INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Strings Algorithms and Machine Learning Applications for Computational Biology

by

Paul Brice Horton II

B.S. (University of Washington) 1989
M.S. (Kyōto University) 1992

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Dan Gusfield, Co-Chair
Professor John Canny, Co-Chair
Professor Gerry Rubin
Professor Christos Papadimitriou

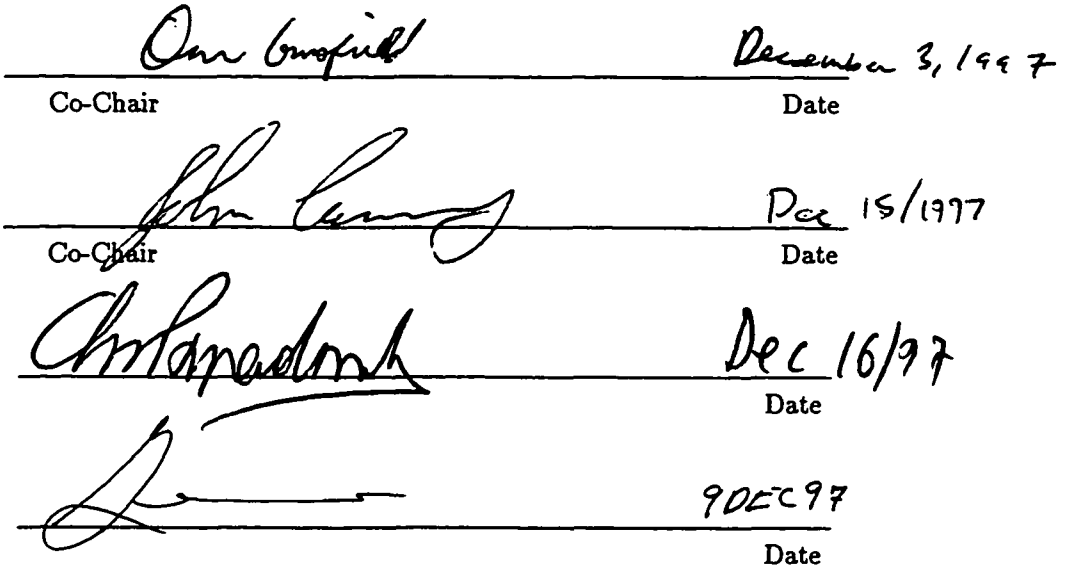
1997

Strings Algorithms and Machine Learning Applications for Computational Biology

by

Paul Brice Horton II

The dissertation of Paul Brice Horton II is approved:



Co-Chair — Date December 3, 1997

Co-Chair — Date Dec 15/1997

Date Dec 16/97

Date 9 DEC 97

University of California at Berkeley

1997

To the memory of Eugene Lawler,

without whom this work would not have begun.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Dan Gusfield for working with me, a student from another school, over a period of several years. Without him this disseratation would not have been possible. I would also like to thank my colleagues Kevin Murphy, Geoff Zweig, and Daniel Wilkerson; both for numerous helpful discussions on technical matters, and also for their constant friendship and good humor. Finally, I would like to thank my sister Jenny, who cheered me on throughout in the unconditional way that only sisters can.

Chapter 1

Introduction

1.1 Introduction

1.1.1 Motivation

Today computer scientists have a unique opportunity to contribute to the quest to understand life, and ultimately ourselves. All fields which involve large amounts of data rely on computers for processing that data, however molecular cell biology has a special relationship to computer science due to the role of the cell as an information processing machine. Computer scientists can apply many of their concepts to biology. For example, the property of self-reference required for organisms to reproduce themselves is also central to the computer science concepts of recursion and computability.

Indeed the analogy between the information contained in living organisms and computer software is compelling. In this analogy nucleic acid is the storage device for biological software. The raw DNA sequence of an organism is analogous to a hexadecimal listing of the binary executable of a suite of programs. After identifying the role of DNA in inheritance, the first breakthrough in understanding that software was the "breaking" of the genetic code in the early 60's [4]. In terms of our analogy the genetic code provides us with a partial disassembler. Partial because it only disassembles the "opcodes" which correspond to sequences of proteins or RNA molecules and not the promoters and transcriptional regulatory elements which are analogous to the if statements and while loops of computer programs. Although much has been learned about such control information, the task of understanding our DNA program is still daunting. Software engineers know that it is very difficult to understand uncommented, assembly code. Even worse, in the case of the programs written in our DNA, not only is the code undocumented but it is not even the product of a rational design.

Despite the magnitude of the problem, rapid progress is being made in the first step of the process, namely obtaining the program listing itself. The genomes (i.e. entire DNA sequence) of *E. coli*, the yeast *Saccaromyces cerevisiae*, and the nematode *C. elegans* have been completely determined. Furthermore rapid progress is being made towards sequencing the larger genomes of the fruit fly *Drosophila melanogaster*, humans and mice. Indeed figure 1.1 shows that the sequence repository GenBank is growing dramatically. The curve is approximately exponential with a doubling time of 22 months.

This thesis identifies two particular areas where advances will help us understand this program listing as it becomes available. The first is the development of more efficient

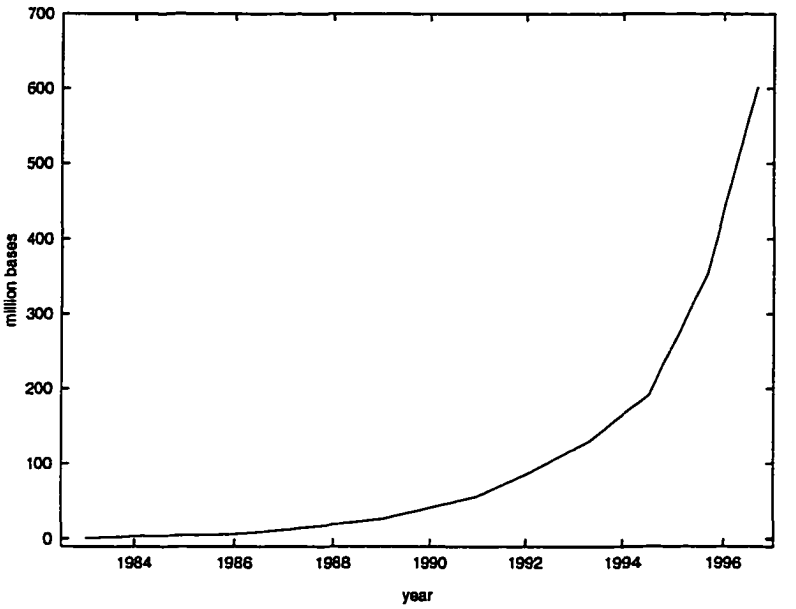


Figure 1.1: The amount of DNA sequence data in GenBank is shown. Reproduced with permission from [5].

algorithms for comparing strings and the second is the development of an effective methodology for learning to classify the function of sequences from labeled training examples.

Perhaps the most powerful technique for understanding sequence function at this point in time is the comparison of sequences to other sequences of known function. This is necessary because we do not understand the mapping of sequence to function well enough to understand a sequence in isolation. It is effective because the process of evolution preserves useful sequence patterns. The first three chapters of this thesis contribute to the understanding of algorithms that compare two or more sequences.

Our inability to directly compute function from sequence also motivates another emerging area of research, the application of machine learning to learn the mapping from sequence to function from statistical information rather than from first principles. In chapter 4 we describe a successful attempt to learn such a mapping from sequence data.

The comments and documentation that will allow future generations to understand DNA programs are slowly being filled in by the painstaking work of molecular biologists. However, the vast majority of the documentation remains missing. We believe that some of the important remaining comments are destined to be filled by researches trained as computer scientists. It is our hope this work will contribute in a small way to that great endeavor.

1.1.2 Overview

In this section we describe the organization of this thesis and briefly summarize each chapter, more detailed introductions can be found at the beginning of each individual chapter. This thesis is arranged in two parts. The first part is on string comparison algorithms and consists of the first three chapters. The second part is the final chapter on the application of machine learning to biology.

Local Multiple Sequence Alignment

This chapter describes results published in the Pacific Symposium on Biocomputing '96 [1]. The main result is a branch and bound algorithm which increases the number of sequences for which a popular formulation of local multiple sequence alignment can be solved from about four to around ten. A minor result is an alternative to the standard enumerative algorithm which saves a time factor of roughly $w/2$, where w is the width of the desired local multiple alignment.

1.1.3 Calculating Edit Distance with the four Russian's Technique

The main result of this chapter is a modification to the four Russians technique which allows the edit distance between two strings of length n to be computed in less that $O(n^2)$ time, namely $O(n^2 \log\log n / \log n)$ time, for alphabets as large as $O(n)$. For small, constant size alphabets the technique does not improve on the asymptotic running time of the standard four Russians algorithm but is does save a factor of up to $\sigma!$ space, where σ is the size of the alphabet. The results of simulation for evaluating the tradeoffs between different algorithms for different submatrix sizes and input sizes is also reported.

1.1.4 Efficiency of the A^* Algorithm for Multiple String Alignment

The main result of this chapter is a proof that the A^* algorithm dominates the much better know precomputational algorithm described by Carillo & Lipman. A minor result is to point out the relationship of A^* to the branch and bound algorithm of Kececioglu which is implemented in the MSA program. It is shown that although Kececioglu's algorithm is very similar to A^*, it needs to be provided with a good feasible solution to be effective while A^* needs none.

1.1.5 Learning to Classify Protein Sequences by their Cellular Localization Sites

The chapter describes results published at the 1996 and 1997 Intelligent Systems in Molecular Biology conferences [2], [3]. The chapter describes a software system which predicts the intracellular localization site of proteins from their amino acid sequence. The problem is posed as a classification problem, mapping expert identified features which can be computed from the input sequence to localization sites. The software currently uses the standard k Nearest Neighbors algorithm to do the classification. A new classification method which can be formulated as a class of Bayesian networks is also introduced. That method and the standard methods of k Nearest Neighbors, decision tree, and Naïve Bayes are compared for the localization problem. The k Nearest Neighbors algorithm showed the best classification accuracy but our new method has the advantage of being much easier for the human expert to interpret. We also show anecdotal success in using the suffix tree data structure to identify interesting sequence features.

Bibliography

[1] Paul Horton. A branch and bound algorithm for local multiple alignment. In *Pacific Symposium on Biocomputing '96*, pages 368–383, 1996.

[2] Paul Horton and Kenta Nakai. A probabilistic classification system for predicting the cellular localization sites of proteins. In *Proceeding of the Fourth International Conference on Intelligent Systems for Molecular Biology*, pages 109–115, Menlo Park, 1996. AAAI Press.

[3] Paul Horton and Kenta Nakai. Better prediction of protein cellular localization sites with the k nearest neighbors classifier. In *Proceeding of the Fifth International Conference on Intelligent Systems for Molecular Biology*, pages 147–152, Menlo Park, 1997. AAAI Press.

[4] M. Nirenberg and P. Leder. Rna codewords and protein synthesis. *Science*, 145:1399–1407, 1964.

[5] Jens Stoye. *Divide-and-Conquer Multiple Sequence Alignment.* PhD thesis, Universität Bielefeld, Postfach 10 01 31, 33501 Bielefeld, 1997.

Part I

String Algorithms

Chapter 2

Local Multiple String Alignment

Abstract

A branch and bound algorithm has been developed to find a set of window positions in a compilation of sequences with globally maximal information content. We have also developed an algorithm for brute force evaluation of solutions which is faster by a factor of the length of the windows than the naïve brute force algorithm. The combination of these two algorithms allows us to solve problems to optimality that were previously amenable only to heuristic algorithms.

2.1 Introduction

In recent years the amount of DNA and protein sequence available to computer analysis has increased dramatically. The availability of such sequence data has inspired formulations of local multiple sequence alignment problems, which are designed to have solutions that give the positions of local patterns within a collection of sequences. In this paper we introduce two algorithms for solving Stormo and Hartzell's formulation of local multiple sequence alignment [12], whose objective function relates to the entropy of the proposed patterns. This paper is divided into sections with the following purposes:

- Describe the local multiple sequence alignment formulation in more detail.
- Introduce an exhaustive algorithm.
- Introduce a modification of an earlier heuristic search algorithm.
- Introduce a branch and bound algorithm.
- Prove the validity of the bound used.
- Describe precomputation and an additional heuristic speedup.
- Describe the data sets used in this study.
- Summarize and discuss the results of the algorithms on the data sets.
- Suggest future directions for research and conclude.

2.2 Problem Formulation

The formulation of local multiple sequence alignment which we use was proposed by Stormo and Hartzell [12]. The general assumption is that each sequence contains one substring that provides a biological function. The problem is difficult because the constraints on the characters in the substring may be somewhat loose. Informally the goal is to select a substring or *window* for each string such that, for most positions i in the window, the distribution of the ith character in each window is far away from the expected distribution based on a prior distribution of characters. More formally, the input is a set of n strings $S_1, S_2, \ldots, S_n$ over a fixed alphabet A, and an integer window length w such that w is between 1 and the length of the shortest string. Let V specify one position in each string, i.e. V is a vector of length n whose elements v_i are integers satisfying $\forall i\ 1 \leq v_i \leq l_i$, where l_i is the length of string i. The frequency of a character b in V is denoted by the matrix entry $F(V, b)$, where $\forall V\ \sum_{b \in A} F(V, b) = 1$, and the vector of frequencies for all the characters at a given set of positions V is denoted $F(V)$. The prior probabilities of the characters is given by a vector P, with element p_b of P representing the prior probability of character b. We can now define the information content (also known as the relative entropy) introduced by Schneider *et al* [11] which provides a measure of the distance between the distributions $F(V)$ and P. The function is given by

$$I(V) = \sum_{b \in A} F(V, b) * log_2(F(V, b)/p_b).$$

Schneider *et al* denote this function $R^*_{sequence}$ and use it to extend the standard information theory entropy,

$$H(\, F(V)\,) = -\sum_{b \in A} F(V, b) * log_2 F(V, b),$$

to include prior probabilities. Bailey [2] shows that $I(V)$ can be interpreted as the ratio of the likelihood that a random character generator that generates characters according to the distribution $F(V)$ would generate the n characters observed at position V, to the likelihood that a generator using the distribution P would generate those n characters. The appropriateness of the scoring function defined here further is considered in the discussion.

We wish to maximize the information content over a window of length w. Thus our evaluation function becomes

$$E(V) = \sum_{i=0}^{w-1} I(V+i),$$

where we abuse notation to denote $(v_1+i, v_2+i, \ldots, v_n+i)$ as $V+i$ for a scalar i, and define $E(V)$ only for V such that $\forall i\ 1 \leq v_i \leq l_i - w + 1$ to prevent the window from extending past the end of the sequences. The alignment problem then is to find a V such that $E(V)$ is maximal.

