probability. This process was repeated for every type 1 string in the learning data set, then for the two other Group A learning data sets, and then for all other learning data sets in Groups B - G. Finally, the same set of experiments was run with the 22 test data sets. All of the experiments were repeated six times -- once for a channel constraint parameter of 0 (indicating that the channel constraint was not to be used) and then once each for the five parameter values, 1, 2, 3, 4, 5.

Group A	Types 1, 2, 3
Group B	Types 4, 5
Group C	Types 6, 7, 8, 9, 10, 11, 12
Group D	Types 13, 14, 15
Group E	Types 16, 17, 18
Group F	Types 19, 20
Group G	Types 21, 22

Figure 6.4. Chromosome Groupings.

It would be reasonable to expect that since the channel constraint is a heuristic which yields sub-optimal string-to-network alignments, the classification experiments would yield fewer correct classifications. Also, since decreasing the size of the channel constraint decreases the number of cells computed in the matrix, it would be expected that lower values of the constraint parameter would yield even lower correct classifications. As the data represented in Figure 6.5 and the following graphic figures show, however, both expectations prove false.

Data Type:	LD		TD		LD		TD		LD		TD		LD		TD		LD		TD		LD		TD	
Channel:	1		1		2		2		3		3		4		4		5		5		Mean		Mean	
Chrom 1	99	990	99	993	99	989	99	992	99	989	99	992	99	969	99	991	99	988	99	990	99	981	97	971
Chrom 2	100	999	100	993	100	999	98	988	100	999	98	984	100	997	98	982	100	996	98	981	98	983	97	953
Chrom 3	100	1000	99	999	100	1000	99	990	100	1000	99	991	100	1000	99	992	100	1000	100	997	100	1000	100	999
Chrom 4	99	989	97	950	99	988	97	956	99	987	96	932	99	986	93	922	99	986	94	920	99	986	94	919
Chrom 5	98	983	99	964	98	981	99	981	98	980	98	977	98	979	98	978	98	979	98	978	98	979	96	976
Chrom 6	87	954	84	858	83	953	85	826	91	910	79	799	78	884	78	770	83	860	77	743	78	735	59	569
Chrom 7	95	942	93	919	94	927	91	907	92	920	91	905	92	917	90	898	91	910	90	894	91	900	90	879
Chrom 8	98	959	98	924	97	947	92	893	97	945	96	899	94	937	87	841	94	933	87	851	96	921	86	841
Chrom 9	97	960	97	862	97	955	97	859	96	946	97	854	94	938	86	847	94	939	85	840	94	933	86	833
Chrom 10	99	989	97	948	99	981	97	946	99	980	96	947	99	985	95	943	99	984	95	940	99	983	95	940
Chrom 11	99	974	95	938	99	974	94	935	99	973	95	935	99	971	94	931	99	971	94	930	99	971	94	928
Chrom 12	99	981	95	934	98	982	94	931	99	981	91	915	98	979	91	908	98	973	91	907	98	970	91	905
Chrom 13	100	997	97	966	100	994	97	964	100	995	97	961	100	993	97	960	100	994	97	959	100	994	97	957
Chrom 14	98	982	98	964	97	976	97	953	97	975	97	952	97	974	97	951	97	973	97	949	97	973	97	945
Chrom 15	100	999	98	978	100	997	98	977	100	997	98	979	100	997	98	979	100	997	98	979	100	997	98	979
Chrom 16	97	971	98	965	97	965	98	967	97	963	97	957	97	962	96	951	95	952	96	950	95	952	96	948
Chrom 17	99	968	91	910	98	957	91	905	97	949	90	897	96	936	89	886	93	928	88	876	95	936	88	874
Chrom 18	99	984	97	955	99	984	97	957	99	983	96	956	99	983	96	955	99	983	96	955	99	983	96	955
Chrom 19	96	945	88	889	96	943	86	874	96	943	86	877	96	943	86	879	96	943	86	879	96	943	86	879
Chrom 20	95	941	91	900	95	941	91	899	95	940	92	901	93	940	92	901	93	939	92	901	95	939	92	901
Chrom 21	98	968	95	934	98	969	96	957	98	969	96	958	98	969	96	958	98	969	96	958	98	969	96	958
Chrom 22	100	996	95	930	100	992	92	898	100	990	93	896	99	983	93	886	99	980	92	882	99	980	92	882
Ave.	98.3	976.4	94.7	940.6	97.9	971.9	94.0	933.8	97.4	967.4	93.5	928.4	97.2	965.6	93.2	925.0	94.9	942.6	93.0	921.8	96.3	955.3	92.0	909.7
Std Dev.	1.5	18.9	4.2	39.7	1.9	22.1	4.4	45.1	2.4	25.1	4.9	50.1	2.9	29.5	5.1	54.3	3.5	33.5	5.5	59.4	4.6	31.0	8.3	86.9
Variance	2.4	372.5	18.3	1649.8	3.9	510.1	20.6	2127.4	6.1	662.5	25.5	2634.4	8.8	909.1	27.4	3067.2	12.7	1172.7	31.0	3700.7	22.4	2720.9	72.3	7901.5