2.3 An Exhaustive Algorithm

Let l be the length of the longest string. A naïve algorithm would calculate the frequencies for the characters in the n strings for each of the w columns covered by each possible window position. There are $O(l^n)$ possible window positions so this would require $O(w(l^n))$ calculations of F and $O(nw(l^n))$ time. However for fixed n, it is possible to remove the factor of w from the running time. The main observation is this: Let V be a set of window positions such that $\forall i\ v_i \leq l_i - w$, and $V' = V + 1$. In other words let V' be V shifted one base to the right. Then $E(V')$ may be obtained as follows:

$$E(V') = E(V) - I(V) + I(V+w).$$

Using this observation to good advantage the algorithm is straightforward:

$Max\_E \longleftarrow -\infty$

Iterate over all starting vectors V such that at least one element is equal to one and for all elements v_i of V, $1 \leq v_i \leq l_i - w + 1$.

$E\_V \longleftarrow \sum_{i=0}^{w-1} I(V+i)$

Do

$Max\_E \longleftarrow \max\{Max\_E, E\_V\}$

$E\_V \longleftarrow E\_V - I(V) + I(V+w)$

$V \longleftarrow V + 1$

Loop while no element v_i of V exceeds l_i

Return Max_E

There are $l^n - (l-1)^n = O(nl^{n-1})$ starting vectors when at least one element is constrained to equal one. Each starting vector requires $O(wn)$ time for initialization. For all other vectors, $E(V)$ can be calculated by adding the information content of one column and subtracting the information of another (actually the value of $I(V+w)$ can be cached to avoid the need to calculate $I(V)$ w iterations later, thus reducing the information content calculation to one column per iteration), which requires $O(n)$ time per vector. Thus the overall running time is $O(n^2wl^{n-1}+nl^n)$, which is $O(nl^n)$ as long as nw is not much greater than l. In practice for DNA sequences with $n \leq 7$ we precompute the value of $I(V)$ for all possible vectors V thus reducing the time to $O(nwl^{n-1}+l^n)$; if array lookups are counted as constant time operations.

2.4 Heuristic Search

The optimization problem we are concerned with can be formulated as a search problem over a search tree whose leaves represent each possible choice of window positions. More formally, the search tree can be constructed as follows: start with a root at level 0, then give the root $l_1 - w + 1$ child nodes representing each possible position for a window in the first sequence. Likewise for i from 1 to $n-1$ give each node at level i $l_{i+1} - w + 1$ child nodes to represent the possible choices of the window position in S_{i+1}. There is a one to one correspondence between paths from the root to leaves in the tree and the possible assignments of the window positions for the sequences.

Stormo and Hartzell [12] used a form of greedy search on the search tree as a heuristic for quickly finding good, but not necessarily optimal, solutions. Their algorithm actually performs $l_1 - w + 1$ greedy searchs in parallel, one starting from each node at level 1 of the tree, and returns the best solution found. The greedy search can be described iteratively from $i = 1$ to $n-1$. When the level of the current node is i, then for each child of the current node, calculate the function $E(v_1, v_2, \ldots, v_{i+1})$ (simply denoted E for the rest of this section) for $S_1, S_2, \ldots, S_{i+1}$, where $(v_1, v_2, \ldots, v_{i+1})$ is the set of window positions implied by the path from the root to the child node. If there is one child whose path maximizes E then make that child the current node and iterate. Otherwise in the case of a tie between

the children at level $i+1$ for maximizing E then "split" the search into independent greedy searches for each child starting with that child as its current node.

We applied a similar heuristic that can be described as a kind of beam search. This type of beam search was suggested for multiple alignment problems by Bacon and Anderson [1]. The algorithm descends the tree in a breadth first search fashion except that at each level of the tree it only keeps the nodes corresponding to the top k values of E for that level. The parameter k is user specified. When k is equal to one this type of search is equivalent to a simple greedy search.

2.5 Branch and Bound

We developed a branch and bound algorithm which uses the same search tree as the beam search algorithm but finds a guaranteed optimal solution. The algorithm is essentially depth first search with pruning. An inequality is used to compute an upper bound on $E(V)$ for any V whose first i elements are determined by the path to the node at level i of the search tree. In order to describe the bound formally we need to introduce some more notation. Let

$$E(v_1, v_2, \ldots, v_i, v_{i+1} \rightsquigarrow v_n) = \max_{v_{i+1}, v_{i+2}, \ldots, v_n} E(v_1, v_2, \ldots, v_n),$$

$$E(v_{i+1} \rightsquigarrow v_n) = \max_{v_{i+1}, v_{i+2}, \ldots, v_n} E(v_{i+1}, v_{i+2}, \ldots, v_n).$$

In words, $E(v_1, v_2, \ldots, v_i, v_{i+1}, \rightsquigarrow v_n)$ is the optimal value of $E(V)$ for a set of window positions V which starts with a path to a particular node of the search tree at level i; while $E(v_{i+1} \rightsquigarrow v_n)$ is the optimal value over V' of the evaluation function $E(V')$ where V' is a set of window positions in sequences $S_{i+1}, S_{i+2}, \ldots, S_n$. The upper bound we use can now be written as:

$$\begin{aligned} n * E(v_1, v_2, \ldots, v_i, v_{i+1} \rightsquigarrow v_n) \leq \\ i * E(v_1, v_2, \ldots, v_i) + \\ (n - i) * E(v_{i+1} \rightsquigarrow v_n). \end{aligned} \tag{2.1}$$

Note that the second term of the right hand side can be obtained by solving a smaller version of the original problem. Utilizing that fact we have developed a recursive algorithm that uses two arrays (which are defined to be global variables) to hold bounds:

> Array L is defined such that $L[i]$ will be assigned a lower bound for the maximum value of E for the sequences $S_i, S_{i+1}, \ldots, S_n$,
>
> Array U is initiated to have undefined values for all its elements and will be assigned values during the execution of the algorithm such that $U[i]$ is either undefined or it contains the exact maximal value of E for the sequences $S_i, S_{i+1}, \ldots, S_n$.

The level at which the algorithm begins pruning is denoted d, which is a user defined parameter. A description of the algorithm is given in table 2.1. Pruning is generally ineffective at the first 2 levels of the tree, therefore we used $d = 3$ when the program was called with less than 8 sequences, i.e. when $n - start\_seq + 1 \leq 7$, and $d = 4$ otherwise. Note that the array L can be computed in one pass of the beam search heuristic by inputing the sequences in reverse order and letting $L[i]$ be the best candidate for the sequences $S_n, S_{n-1}, \ldots, S_i$.

2.6 Proof of the Bound

We first prove the inequality for $I(V)$ and then for $E(V)$. Let F, F_0, and F_1 be probability vectors with elements $F(b)$ specifying the probability of generating character b. Furthermore let the probability vectors satisfy the relation:

$$\forall b \in A \; F(b) = \theta F_0(b) + (1 - \theta) F_1(b), \; 0 \leq \theta \leq 1.$$

Define I analogously with $I(V)$ defined earlier, i.e.

$$I = \sum_{b \in A} F(b) * log_2(F(b)/p_b).$$

Where the p_b's are constants. Define I_0 and I_1 likewise using F_0 and F_1.

Theorem 1 $I \leq \theta I_0 + (1 - \theta) I_1$.

Description of Branch and Bound Algorithm

Use the beam search heuristic to calculate the values of an array L such that $L[i]$ holds a lower bound for the maximum value of E for the sequences $S_i, S_{i+1}, \ldots, S_n$.

Call Program(1).

Program(*start_seq*)

if *start_seq* $\geq n - d$ then
    Calculate $U[$*start_seq*$]$ with the exhaustive algorithm
    Return $U[$*start_seq*$]$

for($i = n$ to *start_seq* $+ d$ step -1)
    if $U[i]$ is undefined $U[i] \longleftarrow$ Program(i)
endfor

$Max\_E \longleftarrow -\infty$

Visit all the nodes at level d of the search tree using the exhaustive algorithm described above. For each node, use $U[$*start_seq* $+ d]$, $L[$*start_seq*$]$, the value of $E(v_1, v_2, \ldots, v_d)$ for the path $v_1, v_2, \ldots, v_d$ leading to the node, and inequality 2.1 to determine if the node can be eliminated.
    For each node generated that cannot be eliminated by inequality 2.1
        perform a depth first search,
            pruning any node generated that can be eliminated with the inequality.
        For each node generated at level n update Max_E if necessary.

Return Max_E

Table 2.1: The main function "Program" is recursively called with the argument *start_seq* specifying from which sequence "Program" is to begin. For example, if "Program" were called with *start_seq* $= 5$, it would return an optimal solution for input consisting of the 5th through nth sequences.

The meat of the proof is provided by the following lemma:

Lemma 1

$$H \geq \theta H_0 + (1-\theta)H_1 \tag{2.2}$$

where:

$$\begin{aligned} H &= -\sum_{b \in A} F(b) * log_2 F(b), \\ H_0 &= -\sum_{b \in A} F_0(b) * log_2 F_0(b), \\ H_1 &= -\sum_{b \in A} F_1(b) * log_2 F_1(b). \end{aligned}$$

Proof of lemma 1:

The proof closely follows the proof of a similar theorem by Gallager [5]. Consider F, F_0, and F_1 to be probability vectors over the sample space B equal to the space of generating single characters from the alphabet A. Furthermore consider the probabilities $F_0(b)$ to be conditioned on a binary variable z with sample space $Z = \{0, 1\}$ such that

$$F_0(b) = F_{B|Z}(b|0), F_1(b) = F_{B|Z}(b|1).$$

Let $z = 0$ with probability θ. Then inequality 2.2 is equivalent to the following series of equations:

$$\begin{aligned} H(B) \geq & \sum_b Prob[z=0] * F_0(b) * log_2 F_0(b) + \\ & \sum_b Prob[z=1] * F_1(b) * log_2 F_1(b) \end{aligned}$$

$$H(B) \geq \textstyle\sum_{b,z} Prob[b,z] * log_2(Prob[b|z])$$

$$H(B) \geq H(B|Z).$$

The steps follow from the identity $Prob[b,z] = Prob[z]Prob[b|z]$ and the standard information theory definitions of $H(B)$ and $H(B|Z)$. The last inequality is a well know fact from information theory. This completes the proof of lemma 1.

The proof of theorem 1 follows from lemma 1 by noting that I can be written as: $I = -H - F \cdot C$, where C is a vector defined such that $C(i) = log_2(p_i)$. C is a constant vector with respect to F so we have:

$$F \cdot C = \theta F_0 \cdot C + (1-\theta) F_1 \cdot C,$$

$$I - \theta I_0 - (1-\theta) I_1 = \theta H_0 + (1-\theta) H_1 - H$$

By lemma 1 the right hand side of the last equation is less than zero, thus finishing the proof of theorem 1.

2.7 Precomputation and Heuristic Speedup

2.7.1 Precomputation

The function $I(V)$ requires the calculation of several logarithms. Without precomputation those calculations would become a bottleneck, thus we employed two kinds of precomputation. First, as mentioned in the section on the exhaustive algorithm, for each possible column of up to 7 characters we precompute the value of $I(V)$ for that column. For nucleic acid sequences this requires $4^7 + 4^6 + \cdots + 4 \approx 22K$ double precision numbers worth of memory. This kind of precomputation is less effective for the larger alphabet size required for amino acid sequences but precomputing the values for columns of up to 3 or 4 amino acids is possible.

For columns with too many characters to cover with the above form of precomputation it is still possible to avoid calculating logarithms during the main phase of the algorithm. The key observation is that the character frequencies for which logarithms need to be computed always equal the ratio of the number of times a character is observed, which must be a nonnegative integer no greater than the number of sequences in the subproblem, divided by the number of sequences in the subproblem, which can be no greater than n. Thus there are only $O(n^2)$ possible frequencies. To exploit this fact, we precompute a $n+1 \times n+1$ array M, where for $i \leq j$ the $M[i][j]$ entry holds the value of $\frac{i}{j} * log_2(\frac{i}{j})$. Although this array does not allow $I(V)$ to be calculated with a single array lookup, in still saves time in practice by avoiding repeatedly computing costly logarithms.

2.7.2 Heuristic Input Ordering Speedup

Recall that when descending the search tree, the condition for eliminating a node and its descendants is dependent on the value of E for the path to the node and on the value of E for the optimal set of window positions maximized independently over the remaining sequences. Thus more nodes can be eliminated early if the sequences with the lowest maximal value of E come last in the input. To give a concrete example consider the case where there are 11 input sequences and one is trying to eliminate nodes at the fourth level of the tree. If the maximal value of E for the last 7 sequences of the input is low then the right side of inequality 2.1 will provide a lower upper bound on the value of E for descendents of the node in question.

We exploit this observation by ordering the sequences based on the results of running the heuristic beam search on the initial, arbitrary ordering of the input. The input sequences are sorted in descending order according to the value of $\sum_{i=0}^{w-1} log_2(F(i, B(i,j))/p_{B(i,j)})$, where $F(i, B(i,j))$ denotes the frequency in column i, in the window position vector chosen by the heuristic, of the character in the ith column of the window in the jth sequence. And $p_{B(i,j)}$ denotes the prior probability of that character. The logic behind this is that if you believe that there is one strong signal in the data and that the heuristic can find that signal, then the sequences where that signal is weakest should have a low optimal value of E and thus should come last.

2.8 Data Sets

2.8.1 LexA Data Set

For this study we used a set of 11 E.coli DNA sequences of length 200, each of which is known to contain at least one binding site for the protein LexA. We copied this data set directly from Hertz *et al* [6]. LexA binds in or near promoters and may bind to sites which occur on either the sense strand or the anti-sense strand of the DNA. Thus we needed to consider possible occurrences of the pattern on both the sequence and the reverse complement of the sequence. Instead of designing our program to consider the reverse complement of each input sequence, we simply concatenated the reverse complement of each input sequence onto itself to create 11 sequences of length 400 each. This added $w-1$

bogus window positions in the middle of each input string, but as these window positions never appeared in a solution the solutions obtained were valid. As in Hertz *et al* we used prior probabilities of 0.25 for each of the four DNA bases.

2.8.2 Artificial Data

Intuitively one would expect stronger motifs to be easier to find. In order to investigate the relationship between motif strength and the running time of our algorithm we generated sets of artificial sequences with artificial motifs of varying strength planted in them. We somewhat arbitrarily chose to generate sets of nine sequences of length 320, planted with motifs of length 20.

The sequence generator receives four parameters: the length of the sequences to be generated l, the number of sequences to be generated n, the length of the motif w, and the minimum information content c of each column of the motif. The generator can be described with the following pseudo-code:

Generate n DNA sequences of length l using a uniform distribution over the four possible bases.

For $i = 1$ to w
    Do
        Generate a motif column i of length n using a
        uniform distribution over the four possible bases.
    Loop while the information content of motif $i <= c$.
EndFor

Generate a set of n random motif starting positions using the uniform distribution over all possible starting positions.

Replace the characters in the sequences whose positions correspond to motif columns with the characters from the motif columns generated above.

2.9 Results

2.9.1 LexA Dataset

We ran the branch and bound program on the LexA data set with three different window lengths. The patterns found by the branch and bound program with a window length of 24 are shown in table 2.2. The running time was quite slow, requiring 71 hours on a Hewlett Packard 9000/715 workstation. The program ran faster with window sizes of 20 and 22, requiring 20 and 28 hours respectively. We must mention that the sequence input ordering heuristic was used for the window size of 24. Without reordering the sequences the program took so long to execute that it had to be terminated prematurely, however judging from the pace of its progress we estimate the reordering to have sped the calculation up by at least a factor of 10. The reordering heuristic was not necessary for window sizes of 20 and 22. Tables 2.3, 2.4, and 2.5 show how many nodes survived at each level of the search tree with the different window sizes. For window sizes of 20 and 22 the number of nodes peaks at level 4 where we began pruning. For the length 24 window, at level 5 or greater, many more nodes survive than for either the 20 or the 22 length window; which we would expect from the difference in run times. The relatively high effectiveness of the bound for level 4 with the length 24 window was evidently due to the use of the reordering heuristic, since that heuristic is designed to facilitate pruning effectively towards the top of the tree. Even with the least favorable window size of 24, the bound does well in terms of the percent of nodes eliminated. Remembering that the fan out for this particular search tree is $400 - 24 + 1 = 377$ for each level of the tree we can see that the bound eliminates all but 6×10^{-4} percent of the nodes at level 4 and all but 7×10^{-5} percent of the nodes at level 5 of the search tree. It is not clear exactly why significantly fewer nodes are pruned overall for the length 24 window size. The information content per column for the length 24, 22, and 20 windows is 1.22, 1.29, and 1.36 respectively. The only clear conclusion we can draw is that the relationship between information content per column and running time is not linear. This result was also confirmed in our experiments with artificial data described in the next section.

We did not run extensive empirical tests of the performance of our beam search heuristic with the heuristic of Stormo and Hartzell [12]. However the results from very limited data suggest that our heuristic performs somewhat better. Table 2.6 shows how their results

Sequence	Pattern Found	
umu-operon	CTACTGTATATAAAAACAGTATAA	
cloacin-df13	ATACTGTGTATATATACAGTATTT	
recn	TTACTGTATATAAAACCAGTTTAT	
uvrb	ATACTGGATAAAAAAACAGTTCAT	(complementary strand)
reca	ATACTGTATGAGCATACAGTATAA	
sula	TTACTGTATGGATGTACAGTACAT	(complementary strand)
colicin-ib	TATATGGATACATATACAGTACTA	(complementary strand)
colicin-ia	CATATGGATACATATACAGTATTA	(complementary strand)
colicin-e1	ATGCTGTATATAAAACCAGTGGTT	
uvrd	AATCTGTATATATACCCAGCTTTT	
uvra	ATACTGTATATTCATTCAGGTCAA	

Table 2.2: The name of each sequence and the 24 base window found by the branch and bound algorithm are given. When applicable the occurrence of the window on the complementary strand is indicated.

Level of Tree	Number of Nodes that Survive
4	131232
5	499498
6	3489962
7	597786
8	133618
9	200389
10	27146

Table 2.3: Results for the branch and bound program on the LexA dataset with a window width of 24. The number of nodes surviving at each level of the search tree for which pruning is done is shown.

Level of Tree	Number of Nodes that Survive
4	1751830
5	272865
6	44539
7	18625
8	22903
9	11818
10	10420

Table 2.4: Results for the branch and bound program on the LexA dataset with a window width of 22. The number of nodes surviving at each level of the search tree for which pruning is done is shown.

Level of Tree	Number of Nodes that Survive
4	100652
5	32120
6	7372
7	6468
8	7491
9	7789
10	5508

Table 2.5: Results for the branch and bound program on the LexA dataset with a window width of 24. The number of nodes surviving at each level of the search tree for which pruning is done is shown.

Search Type	Window Size	E(V) of Solution Found	Times found / # of trials
Beam Best		27.241	5/5
Stormo Best	20	27.241	1/5
Beam Best		28.426	5/5
Stormo Best	22	28.426	2/5
Beam Best		29.279	2/5
Beam Other		29.181	3/5
Stormo Best	24	28.926	2/5

Table 2.6: Beam Best labels the best solution found by our beam search heuristic while Beam Other labels a worse solution which was also found. A trial consists of randomly rearranging the order of the sequences and running the programs. The numbers for Stormo and Hartzell's heuristic were taken from their paper, while we randomly generated our own five orderings for our numbers.

compare with the beam search heuristic. We chose to keep the number of candidate paths, i.e. the beam width, equal to the length of the sequence, a parameter setting which should keep the amount of memory and time resources used by the two heuristics roughly equal (recall that Stormo and Hartzell's algorithm performs one greedy search for each window in the first sequence). Note that that although the difference is small our beam search always finds an equal or better solution. If one increases the number of candidates kept to 1700 from 400, then our beam search heuristic finds the optimal solution for a window size of 24 for all five orderings. The running time with the 1700 setting is just under 20 minutes.

2.9.2 Artificial Dataset

The running times with different window sizes on the LexA data set provides anecdotal evidence that a higher information content per position correlates with shorter running times. The results of a more systematic investigation of this correlation are shown in figure 2.1. Again the input ordering heuristic was necessary for the harder input sets, in this case the ones with motif strengths of 21.4 and 21.8 bits. It can be seen that the computation becomes infeasible as the information content drops below 21.4 bits for a motif length of 20.

To get a feel for the significance of this motif strength we tried to determine what information content could be expected when no motif was present. When run on sets of nine random sequences of length 320 (without any planted motifs), our beam search heuristic

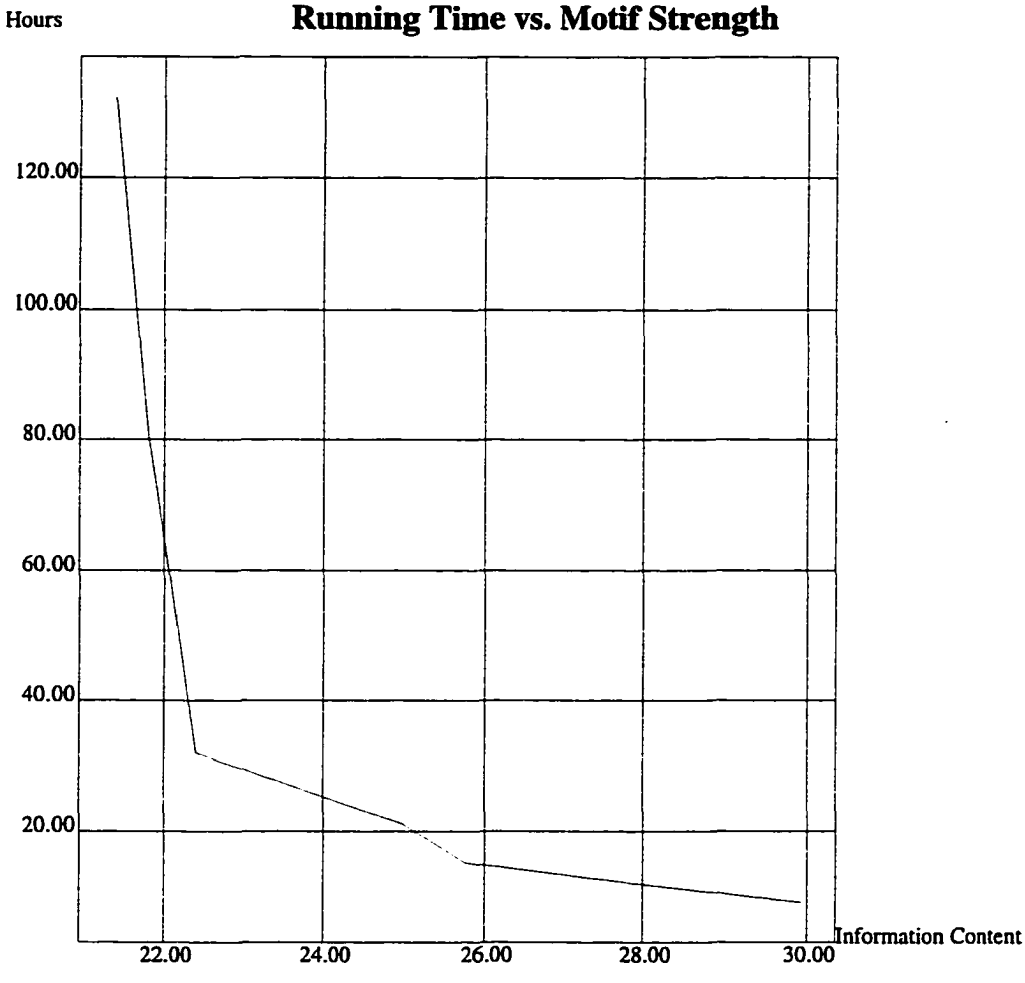


Figure 2.1: The y-axis gives the running time in hours of the branch and bound algorithm, while the x-axis gives the total information content of the optimal set of window positions. The data shown is from the artificial data set.

consistently found length 20 windows with an information content of approximately 18.8 bits. Thus for data sets of the size tested here, the possibility remains of subtle motifs appearing with strengths in the range of 18.8 to 21.4 bits over 20 columns which cannot be found with the current implementation of our branch and bound algorithm.

2.10 Discussion

2.10.1 Appropriateness of the Scoring Function

The work here only uses first order statistics to score alignments. It is easy to see that it will generally be difficult to accurately estimate second or higher order statistics from sample sizes of around 10 sequences. However it is not obvious what function of the first order statistics should be used to score an alignment. Ideally we would like to have an argument which shows from first principles that the scoring function used in this work is clearly the correct choice. Indeed some theoretical work has been done along those lines, in particular Berg & Hippel [4] derive a relationship which relates the information content to the number of protein molecules which would be expected to be bound to spurious random sites at equilibrium. However we will settle for a much more modest goal, namely to use three simple examples to show that the scoring function used here does behave as one would expect and that two other plausible scoring functions fail to behave as one would expect. To simplify the discussion we again assume that the prior probability of each base is $\frac{1}{4}$. Thus our scoring function is equivalent to the negative entropy of the observed base frequencies summed over each column of the alignment. The examples we will use are shown in figure 2.2. For our arguments we will assume that observing more sequences would not significantly change the observed frequencies. Of course this is probably not true for such a small sample size of four sequences but we chose a small number to keep the examples manageable. If this seems unjustified the reader may consider a similar set of examples in which each sequence is replaced by a large number of identical sequences. We argue that a good scoring function should score the three alignments from figure 2.2 as $\alpha = \beta > \Gamma$. Why should α score the same as β? In words α represents the statement that the protein molecule that binds to its site will bind anywhere it finds an "a", while the protein that recognized β's site will bind anywhere it finds an "a" or a "c" followed by a "g" or a "t". In both cases we would expect the protein to wastefully bind to $\frac{1}{4}$ of the positions in a random DNA sequence. Since it is the wasteful, and indeed potentially harmful, binding of the recognizing protein to spurious sites that nature wants to avoid, we believe that α and β should receive the same score. It is less obvious how one should put Γ in words, perhaps something like the protein preferably binds to an "a" in the first position but sometimes accepts a "c" or a "t" and preferably binds to "g" in the second position but sometime accepts an "a" or "t". In any case it seems obvious that the pattern describing β is strictly more specific that

the pattern describing Γ and therefore it should score higher. It can be verified that the entropy scoring function does indeed score $\alpha = \beta > \Gamma$, however some possible alternative scoring functions do not give the same relationship. One plausible scoring scheme would be the product of the ratio of the expected to the observed frequency of the most common base in each column. For four sequences the expected frequency of each base is 1 so the score for α would be $(\frac{1}{1})(\frac{4}{1}) = 4$, likewise the score for β would be $(\frac{2}{1})(\frac{2}{1}) = 4$. Thus this scoring function does indeed score $\alpha = \beta$. Unfortunately, since the scoring function only depends on the frequency of the most common base, Γ would also have a score of 4. One could imagine using a function of the ratio of observed to the expected frequency for each base instead of just the most common base to cause Γ to score lower than β, but it is hard to see how that can be done without losing the equality between the scores for α and β. Intuitively a good pattern is as much different from random chance as possible, so one might propose simply using the probability that a given set of observed base frequencies would be generated by random chance. The probability that random chance would generate an alignment giving the frequencies of α is

$$(\frac{1}{4}^4)(4!\frac{1}{4}^4) = 24/4^8,$$

since there are 4! different columns of length 4 that contain one occurence of each base. Likewise the probability of random chance generating an alignment giving the frequencies of β is

$$(\binom{4}{2}\frac{1}{4}^4)(\binom{4}{2}\frac{1}{4}^4) = 36/4^8.$$

So this scoring function gives α a better score than β. We hope that these simple examples will convince the reader that the entropy based scoring function is at least reasonable, and superior to some other plausible alternatives.

2.10.2 Summary and Future Directions

Our main result was that we were able to solve a non-contrived data set of reasonable size to global optimality. The data set we used was previously used to demonstrate the heuristic of Stormo [6]. Also the size of the problem ($O(400^{11})$) solved here is comparable in size to the problem sizes used to test two other heuristic algorithms for maximizing functions very similar to $E(V)$. Specifically the data sets used to test an expectation maximization algorithm by Lawrence and Reilly [10] and an algorithm using Gibbs Sampling by Lawrence

α	β	Γ
aa	ag	ag
ac	at	at
ag	cg	cg
at	ct	ta

$$\begin{bmatrix} 1 & \frac{1}{4} \\ 0 & \frac{1}{4} \\ 0 & \frac{1}{4} \\ 0 & \frac{1}{4} \end{bmatrix} \quad \begin{bmatrix} \frac{1}{2} & 0 \\ \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \\ 0 & \frac{1}{2} \end{bmatrix} \quad \begin{bmatrix} \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & 0 \\ 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} \end{bmatrix}$$

Figure 2.2: The columns of three hypothetical alignments and their observed base frequencies are shown.

et al [9] were not much bigger than $O(400^{11})$. We must note here that in practice these heuristics produce good solutions and are more flexible in terms of modifying the objective function, not to mention that they run much faster than our branch and bound algorithm. Our point is that it is somewhat surprising that a problem as large as the one presented here could be solved to optimality.