Figure 6.5. Classification Experiment Results Summary.

Figure 6.6 shows the correct classification percentages averaged over all strings in the 22 learning data sets. Figure 6.7 is a similar graph but is based on experiments using the test data sets. Since the strings in the learning data sets were used to infer the Markov networks, the experiments expectedly yield a higher percentage of correct classifications than those strings in the test data sets. Both graphs show a relatively high percentage of correct classifications. Each also shows, however, that decreasing the size of the channel constraint parameter increases the correct classification percentage.

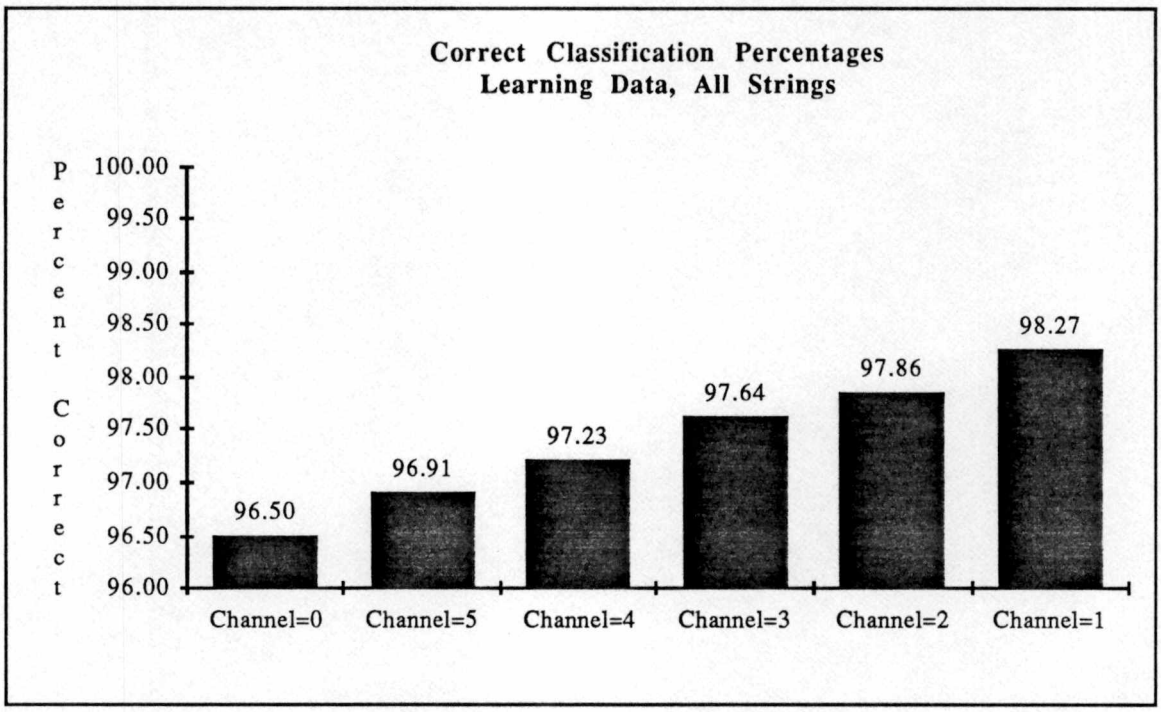


Figure 6.6. Correct Classification Percentages -- Learning Data Sets.