Branch and Bound algorithms have been effective in optimally constructing global multiple alignments, where *global* refers to aligning the whole strings rather than local patterns in the strings, but for somewhat smaller problem sizes. For example, in regards to his branch and bound algorithm for the maximum weight trace formulation of multiple alignment Kececioglu [8] states "...we can solve instances on as many as 6 sequences of length 250 in a few minutes. These are among the largest instances that have been solved to optimality to date for any formulation of multiple sequence alignment."

To increase the utility of the branch and bound algorithm three improvements need to be made. One needed improvement would be to speed up the algorithm so that data sets with 20 or more long (length >= 300) sequences could be handled instead of the current limit of about 10. To address this we have considered searching a search tree in which the

nodes represent the displacement of windows relative to each other rather than the absolute positions of windows, as well as implementing the algorithm on a parallel machine. Unfortunately we have not significantly developed these ideas at this time.

The second improvement would be to include "regularizers" to adjust for small sample effects in the evaluation function. In general regularizers adjust the observed frequencies before using them to estimate probabilities. Many regularizers can be implemented by adding "pseudocounts" to the number of occurences of each character in an alignment column. For example the well known Laplacian regularizer adds one occurence of each character to the observed counts. Of course the probability estimates taken from the modified frequencies are normalized to add to one. A Laplacian regularizer can be used with the current implementation of our program by simply adding dummy sequences of length w, one for each character in the alphabet consisting entirely of that character as shown in figure 2.3. Since there is only one possible window position for the dummy sequences, their addition does not increase the search space. However Karplus [7] has shown that, at least for protein sequences, the Laplacian regularizer performs poorly. He finds Dirichlet mixture models to be the best choice. Mixture models allow one to express prior knowledge of the form "A" and "T" tend to occur together, therefore when one observes an "A" one might want to add a small amount to the count for "T" but not for "C" and "G". Indeed, as can be observed anecdotally in table 2.2, "A" and "T" do tend to occur together in columns. This is explained by the fact that for a DNA site to bind to a protein some disruption of the local DNA structure may be necessary; "A" and "T" bind by only two hydrogen bonds instead of the three bonds formed by "C" and "G" and therefore "AT rich" regions of DNA may be easier to deform. Unfortunately mixture models cannot be implemented by simply adding the same pseudocounts to each alignment. A compromise would be to use fractional pseudocounts, which Karplus found to be much better than a straight Laplacian regularizer. That would require some modification to our program but not to the basic algorithm.

The third improvement would be to incorporate the so called "ZOOPS" model proposed by Bailey and Elkan [3]. This model allows for the possibility that some sequences do not contain the motif. It is an attractive model computationally because it adds only one bit per sequence to the search space. Moreover the model can be used iteratively to locate motifs that occur a different number of times on different sequences, without any prior knowledge

```
AAAA
CCCC
GGGG
TTTT
CTACTGTATATAAAAACAGTATAA
ATACTGTGTATATATACAGTATTT
TTACTGTATATAAAACCAGTTTAT
```

Figure 2.3: The use of dummy sequences to implement a Laplacian regularizer is shown. In this example the window length would be four.

of the number of times the motifs occur.

In conclusion, we have developed a branch and bound algorithm for a widely used formulation of multiple sequence alignment. This algorithm has allowed problems to be solved to optimality that were previously amenable only to heuristic algorithms.

Bibliography

[1] D. J. Bacon and W. F. Anderson. Multiple sequence alignment. *Journal of Molecular Biology*, 191:153–161, 1986.

[2] Timothy L. Bailey. Likelihood vs. information in aligning biopolymer sequences. Technical Report CS93-318, UCSD, February 1993.

[3] Timothy L. Bailey and Charles Elkan. The value of prior knowledge in discovering motifs with meme. In *Proceeding of the Third International Conference on Intelligent Systems for Molecular Biology*, pages 21–38. AAAI Press, 1995.

[4] Otto G. Berg and Peter H. von Hippel. Statistical-mechanical theory and application to operators and promoters. *Journal of Molecular Biology*, 193:723–750, 1987.

[5] Robert G. Gallager. *Information Theory and Reliable Communication.* John Wiley & Sons, 1968.

[6] G. Z. Hertz, G. W. Hartzell, and G. D. Stormo. Identification of consensus patterns in unaligned dna sequences known to be functionally related. *CABIOS*, 6(2):81–92, 1990.

[7] Kevin Karplus. Evaluating regularizers for estimating distributions of amino acids. In *Proceeding of the Third International Conference on Intelligent Systems for Molecular Biology*, pages 188–196, Menlo Park, 1995. AAAI Press.

[8] John kececioglu. The maximum weight trace problem in multiple sequence alignment. In *Proceeding of Combinatorial Pattern Matching*, pages 106–119, 1993.

[9] C. E. Lawrence, S. F. Altschul, M. B. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: A gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, 1993.

[10] Charles E. Lawrence and Andrew A. Reilly. An expectation maximization (em) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences. *PROTEINS*, 7:41–51, 1990.

[11] T. Schneider, G. Stormo, L. Gold, and A. Ehrenfeucht. Information content of binding sites on nucleotide sequences. *Journal of Molecular Biology*, 188:415–431, 1986.

[12] Gary Stormo and George W. Hartzell III. Identifying protein-binding sites from unaligned dna fragments. *Nucleic Acids Research*, 86:1183–1187, 1989.

Chapter 3

Calculating Edit Distance with the four Russian's Technique

Abstract

In this paper we describe a technique which reduces the table size needed for using the Four Russians paradigm to compute the edit distance of two strings when the edit costs are symmetric, e.g. when the cost of indels and mismatches is 1 and the cost of a match is 0. When the length of the strings, n, is sufficiently long the savings is nearly a factor of $\sigma!$, where σ is the size of the alphabet. In the case of an infinite size alphabet the technique lowers the running time from $O(n^2)$ to $O(n^2/\Gamma^{-1}(n+1))$. In addition we note that in general it is more efficient to cache subproblems as they are needed rather than precompute all possible subproblems, some of which may turn out to be unnecessary. We present empirical results which demonstrate the merit of these two ideas.

3.1 Introduction

The four Russians paradigm, first introduced by [1] amounts to storing the answers to small subproblems in a table, and then using table lookups to solve a larger problem. The first work which applied this paradigm to the problem of computing string edit distance was that of [4] which succeeded in reducing the running time of edit distance computation from $O(n^2)$ to $O(n^2/\log^2 n)$ with a unit-cost RAM model or $O(n^2/\log n)$ with a traditional machine model. In practice however, the large size of the tables needed to store subproblems limits the usefulness of their technique. In this work we exploit the equivalency of groups of subproblems to reduce the storage requirements by nearly $\sigma!$, where σ is the size of the alphabet.

3.2 Four Russians for Edit Distance

In this section we briefly review the technique introduced in [4] for using the Four Russians technique to efficiently compute edit distance. The reader is referred to [3] for proofs and a more complete exposition. We define a submatrix to be a $t \times t$ submatrix of the standard edit distance dynamic programming table. Note that the lower and right edges of the submatrix are a function of four things: the upper and left edges of the submatrix and the substrings corresponding to those edges, as illustrated in figure 3.1. We will use a lookup table to compute the lower and right edges from the those four inputs. It is clear that if we can implement such a table then we can calculate the edit distance by looking up

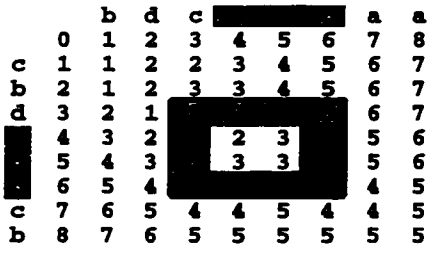


Figure 3.1: A submatrix of a standard dynamic programming table is shown. Note that the area and substrings shaded in light gray are sufficient to calculate the area shaded in darker gray

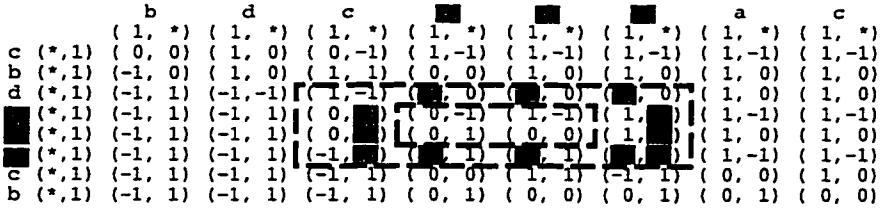


Figure 3.2: A submatrix of a dynamic programming table using relative offsets instead of absolute edit distances is shown. The first number of each ordered pair is the difference between the edit distance of its cell and the cell directly to its left. The second number of each ordered pair is the difference between the cell and the cell directly above it. Note that the entries and substrings shaded in light gray are sufficient to calculate the entries shaded in darker gray.

overlapping submatrices. The potential win comes because we only need to calculate the dynamic programming entries which correspond to the *insides* of the submatrices once for each possible set of inputs.

Assuming unweighted edit distance, we state without proof that adjacent cells in the standard dynamic programming table cannot differ by more than one. Thus the submatrix edge entries may be defined as offsets relative to their left or upper neighbors, with the actual value of the upper left hand corner entry being irrelevant to the task of computing the offsets. A typical submatrix of offset values is shown in figure 3.2. Since the offset entries must be in $\{-1, 0, 1\}$ the total number of possible offset inputs is $3^{2(t-1)}$. The submatrices are of size $t \times t$ but the upper and lower edges are provided as input so only the two corresponding substrings of length $t - 1$ are needed. Thus the overall number of possible inputs becomes $(3\sigma)^{2(t-1)}$. If t is set to be $\log_{3\sigma} n$ then the precomputation time becomes $O(n \log^2 n)$, which is less than the $O(n^2/\log^2 n)$ time needed to use the submatrix

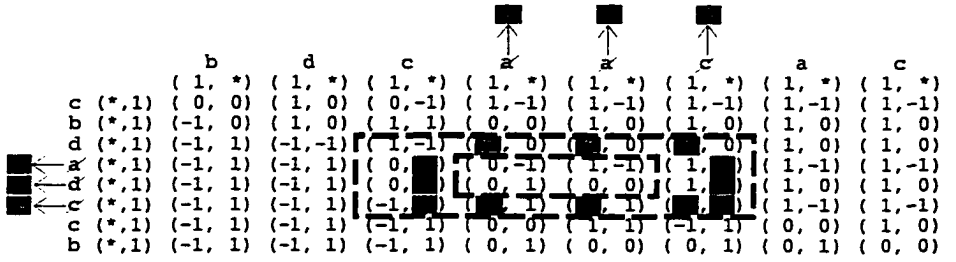


Figure 3.3: A submatrix of a dynamic programming table using the canonical form of substrings for input is shown. As in the previous figure the first number of each ordered pair is the difference between the edit distance of its cell and the cell directly to its left. The second number of each ordered pair is the difference between the cell and the cell directly above it. Again the entries and substrings shaded in light gray are sufficient to calculate the entries shaded in darker gray.

lookup table to do the actual edit distance calculation. The space needed for the lookup table becomes $O(n \log n)$.

3.3 Saving Space with Canonical Strings

3.3.1 Canonical Strings

The technique for saving space is a simple one. Given that the edit costs are symmetric we observe that the edit distance between two strings is unchanged when all the occurrences of any two characters are switched, this is because the edit distance depends only on where the matches and mismatches occur. We exploit this fact to reduce the number of substring pairs that must be stored for the four Russians algorithm. Let string S be the string formed by concatenating the 2 substrings that are part of an entry in the table used by the four Russians. We map S onto its canonical string T by associating the first character occuring in S with the first character of the alphabet and the second distinct character in S with the second character of the alphabet etc. Thus with the English alphabet both $S = zyyzx$ and $S = qccqg$ would be mapped to $T = abbac$. Normally the concatenation of two substrings is used for the table lookup, however we observe that when computing edit distance, instead of directly using the concatenation of the substrings of each submatrix for the table lookup it is possible to use the canonical forms of that concatenation instead. The same example submatrix shown before is shown again in figure 3.3 with $S = aacabc$ mapped to $T = aabadb$.

3.3.2 Savings

The motivation for defining a set of canonical strings is that there are fewer canonical strings of a given length than number of possible strings of the same length. For an alphabet of size σ the number of possible strings of length t is σ^t, however counting the number of canonical strings is less straightforward. For the small alphabet sizes it is possible to calculate the number of canonical strings as a function of their length t by *ad hoc* reasoning. For example with an alphabet of a, b the first character of the canonical string must be "a" and the succeeding characters can be either "a" or "b", so the number of strings is 2^{t-1}. With an alphabet of a, b, c we observe that only a's can appear before the first "b" and any character can appear after the first "b". Thus the number of strings with the first "b" in position i is 3^{t-i} and therefore the total number of strings with one or more b's in them is:

$$\sum_{i=2}^{t} 3^{t-1} = \sum_{i=0}^{t-2} 3^i = \frac{3^{t-1}-1}{2}$$

Since the first "b" can appear in any position from 2 to t. Adding one for the string of all a's gives

$$N(t,3) = \frac{3^{t-1}+1}{2},$$

where we define $N(t,\sigma)$ to be the number of canonical strings of length t for an alphabet size of σ. Table 3.1 shows the solutions for alphabet sizes of 2, 3, and 4, but as one can see this sort of *ad hoc* reasoning becomes difficult for larger values of σ. To obtain a general formula for calculating $N(t,\sigma)$ we define $S(t,\sigma)$ to be the number of canonical strings of length t in which *exactly* σ distinct characters appear. Clearly,

$$N(t,\sigma) = \sum_{i=1}^{\sigma} S(t,i).$$

It is not hard to see that the following recurrence must hold:

$$S(t,\sigma) = \sigma S(t-1,\sigma) + S(t-1,\sigma-1).$$

The first term on the right hand side represents extending a length $t-1$ canonical substring in which all σ characters have already appeared, this can be done by adding any of the σ characters. The second term represents extending a length $t-1$ canonical substring in which only the first $\sigma-1$ characters have appeared, this can only by done by adding the σth character. This recurrence is convenient to use with dynamic programming, which we have used to calculate $N(t,\sigma)$ for specific values of σ and t as shown in table 3.2.

Number of canonical strings for small alphabets

σ	$N(\sigma, t)$
2	2^{t-1}
3	$\frac{3^{t-1}+1}{2}$
4	$\frac{4^{t-1}+2}{6} + 2^{t-2}$

Table 3.1: Formulas for the number of canonical strings of length t are shown for alphabet sizes of 2, 3, and 4.

Although the recurrence above is sufficient to calculate values of interest a more direct summation formula may be obtained. In preparation we first state the following Lemma:

Lemma 1 *The number of possible strings of length $t \geq \sigma$ using all of σ characters at least once is $\sigma! \cdot S(t, \sigma)$, where $S(t, \sigma)$ is the Stirling number of the second kind defined by:*

$$(\sigma!) \cdot S(t, \sigma) = \sigma^t - \binom{\sigma}{1}(\sigma - 1)^t + \binom{\sigma}{2}(\sigma - 2)^t - \cdots + (-1)^{\sigma-1}\binom{\sigma}{\sigma-1}1^t$$

Proof: The basis of this lemma is an application of the inclusion-exclusion principle. A proof can be found in textbooks which include an introduction to combinatorics, for example see [2]. For completeness the proof of this lemma is also given in appendix 1.

Theorem 1 *The number of canonical strings of length t using exactly r characters is $S(t, r)$.*

Proof: Consider an equivalency class of strings defined by the operation of renaming characters. There are $r!$ orderings of the r names of the characters which appear, so each equivalency class has $r!$ members. Note that a string is a canonical string if and only if the order in which its characters first appear is the canonical order. Thus each equivalency class contains exactly one canonical string. Finally note that each string belongs to exactly one equivalency class. These facts combined with Lemma 1 complete the proof.

3.3.3 Complexity with Large Alphabet Sizes

The standard four Russians algorithm depends on a small alphabet size, for large alphabet sizes, e.g. $\sigma = \theta(n)$ the algorithm is completely ineffective. This is obvious by

Size of Savings for different alphabet sizes and substring lengths

σ	t	$N(\sigma,t)$	$\sigma^t/N(\sigma,t)$	$\sigma!$
3	4	14	5.79	6
3	6	122	5.98	6
4	4	15	17.07	24
4	6	187	21.90	24
4	8	2795	23.45	24
4	10	43947	23.86	24
20	4	15	1.1×10^4	2.4×10^{18}
20	6	203	3.2×10^5	2.4×10^{18}
20	8	4140	6.2×10^6	2.4×10^{18}
20	10	115975	8.8×10^7	2.4×10^{18}

Table 3.2: The number of canonical strings for different length strings and alphabet sizes σ are shown with the ratio of the number of possible strings without canonicalizing to the number of canonical strings. For ease of comparison the upper bound on that ratio, $\sigma!$, is also shown.

observing that the number of substrings of length two would already be $O(n^2)$. However the use of canonical strings reduces the table size so effectively with large alphabets that the asymptotic time complexity of computing edit distance for two strings of length n becomes $O(n^2/\Gamma^{-1}(n+1))$. Where Γ^{-1} denotes the inverse of the Γ function, $\Gamma(n+1) = n!$. Note that asymptotically $\Gamma^{-1}(n+1)$ grows slightly faster than $\frac{\log n}{\log\log n}$, (proved in Appendix 2). To prove this running time we note that the number of canonical strings of length t is bounded by $t!$. This is because the first character must be 'a', the second character must be in {'a', 'b'}, the third character must be in {'a', 'b', 'c'}, ... Using $t \times t$ submatrices the input strings are the concatenation of two length t substrings. Thus by setting $t = \Gamma^{-1}(n+1)/2$ we limit the number of possible input strings to no more than n. Each input string has $3^{2(t-1)}$ offset matrices, each of which can be computed in $O(t^2)$ time. Thus the overall precomputation time is $O((n)t^2 3^{2t})$ time. As proven in appendix 3 this is $o(n^2/t)$. With the precomputation finished the edit distance can be computed with $\frac{n^2}{(t-1)^2}$ table lookups. If constant time array access or hashing is allowed in the complexity calculation the canonical string can be computed in $O(t)$ time. This is certainly reasonable for alphabet sizes that are not infinite but still much too large for the standard four Russians algorithm. For example when $\sigma = \theta(n)$ an algorithm such as that shown in table 3.3 can be used to calculate the canonical string in $O(t)$ time. Thus the overall running time becouse $O(n^2/t)$. Again using the fact that $t = \omega(\frac{\log n}{\log\log n})$ this running time is $o(\frac{n^2 \log\log n}{\log n})$.

In some cases, even if the alphabet size is $\omega(n)$ the canonicalizing algorithm may be used. The number of distinct characters appearing in the strings is $O(n)$, so if individual characters can be compared in constant time it is possible to sort the set of characters which appear in the strings in $O(n \log n)$ time. This sorted list can then be used in place of the *Alphabet* array in table 3.3 for computing canonical substrings.

3.4 Empirical Study of Performance

3.4.1 On Demand Submatrix Calculation

For the asymptotic analysis we precomputed the values for all possible submatrices before looking at the actual input strings. This of course leads to the possibility of wastefully precomputing submatrices that aren't ever needed. In fact we would expect this to be common as some offset vectors, for example the one consisting entirely of $+1$, seem unlikely. An alternative approach is to calculate the submatrices the first time they are seen and enter them in the lookup table at that time. We employed this on demand approach for our empirical simulations. One additional benefit of using on demand calculation of submatrices is that even if t is foolishly chosen to be $\gg \log n$ the time and space required for the lookup table is still $O(n^2 t)$ instead of becoming exponential in n.

3.5 Simulation

We implemented the four Russians algorithm for computing edit distance with the option of using canonical strings in Perl. The program does compute the edit distance but its main function is to count the number of submatrices which need to be computed, (to emphasize the counting aspect of the program we will sometimes refer to it as the simulator.) To simplify the implementation the program truncates the input sequences as necessary to make their length be a multiple of $t - 1$, for example with $t = 4$ an input string of length 800 would be truncated to length 798.

3.6 Datasets

We used three data sets: an artificial data set, a set of 5S RNA sequences, and a set of DNA sequences containing LexA binding sites. The artificial sequences were randomly

```
// s is the input string.
// t is the canonical string to be computed.
// Alphabet is a static array of characters representing
//     the alphabet.
// Mapping is an array of pointers which point to
//     elements of Alphabet
// current_character indexes Alphabet
// Note that the alphabet is numbered from 1.
//     For example Mapping[ 'a' ] ≡ Mapping[ 1 ]
//
Alphabet ← [ 'a', 'b', ···, 'z' ]
current_character ← 1      // points to 'a'
Initialize all elements of Mapping to nil.
For i = 1 to length of s
    if( Mapping[ s[i] ] == nil )
        Mapping[ s[i] ] = Alphabet[ current_character ]
        increment current_character
    t[i] = Mapping [ s[i] ]
```

Table 3.3: Pseudocode for efficiently computing canonical strings is shown. The English alphabet is used for illustration.

generated using a uniform probability of each base at each position. The 5S RNA dataset consisted of 9 sequences of lengths 120 to 122 [5]. The LexA dataset consisted of 11 sequences of length 200. The same LexA dataset was described in chapter 1, but here we did not include the reverse complement of the sequences.

3.7 All Pairs Edit Distance

It is sometimes useful to compute the edit distance between all pairs of a set of sequences. For example some popular programs for generating multiple alignments or phylogenetic trees from nucleotide sequences compute the all pairs edit distance as a first step. It is clear that by aligning the same string many times the same pair of substrings is more likely to occur repeatedly We conducted experiments on the natural datasets to quantify this effect.

3.8 Algorithms and Cost Models

The four Russians algorithm is asymptotically faster than standard dynamic programming but will perform poorly if the time needed to look up a value in a large table is too long. Since the best tradeoff will depend on the architecture of the machine used, we analyzed our simulation results using a parameterized cost model. In our simplified model one unit of time is defined as the time needed to compute one cell of the standard string comparison dynamic programming table. With that scale, we define λ as the time required for a table lookup and τ as the time required per character for computing the canonical version of a string. We used these parameters to analyze the performance of four algorithms

- Standard Dynamic Programming (SDP)
- Standard four Russians (S4R)
- Canonicalized four Russians with no canonical string cache (C4R)
- Canonicalized four Russians with canonical string cache (C4RC)

C4R and C4RC differ in that C4RC keeps an additional lookup table with length $2(t-1)$ substrings as keys and their canonical versions as values. To keep the model manageable we also assume that the cost of looking up a value in this generally much smaller table is also λ.

Algorithms and their Costs

Algorithm	Cost
SDP	n^2
S4R	$t^2 m_r + \lambda r$
C4R	$t^2 m_c + 2(t-1)\tau r + \lambda r$
C4RC	$t^2 m_c + 2(t-1)\tau m_s + 2\lambda r$

Table 3.4: The parameterized running time of four string comparison algorithms under a simplified cost model is shown. The cost model and algorithms are described in the text.

We define m_r to be the number of entries that need to be calculated for the standard four Russians lookup table (or equivalently the number of cache misses), m_c to be the number of entries that need to be calculated for the canonicalized four Russians lookup table, and m_s to be the number of entries that need to be calculated for the canonical string lookup table used by C4RC. For convenience we denote the number of submatrices, $\frac{n^2}{(t-1)^2}$, by r. Note that by definition $r \geq m_r \geq m_c$ and $m_c \geq m_s$. These definitions can be used to calculate the running time of each algorithm as shown in table 3.4.

3.9 Results

3.9.1 Simulation Data

The data collected with the simulator using the artificial sequences is shown in table 3.5. The two natural datasets were used to quantify the effectiveness of the various algorithms on the all pairs edit distance problem. The data collected with the simulation for the 5S ribosomal RNA is shown in table 3.6. The data for the LexA dataset is shown in table 3.7.

3.9.2 Tradeoffs

Artificial Sequences

The memory use of the canonicalized and standard four Russians is shown in figure 3.4. This data convincingly demonstrates that the theoretical space savings gained from canonicalizing is realized in practice. The canonicalizing algorithm has smaller lookup tables for all input sizes but approaches its asymptotic size faster than the standard algorithm. The memory use directly reflects the number of cache misses for the submatrix

Required Lookup Table Sizes

length	t	submatrices without (M_r)	submatrices with (M_c)	substrings (M_s)
50	3	504	306	143
100	3	1664	641	256
200	3	4527	933	256
400	3	10089	1128	256
800	3	16296	1194	256
1600	3	19680	1208	256
3200	3	20601	1213	256
6400	3	20719	1214	256
15000	3	20731	1214	256
48	4	249	239	196
99	4	1023	936	621
198	4	3766	3042	1520
399	4	14072	9994	3024
798	4	49045	27029	4032
1600	4	156860	59667	4096
3200	4	457473	95428	4096
6399	4	1095502	119215	4096
15000	4	2082274	131912	4096
48	5	143	143	110
100	5	625	624	600
200	5	2475	2415	2162
400	5	9500	8984	6804
800	5	36967	32082	19170
1600	5	135895	108160	40602
3200	5	488264	372004	61750
6400	5	1726156	1167474	65280
15000	5	7890743	3896497	65536

Table 3.5: The results of aligning pairs of strings with equal lengths is shown. The number of submatrices with and without canonicalizing and the number of pairs of substrings are shown for different string lengths and values of t. The strings are randomly generated strings over an alphabet of size four.

5S RNA Required Lookup Table Sizes

effective length	t	submatrices without (M_r)	submatrices with (M_c)	substrings (M_s)
719	3	11818	1148	256
718	4	29436	15470	3682
717	5	23151	21033	11633

Table 3.6: The number of submatrices with and without canonicalizing and the number of pairs of substrings are shown for different values of t. The effective length is the square root of the sum of the product of the lengths of the strings in each pair.

LexA Dataset Required Lookup Table Sizes

effective length	t	submatrices without (M_r)	submatrices with (M_c)	substrings (M_s)
1483	3	18594	1213	256
1468	4	123620	51643	4065
1483	5	111799	92085	34182

Table 3.7: The number of submatrices with and without canonicalizing and the number of pairs of substrings are shown for different values of t. The effective length is the square root of the sum of the product of the lengths of the strings in each pair.

lookup table and therefore the cache hit rate is also higher for the canonicalizing algorithm. However, it does take time to obtain the canonical substrings either by calculation or by looking them up in a canonical string cache. Thus for any given input the fastest algorithm will depend on the value of the cost parameters λ and τ. We calculated that relationship for each dataset used with the value of τ fixed at one. Figures 3.5, 3.6, 3.7, 3.8, 3.9, 3.10 show the relationship between the time required and λ for the articial sequences. Figure 3.11 compares the performance of different algorithms across different t values for the pair of length 15000 sequences.

All Pairs

We investigated the performance of the algorithms for the all pairs alignment problem with two datasets. The cost curves for the 5S RNA data are shown in figures 3.12 and 3.13. As can be seen in figure 3.14, the standard four Russians with substrings of length 3 outperforms standard dynamic programming when λ is less than about 3. The cost curves for the LexA sequences are shown in figures 3.15 and 3.16. Figure 3.17 compares

the performance of algorithms for substrings of length 3 and 4.

3.9.3 Conclusions

The simulation results summarized in figures 3.11, 3.14, 3.17 show that for reasonable problem sizes the standard four Russians algorithm can be faster than the standard dynamic programming algorithm for λ values as high as 7. For modern architectures it seems reasonable that a table lookup with a key of 6 characters and six integers in $\{-1, 0, 1\}$ (for $t = 4$) could be done in the time required to compute 7 entries of the dynamic programming table. For the length 15,000 strings the canonicalized four Russians with $t = 5$ also performed better than standard dynamic programming for $\lambda < 4.5$, but not as well as the standard four Russians which did best with $t = 4$. Our choice of setting τ to one was somewhat arbitrary but it appears that for small values of t the time spent computing canonical strings is not justified by the increase lookup table hit rate. This makes some intuitive sense since by computing the canonical string one is investing $O(t)$ time in the hopes of saving $O(t^2)$ time by using the lookup table more effectively. However we should note here that the cost calculations were slightly biased against the canonicalizing algorithms because the same lookup time λ was used to compare methods, even though the smaller lookup tables used by the canonicalizing algorithms might yield faster lookup times.