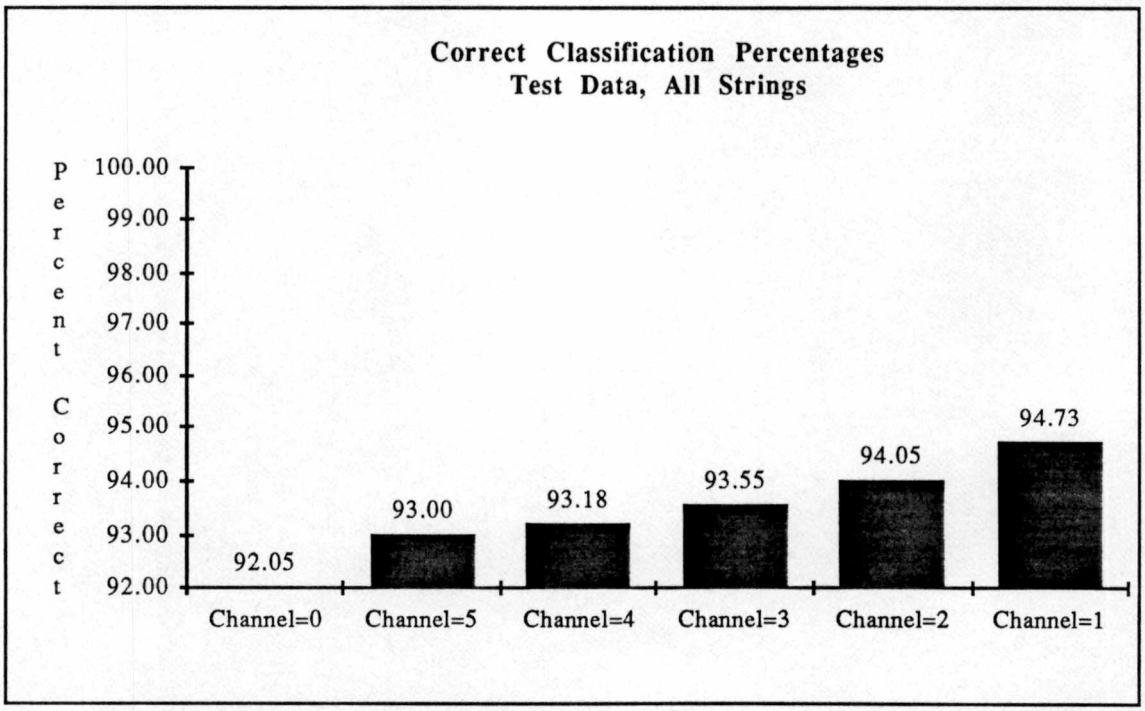


Figure 6.7. Correct Classification Percentages -- Test Data Sets.

Figure 6.8 shows scaled confidence values averaged over all strings in the 22 learning data sets. The scaled confidence value for a given string is obtained by: (1) summing its network probability values (obtained by aligning the string with each network in its corresponding chromosome 'Group'), (2) dividing the highest probability value by this sum, (3) multiplying this value by 1000. This number represents, in a sense, the confidence with which that string was identified by a network. For example, a value of 1000 would indicate that one network yielded a much higher probability value than the other networks in that group. Lower values would indicate that another network (or networks) yielded a probability value(s) closer to the highest probability value -- thus the confidence in the classification would not be as great. Figure 6.9 shows the same graph for the strings in the test data sets. Both graphs show higher confidence values for smaller values of the channel constraint.

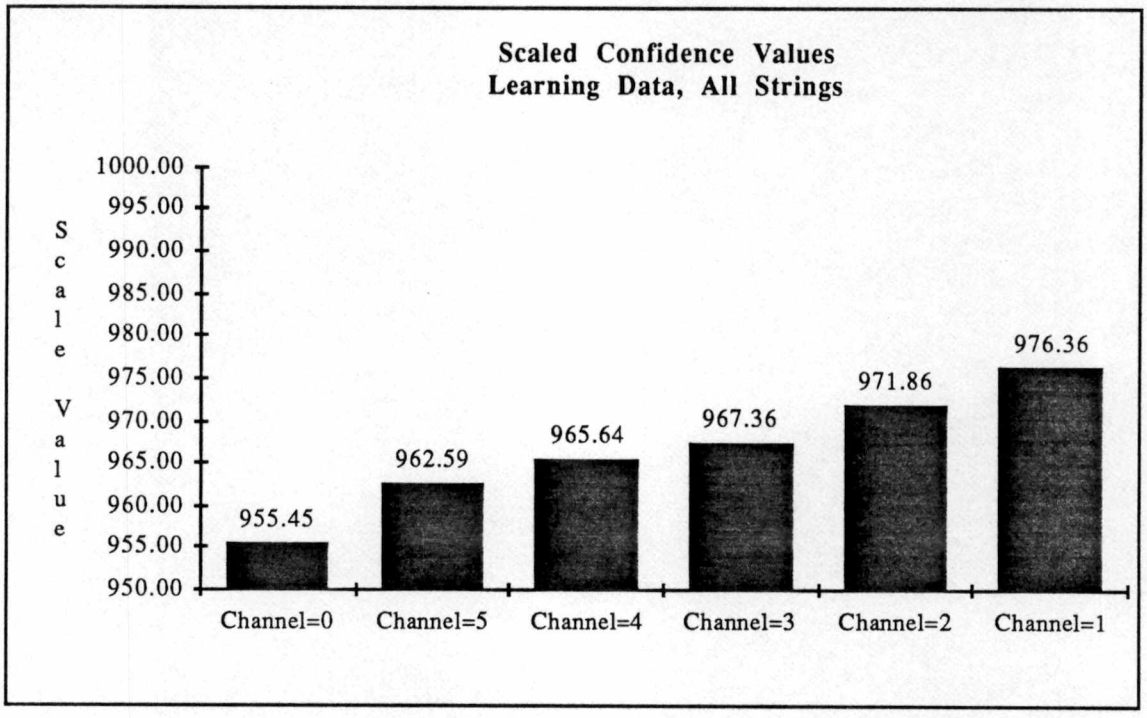


Figure 6.8. Scaled Confidence Values -- Learning Data Sets.

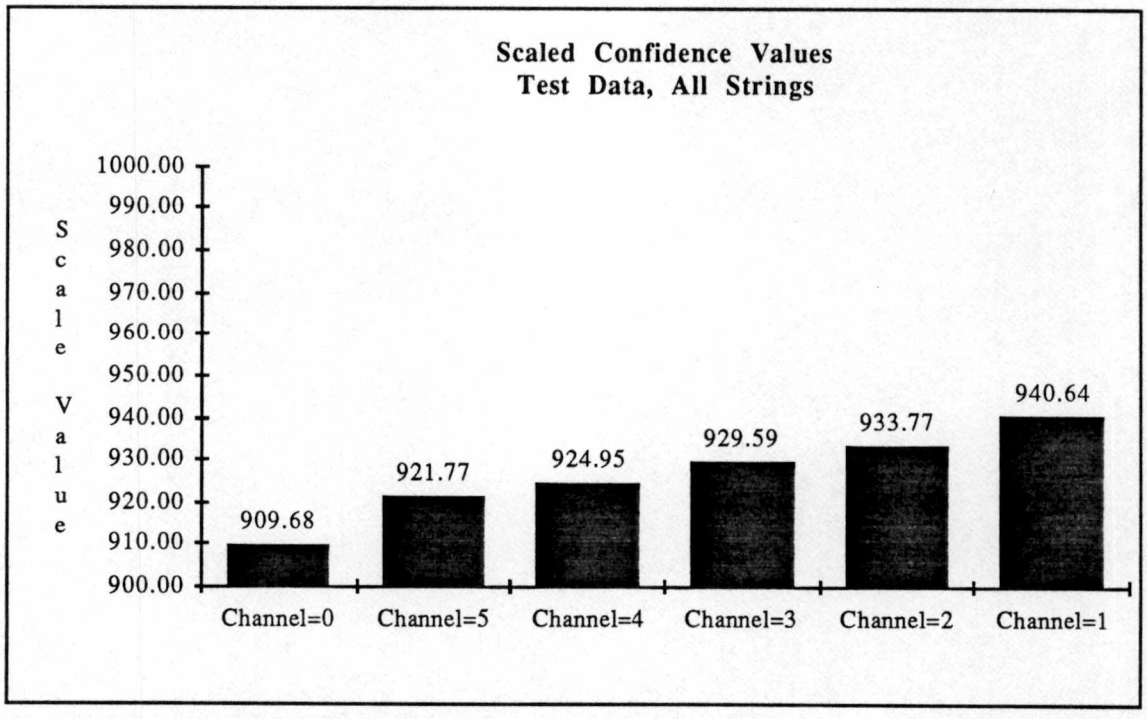


Figure 6.9. Scaled Confidence Values -- Test Data Sets.