Another issue that can be investigated with the simulation results is the expected increased effectiveness of the four Russians approach when applied to the all pairs edit distance problem. The data presented here suggest that that effect is not dramatic. For the 5S RNA sequences the number of submatrices (table 3.6) needed is slightly less than that expected from simply extrapolating the results of table 3.5 onto a string length of ≈ 718, but still more than that needed for artificial sequences of length 400. For the LexA dataset, with an effective length of 1483, the number of submatrices (table 3.7) required is also quite close to what one would expect from extrapolating from table 3.5.

We can also see that the time gained from computing submatrices on demand over precomputing all possible submatrices was not great. For $t = 3$ the number of canonicalized matrices saturates at 1214 which is only one less than the theoretical limit of $N(4, 2(t-1))3^{2(t-1)} = 15 * 3^4 = 1215$. Thus nearly all possible matrices are actually encountered in practice.

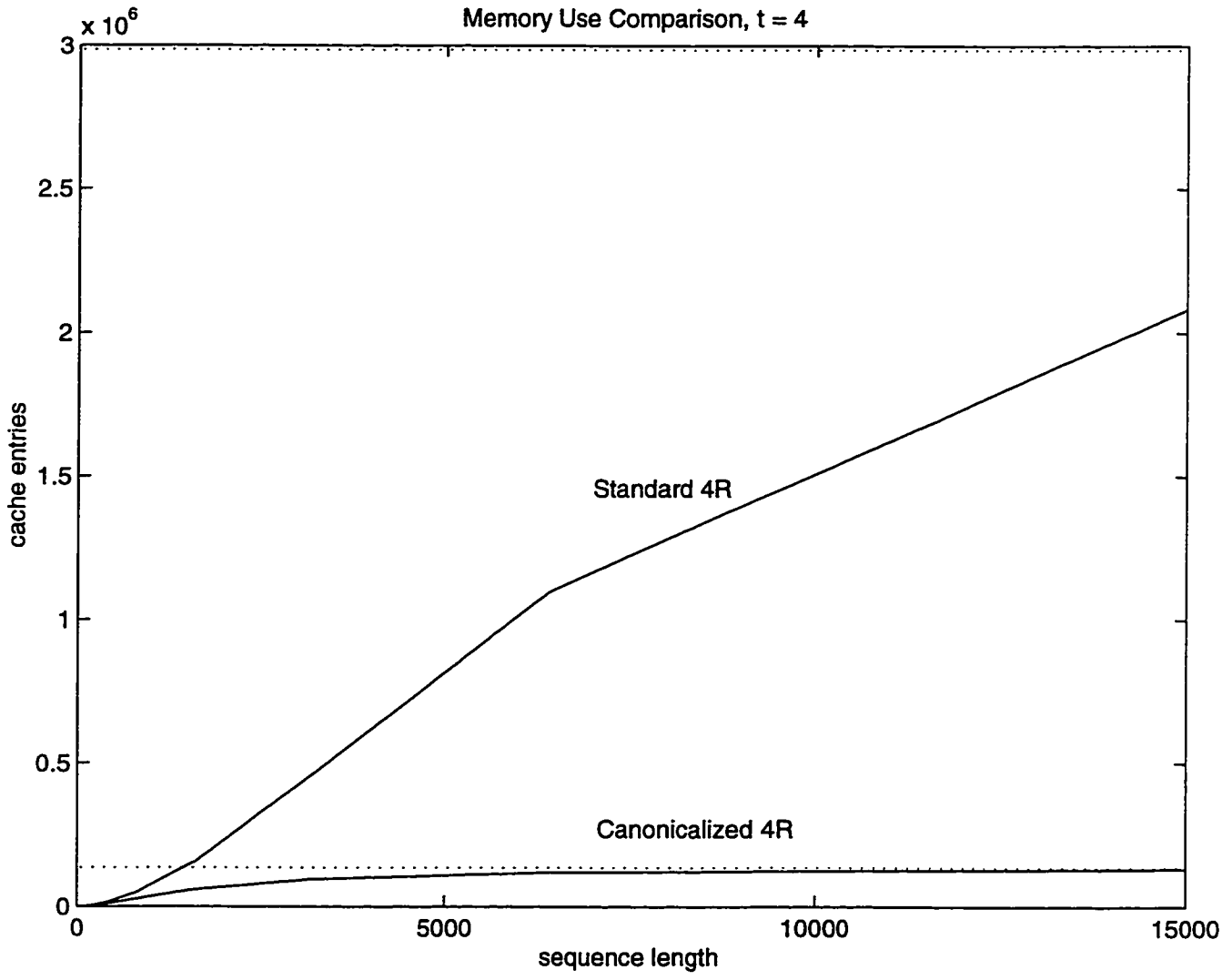


Figure 3.4: The memory use of the canonicalized and standard four Russians algorithm are shown for pairs of artificial sequences of different lengths. An upper bound on the asymptotic memory use of each is shown as a dotted line.

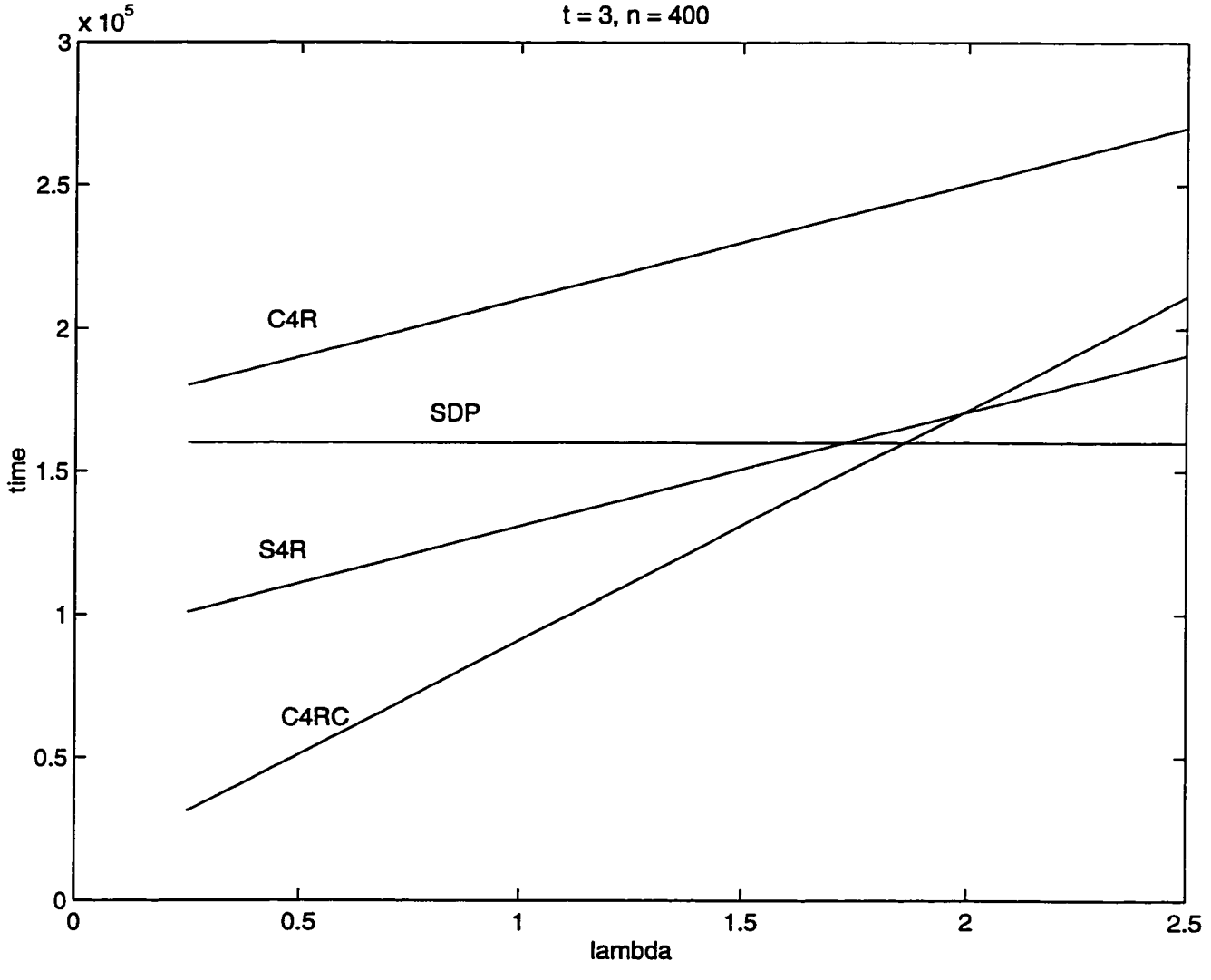


Figure 3.5: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The aligned strings were of length 400 and t was set to $t = 3$.

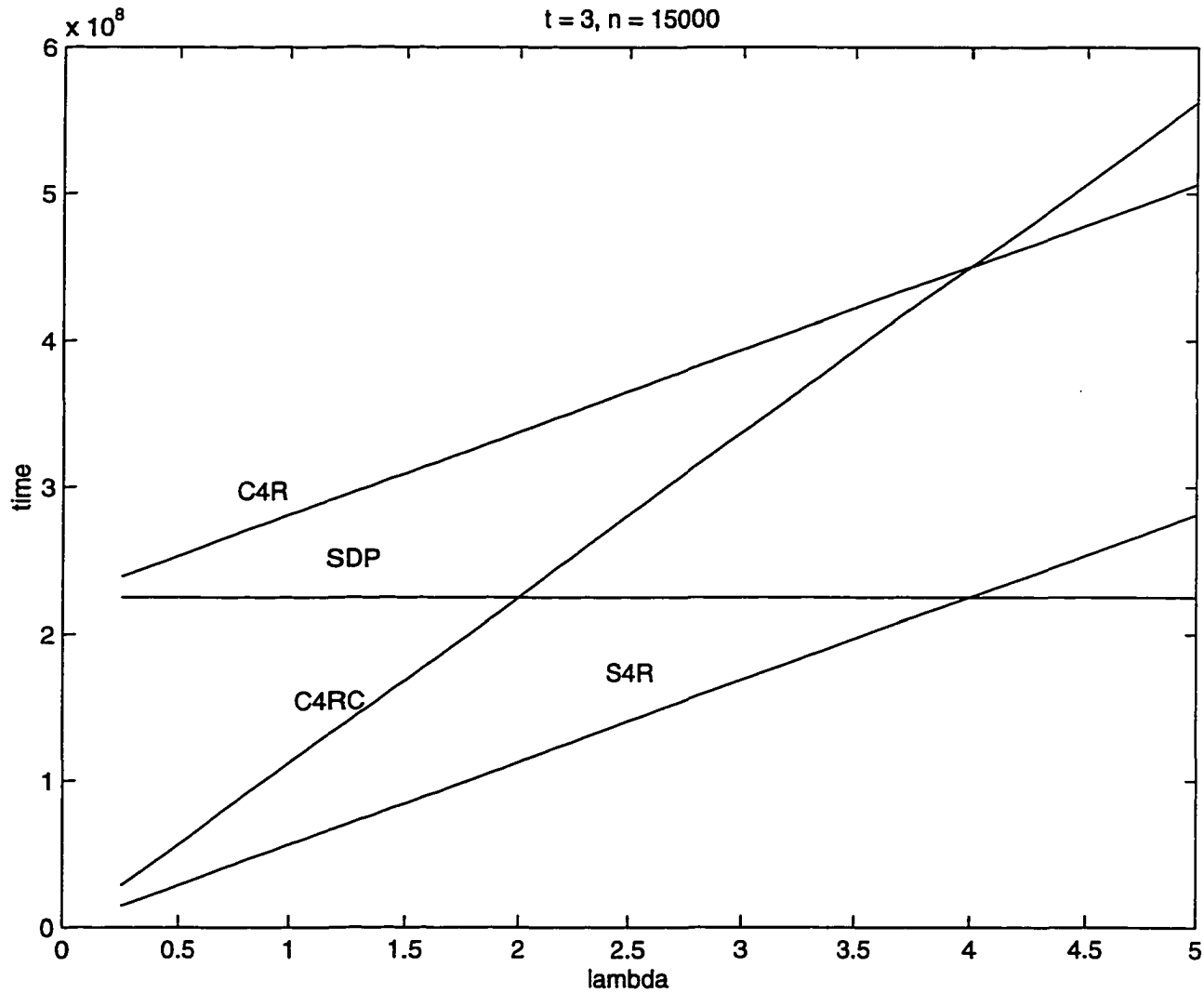


Figure 3.6: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The aligned strings were of length 15000 and t was set to $t = 3$.

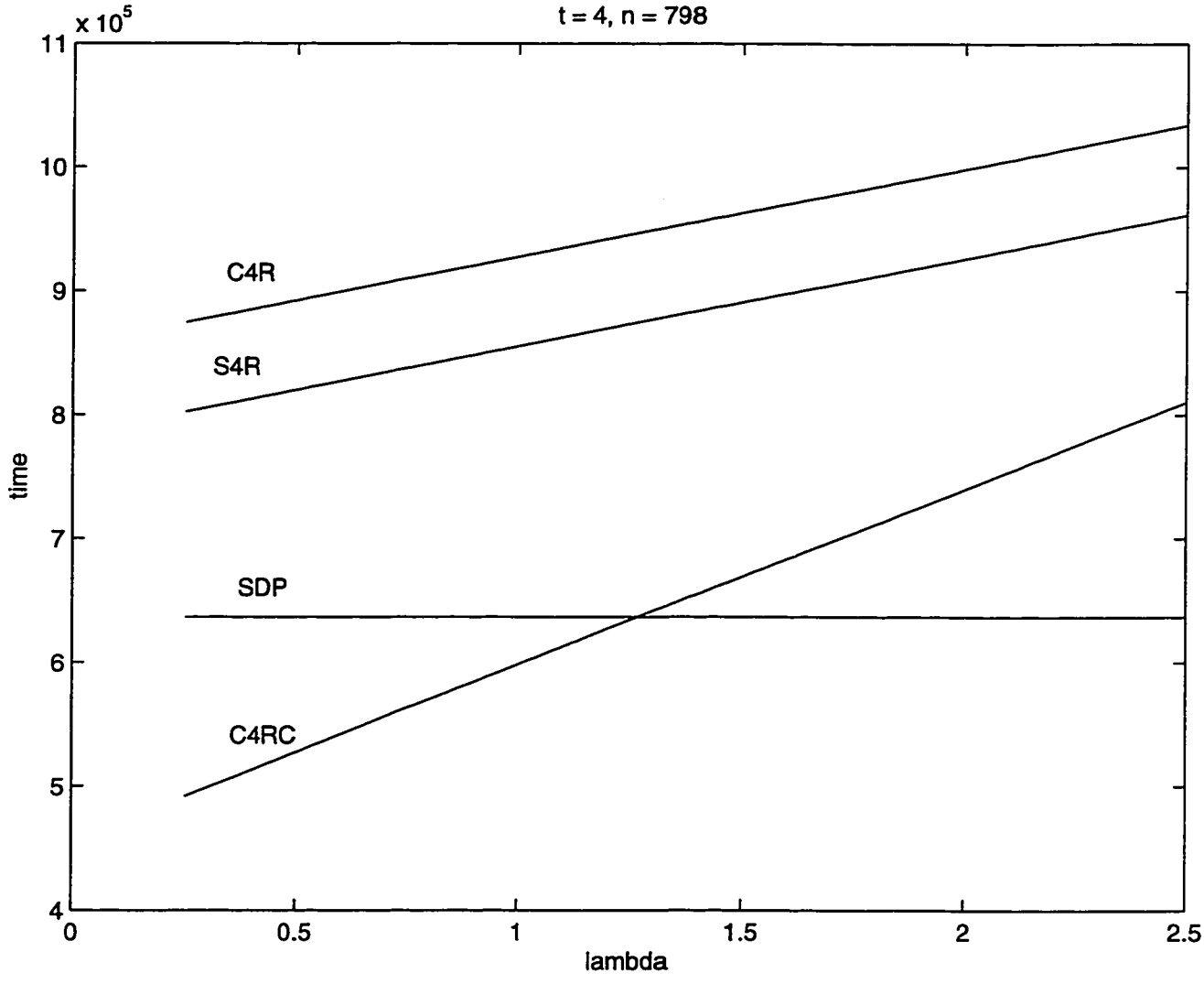


Figure 3.7: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The aligned strings were of length 798 and t was set to $t = 4$.