6.2.1. Speculation on Results.

The classification experiments just described yield results entirely opposite to those expected. Much work remains to be done towards discovering exactly why the experiments using smaller values of the channel constraint parameter yield higher classification results and correspondingly high levels of confidence. However, some preliminary work has been done which has thus far confirmed initial suspicions.

When a string is aligned with a network using the full cost matrix, the dynamic programming procedure has full freedom to map the string into the network using any series of editing operations. The resulting alignment is, of course, optimal. When the channel constraint is used, the dynamic programming procedure does not have this full freedom -- it must restrict its editing trace to one in which the difference between number of insertions and number of deletions falls within the supplied parameter value. It thus confines its mapping to those cells computed in the channel. A string which maps well into the network will still find an optimal or near-optimal alignment and thus will generate a relatively high probability value. A string that does not map easily into the network will either (1) not find a valid mapping at all (this happens when the channel does not reach the last cell in the matrix) and thus will yield a probability value of 0, or (2) will generate an editing trace which includes a relatively high number of expensive operations (insertions and deletions) and thus will yield a low probability value -- thus, strings which do not 'fit' are penalized more heavily than in the procedure of the full matrix computation. Thus, strings which map best into a given network are

those which are of the same type as the network -- this has proven especially true with the procedure when using the channel constraint.

Figure 6.10 illustrates in more detail the results. The figure presents all scaled confidence values generated for chromosome type 6 classification experiments. The first six sub-tables represent the scaled confidence values generated by each of the 7 networks (chromosome types 6 - 12) with which each string of the learning data set was aligned. The first table was generated when the channel constraint was not used (the entire matrix was computed). The tables following represent experiments in which successively smaller values of the channel constraint parameter were used. The last sub-table summarizes the confidence values generated by the type 6 network.



Figure 6.10. Scaled Confidence Values for Chromosome Type 6 Experiments

Confidence values for any string in the learning set can be traced along a row in the first six sub-tables. The confidence values are observed to remain the same or to increase for each string as the channel constraint value decreases. This does not mean that the corresponding probability values increase -- indeed, if the probability values were included they would be shown to decrease as the channel constraint parameter decreases (this is necessarily true since the full matrix computation finds an optimal mapping which cannot be improved upon). But relative to the probability value produced by the type 6 network, the type 7 - 12 networks generate smaller probability values. This accounts for both the increased percentage of correct classifications and for the increased confidence in those classifications.

6.2.2. Problems.

Until the above described phenomena has been studied further, it is best to consider the channel constraint more as a potential CPU time-saver (depending upon the cost of the stages of the given dynamic programming problem) rather than as a technique for obtaining better classification results. However, the results are consistent enough for the human chromosome data to warrant additional work.

A problem surfaced, however, when 'playing around' with the IFC code with some human speech data. Learning data sets of digitized human utterances of the words 'zero', 'one', ... 'nine' were used to construct 10 networks and then to perform classification experiments. The results were entirely consistent (and even more dramatically so) with the work

presented above (i.e., lowering the channel constraint bettered classification percentages) with one important exception. The 'zero' network steadily improved correct classification percentages for channel constraint values of 5, 4, 3 and 2. However, when the constraint was set equal to 1, the correct classification percentage dropped to less than 20%. Some investigative work was done to try to find the problem but, thus far, no clear answers have become evident. There are some indicators that the channel constraint concept should not work as well with relatively short strings (like the speech data strings), but the fact that the procedure did work consistently for the other nine speech network experiments does not readily support those indicators. At this point, the issue must be put aside for further study and at least a cautionary statement issued about the problem.

CHAPTER 7

SUMMARY AND CONCLUSION

7.1. Summary.

A method has been presented for the automatic inference of structural models from sets of sample strings. It has been shown how Markov networks serve as structural models and how dynamic programming has been used as an optimal string-to-network alignment technique. Variations on the basic dynamic programming technique have been shown which can potentially lessen the computational cost of the algorithm. It has also been shown that the primary application of the inferred structural models, classification / recognition of strings, has been improved, at least for work with the human chromosome data, and quite possibly for other applications as well.

Unfortunately, the project has left as many questions unanswered as it has answered others. Thus, this paper will conclude with a brief discussion of some of these questions and some possible new directions for this research.

7.2 New Experiments with Existing Implementation

7.2.1. Use of IFC Code Using Longer Strings

The human chromosome data consisted of strings which ranged in length from 20 to 100 symbols. It was shown in Chapter 6 that use of the

channel constraint lessened the computational cost of the technique for the older, more expensive implementation but actually cost more time with the new code. It would be interesting to run the new code with much longer strings (for example, strings with 500 to 1000 symbols). There might be a break-even point where the channel constraint yields an improvement in the speed for applications using long strings as larger portions of the matrix would be eliminated from the computation.

7.2.2. Use of Channel Constraint in Inference of Markov Networks.

All of the networks used in this research were inferred without use of the channel constraint parameter -- thus, the entire cost matrix was used in building the networks. The code is already constructed to handle inference using any value of the channel constraint parameter. Some interesting results might be obtained from inferring networks from a given sample set of strings using different values of the channel constraint. These networks could then be used in classification experiments to see if those networks inferred with smaller channel constraints yielded better or worse classification results.

7.3. Increased Efficiency of IFC Implementation.

By eliminating the several levels of addressing indirection in the current implementation, the speed of the code might be improved dramatically. A conservative 'guestimate' is made that the speedup might be as high as 200%. If the dynamic allocation of the very large data structures is not important to the research or to production work, then it

might be worthwhile to statically allocate the network and cost matrix and thus eliminate the indirection.

There are probably plenty of other opportunities for tightening and making more efficient the current code. These changes should probably not be made, however, without regard to standard structured programming techniques -- one advantage of the current implementation is that its structured code should be amenable to modification.

7.4. Study of Better Classification Results When Using Smaller Values of the Channel Constraint.

With the one exception noted at the end of Chapter 6, decreasing the size of the channel constraint parameter yields an increase in the percentage of correct classifications. This is a potentially important area of study -- if it can be discovered exactly what makes the smaller constraint values yield better results then perhaps the procedure could be optimized for the purpose of bettering classification results.

7.5. Continuation of Work on Parallel Implementation.

Possibly the most interesting work to date with the inference implementation has been re-writing sections of the code to take advantage of parallel architectures. Although not 'embarrassingly parallel', the algorithm does appear amenable to 'parallelizing'. From initial experiments run on a UNIX-based simulator (simulating a shared-memory architecture) with the same strings used in the serial implementation, the technique appears to produce speedups which level off at 65% of linear speedup when using 15 - 20 processors. The code which has been run on

the simulator is a 'first try', however, and can definitely benefit from additional optimization.

Experiments with distributed parallel architectures (meshes, hypercubes, etc.) should also be conducted. It may be that the use of large numbers of processors in these architectures (which currently run as high as 65,000) might open new doors for this research.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1]Lundsteen, C., Philip, J., and Granum, E., "Quantitative analysis of 6985 digitized trypsin G-banded human metaphase chromosomes", *Clinical Genetics*, vol. 18, pp. 355-370, 1980.

[2]Levinson, S.E., Rabiner, L.R., and Sondhi, M.M., "An Introduction to the application of the theory of probabilistic functions of a Markov process in automatic speech recognition", *Bell Sys. Tech. Jour.*, vol. 62, pp. 1035-1074, 1983.

[3]Thomason, Michael G., Granum, Erik, "Dynamic Programming Inference of Markov Networks from Finite Sets of Sample Strings", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, No. 4, p. 491, July 1986.

[4]Bradley, Stephen P., Hax, Arnoldo C., Magnanti, Thomas L., Applied Mathematical Programming. (Reading, Massachusetts: Addison-Wesley Publishing Company, 1977), p. 453.

[5]Bradley, Stephen P., Hax, Arnoldo C., Magnanti, Thomas L., Applied Mathematical Programming. (Reading, Massachusetts: Addison-Wesley Publishing Company, 1977), pp. 453 - 490..

Appendix

IFC User's Guide

1. Introduction

The following describes how to use the IFC application — both the VMS and UNIX versions.

Note that the commands listed for the UNIX version are case-sensitive. Those for the VMS version are not case-sensitive. To use the IFC application in VMS a special command must be set up within the user's LOGIN.COM file (or in another .COM file which must then be executed). This command is described later.

There are two basic uses of the IFC application, *inference* and *classification*. The inference procedure constructs a new network from a set of strings. The classification procedure produces probability values each of which represents the alignment of a string with a network. Invocation of these two procedures is described below.