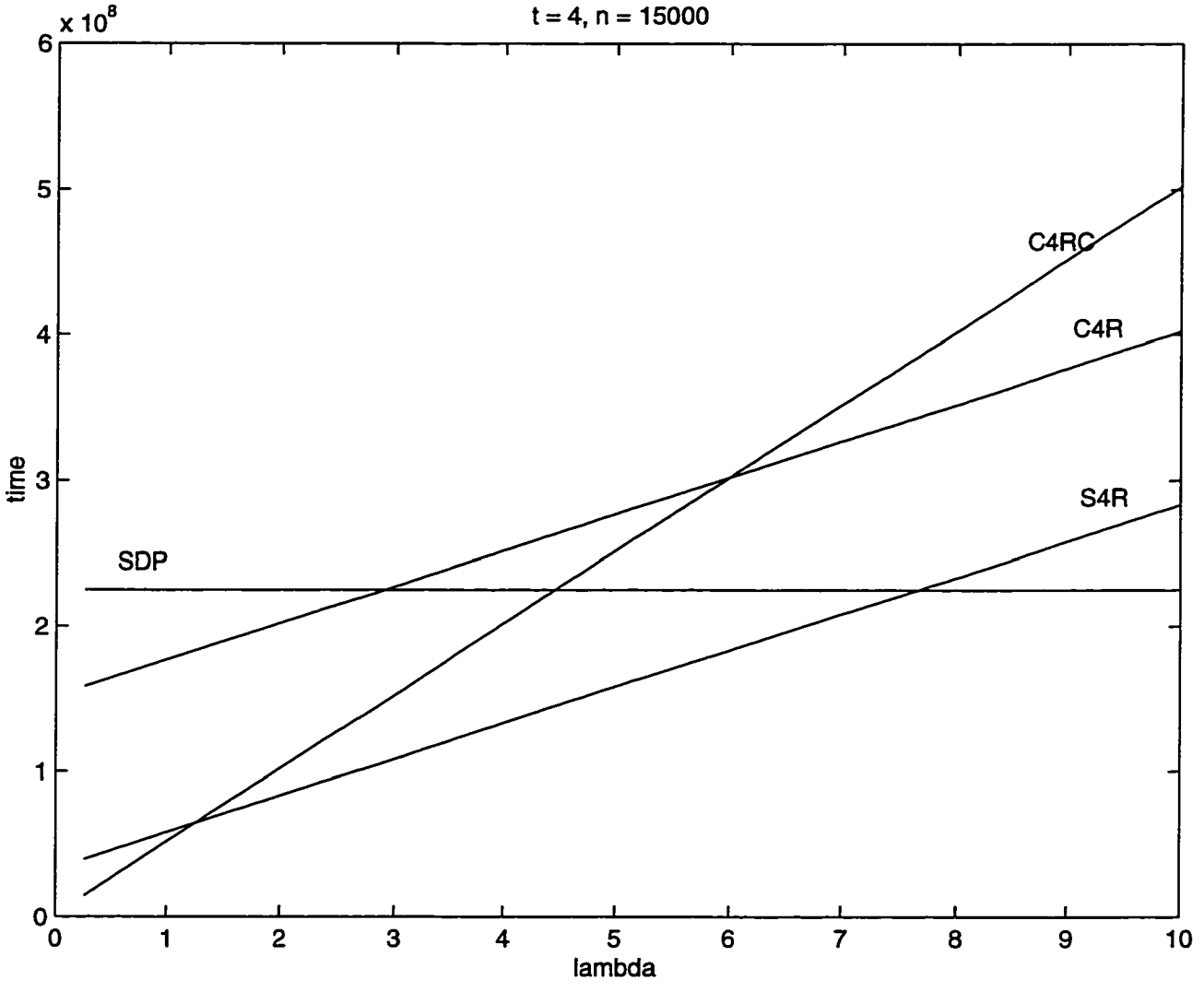


Figure 3.8: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The aligned strings were of length 15000 and t was set to $t = 4$.

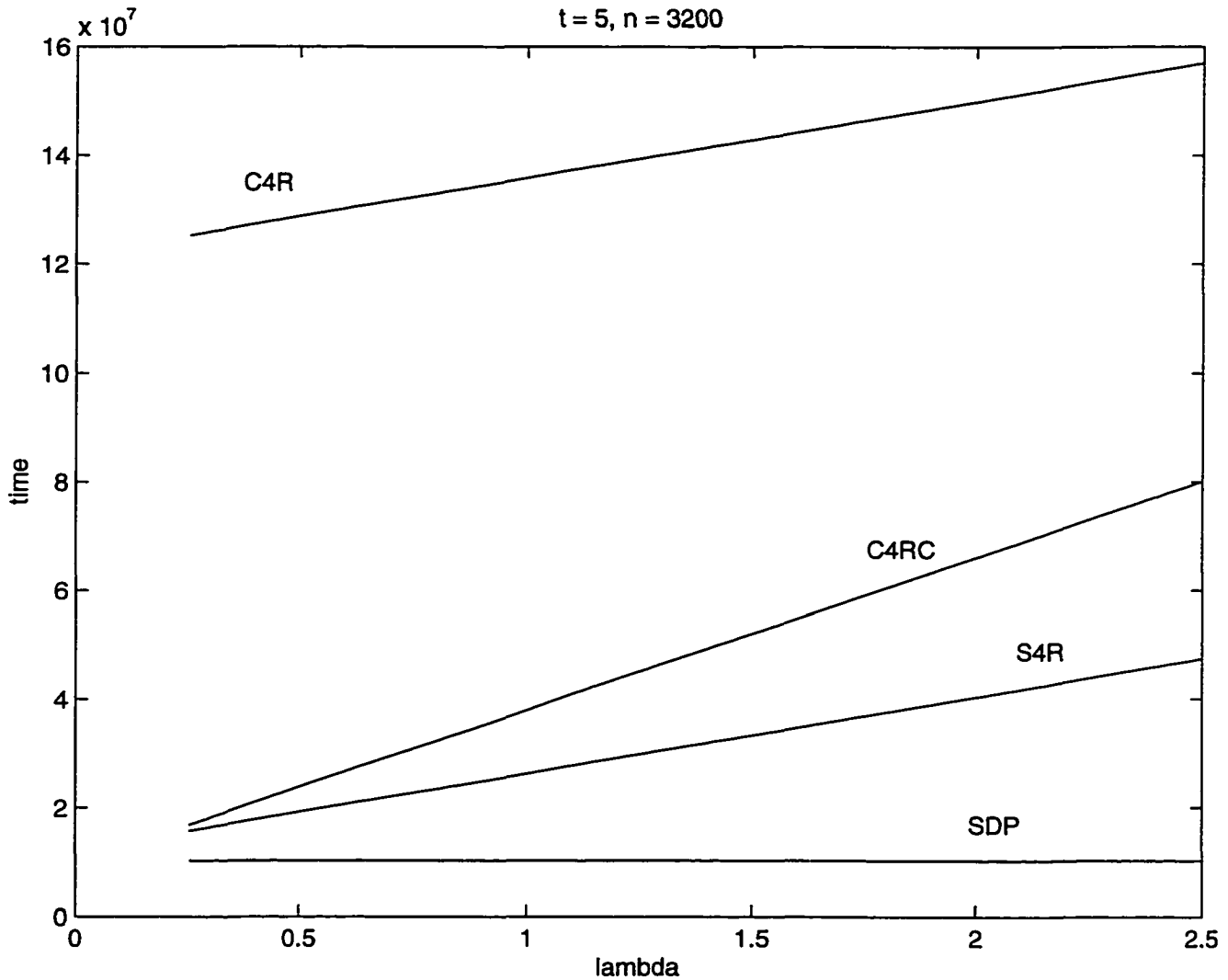


Figure 3.9: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The aligned strings were of length 3200 and t was set to $t = 5$.

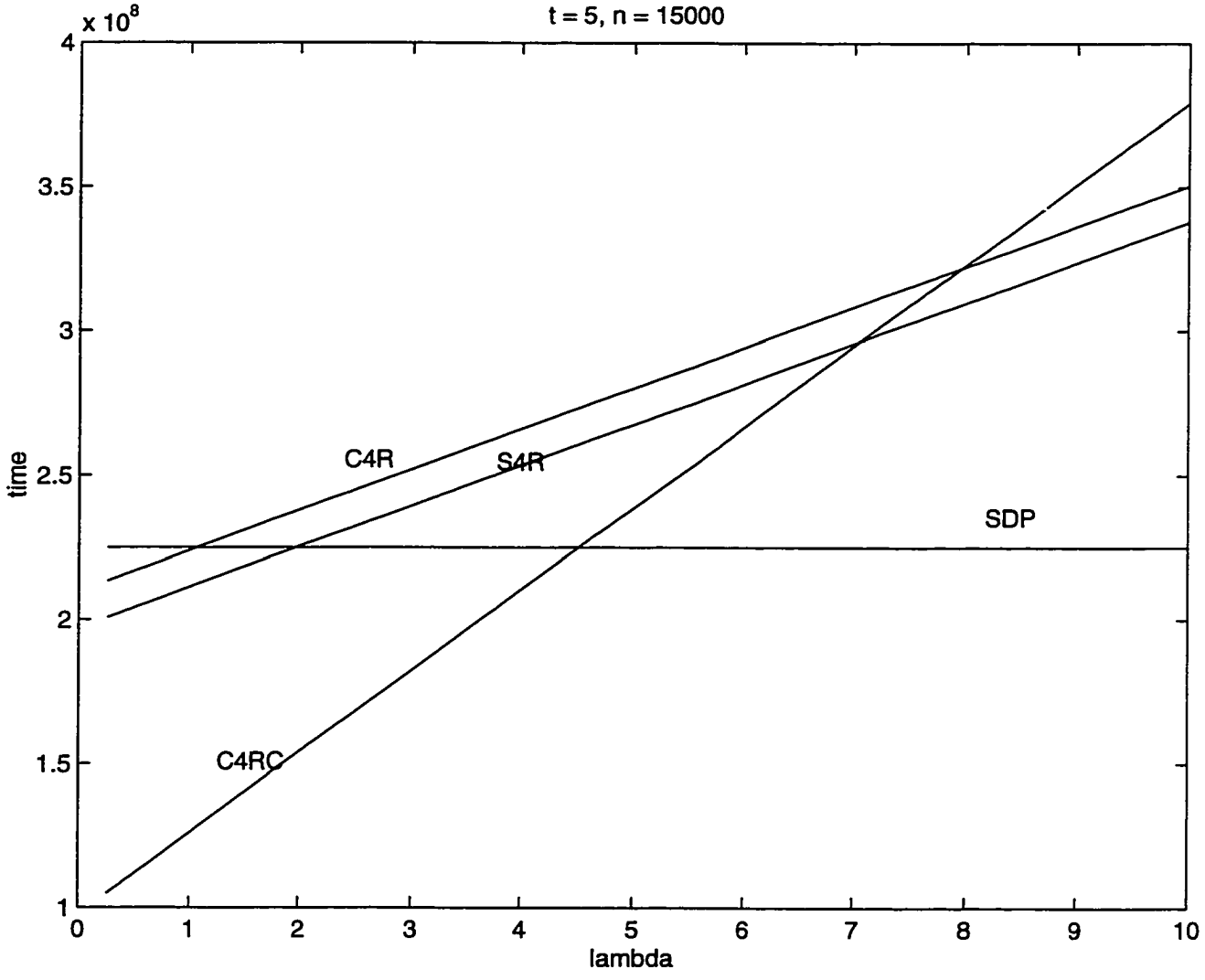


Figure 3.10: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The aligned strings were of length 15000 and t was set to $t = 5$.