2. Inference

To run the inference code, only one parameter is required. An 'I' must be specified to inform the application of the inference request. Although an input set of strings is normally specified, if it is not given then the standard input file will be assumed. The following show sample invocations under VMS and UNIX.

VMS	UNIX
$ ifc i	% ifc i
(sample strings from the standard input)	
$ ifc i/samples=*filename*	% ifc i,samples=*filename*
(sample strings from specified file)	

In the previous command examples, the network file would have produced but would not have been written to a disk file. To save the network to disk, use the following parameter:

VMS	UNIX
$ ifc i/samples=*filename*/ **output=*networkfilename***	% ifc i,samples=*filename*, **output=*networkfilename***

The obvious difference between the VMS and UNIX versions is the parameter delimiter character. The VMS version uses the '/' (slash) character while the UNIX version uses the ',' (comma) character. There are no other visible differences between the two versions.

By default, IFC produces for each string processed two items of information: timing information consisting of three numbers (number of nodes in network, number of cells processed in dynamic programming matrix, and amount of CPU time spent in compute_matrix routines) and the probability of alignment value. Both items are sent to the standard output device by default. To redirect the timing information to a file, use the following additional parameter:

ifc i/samples=*filename*/output=*networkfilename* **/keeptime=*timefile***

To suppress the timing information use the following:

ifc i/samples=*filename*/output=*networkfilename* **/nokeeptime**

Similarly, the probability information is redirected or suppressed as follows:

ifc i/samples=*filename*/output=*networkfilename* **/result=*resultfile***

ifc i/samples=*filename*/output=*networkfilename* **/noresult**

Other options which can be specified at invocation include:

Analysis

Network analysis information can be requested by specifying the following:

ifc i/. . . /**analysis=*analysisfile***

If the '**=analysisfile**' is omitted, the analysis information will be sent to the standard output device. By default, the analysis information is not produced -- it must be requested explicitly.

Channel Constraint

By default, the inference routine does not use the channel constraint -- it assumes a channel constraint parameter of 0. To change this and thus utilize the channel constraint code, specify the following:

ifc i/. . . /**channel=*N***

where N = a positive integer (≥ 0).

3. Classification

To run the classification code, two parameters are required. A 'C must be specified to inform the application of the classification request. A filename for the input network file must be also specified.

Although an input set of strings is normally specified, if it is not given then the standard input file will be assumed. The following shows a sample invocation under VMS.

ifc c/**input**=*inputnetworkfile*/samples=*samplefile*

All other parameters with the exception of the **output** parameter work exactly as they do with the inference code. Since the classification code does **not** modify the network structure, it would not be consistent to allow the network file to be saved to an output file.

4. Comparing Classification Results

Once a set of classification experiments has been run and a series of result files (with probability values) has been produced, classification comparison results are often desired. A special (and separate) application has been written which gives the user comparison results from up to 15 files. This procedure is invoked as follows:

compare outputfile input1 input2 . . . inputN

where N is a positive integer ≤ 15.

The comparison results obtained from processing input1...inputN are written to outputfile.

5. Special Note for VMS Users

So that the IFC application can be invoked from the monitor prompt, it is necessary to have a version of each of the two following command set-ups in an executable .COM file (such as the LOGIN.COM file).

$ IFC :== $ DISKDEVICENAME:[DIRECTORY]IFC.EXE

and

$ COMPARE :== $ DISKDEVICENAME:[DIRECTORY]COMPARE.EXE

where the values of DISKDEVICENAME and DIRECTORY depend upon the local account where the IFC code is installed.

VITA

Gregory Scott Cole was born in Lakeworth, Florida on April 25, 1961. He moved to Knoxville, Tennessee during that same year. He attended elementary, junior high and high schools in Knoxville and was graduated from Central High School in June, 1979. The following September he entered the University of Tennessee, Knoxville, and in June, 1985 received with high honors the Bachelor of Arts degree in Computer Science. In the fall of 1985, he accepted a teaching assistantship at the University of Tennessee, Knoxville and began study towards a Master's degree in Computer Science. This degree was awarded with highest honors in June, 1988.

The author is a member of the Phi Beta Kappa and Golden Key honor societies. He has been employed by the University of Tennessee for the past five years and plans to continue employment there upon graduation.