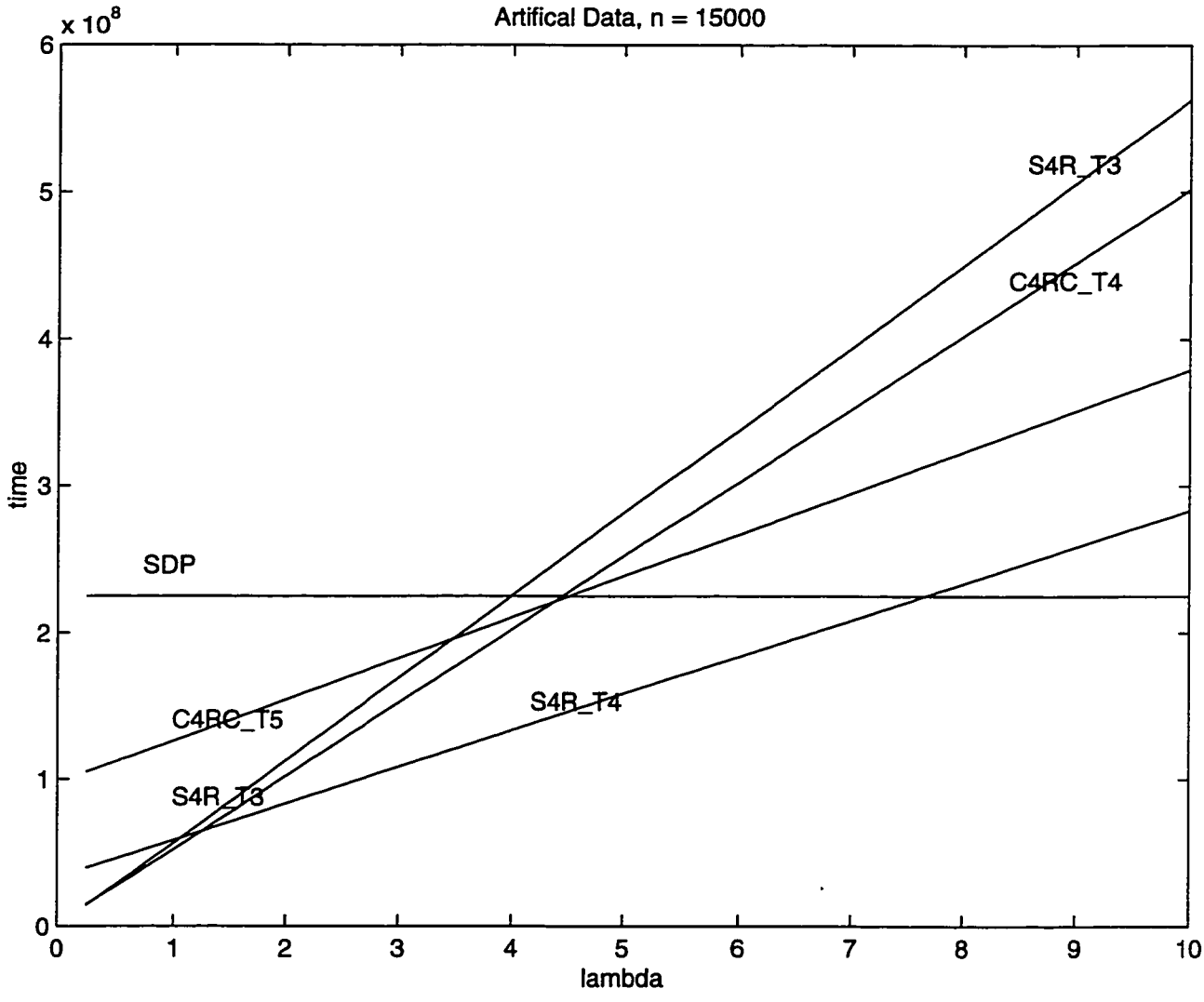


Figure 3.11: The time required for various values of λ is shown for different algorithms and values of t. For example S4R_T4 denotes the standard four Russians algorithm with $t = 4$.

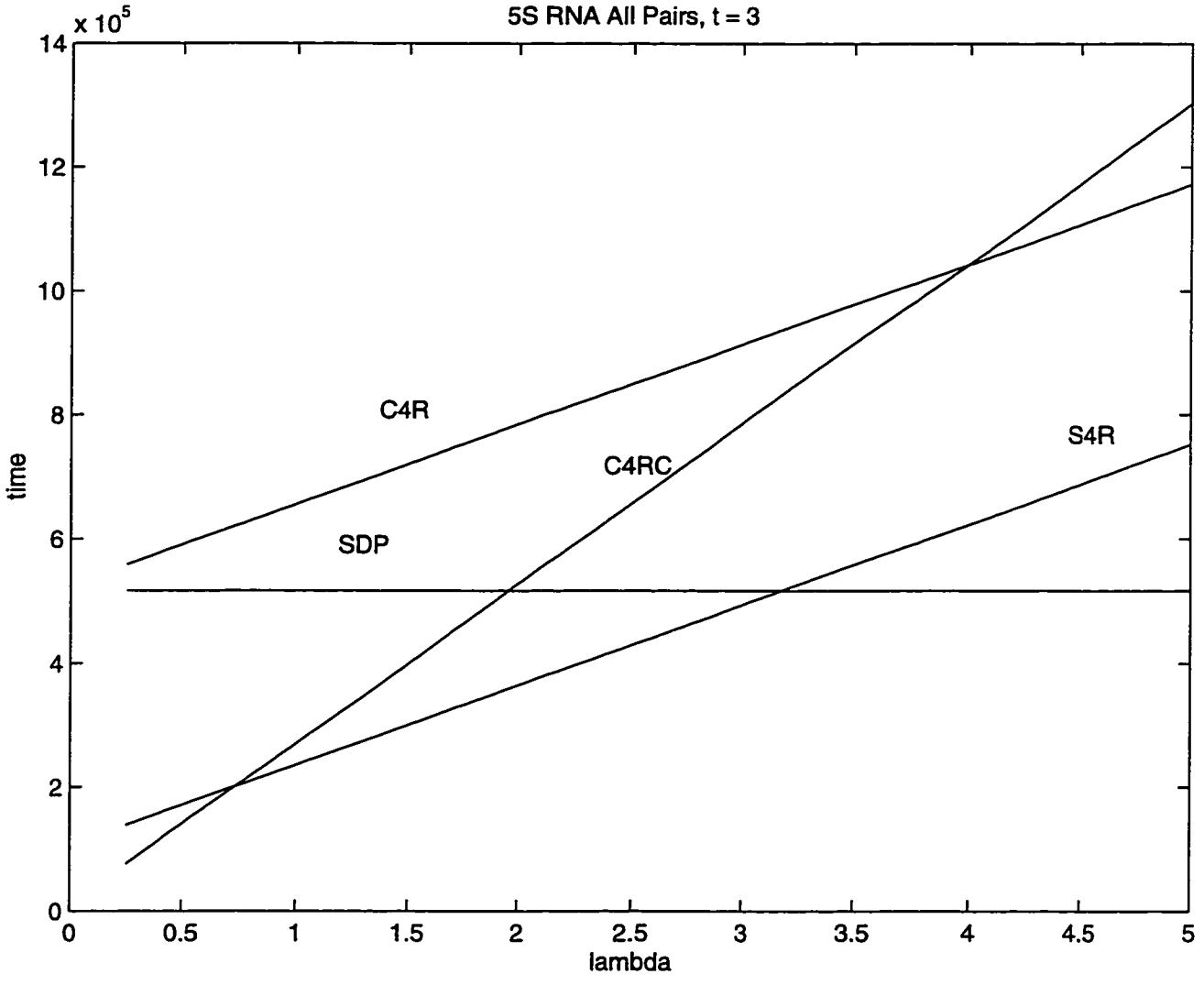


Figure 3.12: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The edit distance of all pairs of the 5S RNA sequences was computed with $t = 3$.

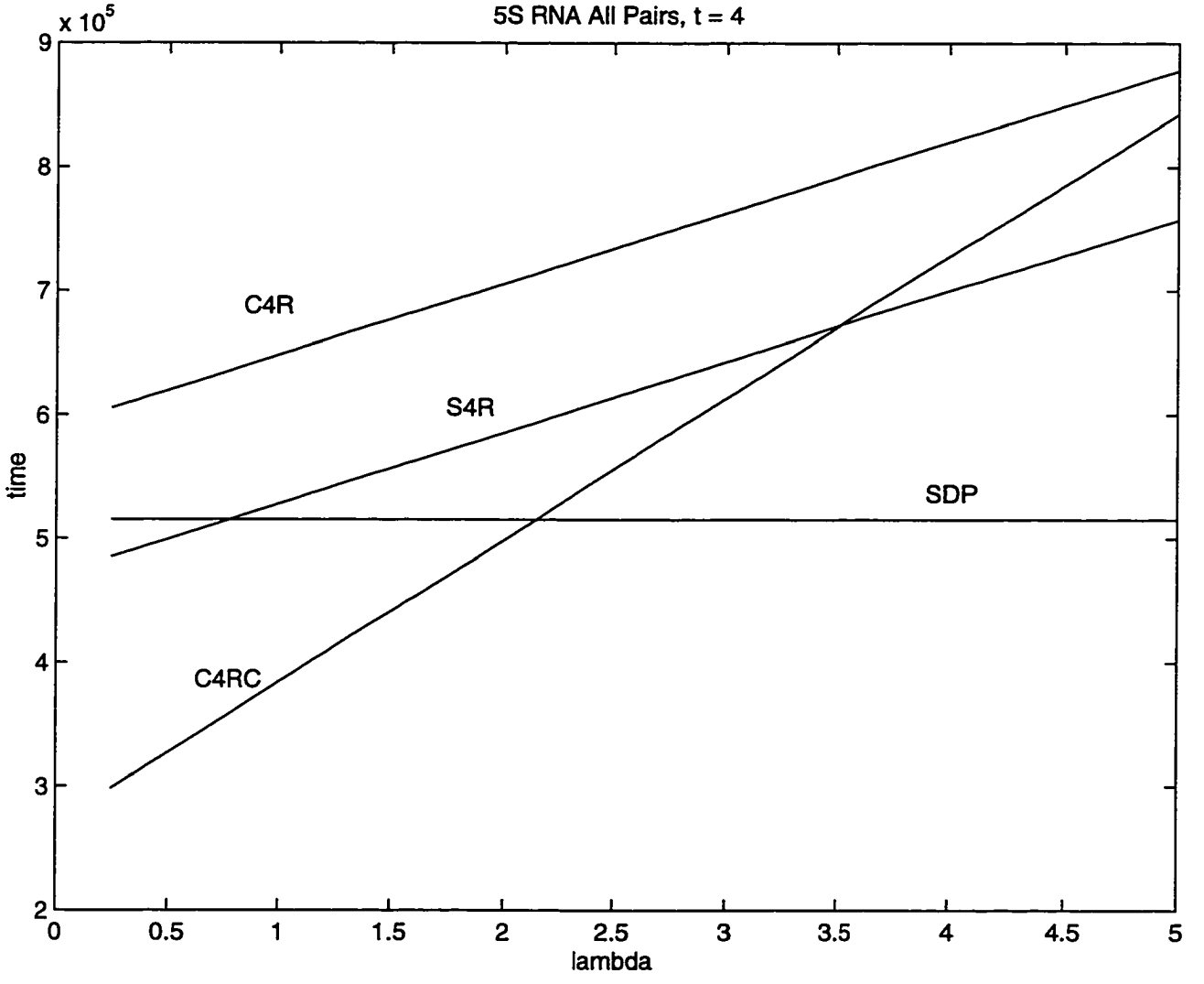


Figure 3.13: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The edit distance of all pairs of the 5S RNA sequences was computed with $t = 4$.

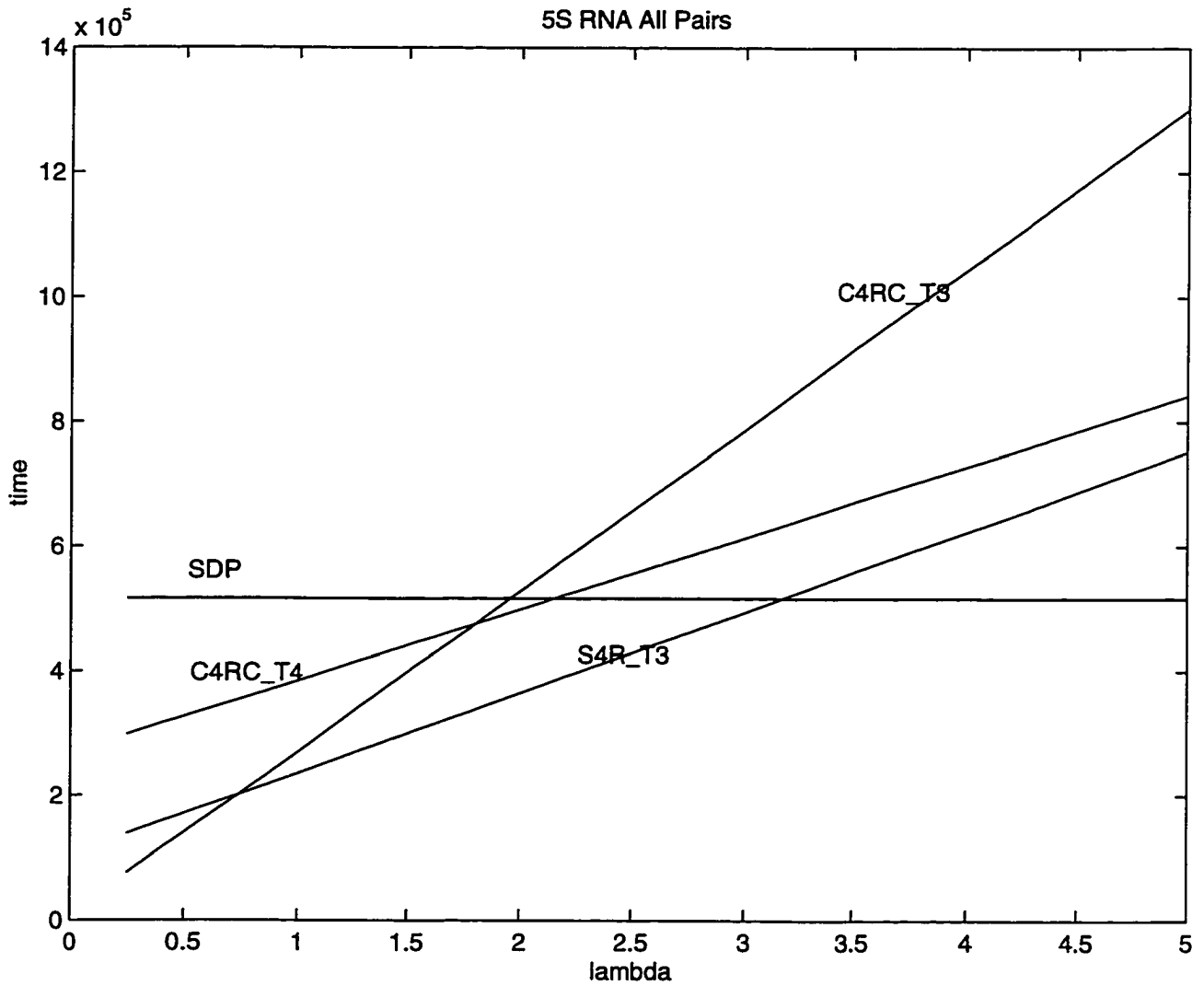


Figure 3.14: The time required to comput the edit distance of all pairs of the 5S RNA seqeunces for various values of λ is shown for different algorithms and values of t. For example S4R_T3 denotes the standard four Russians algorithm with $t = 3$.

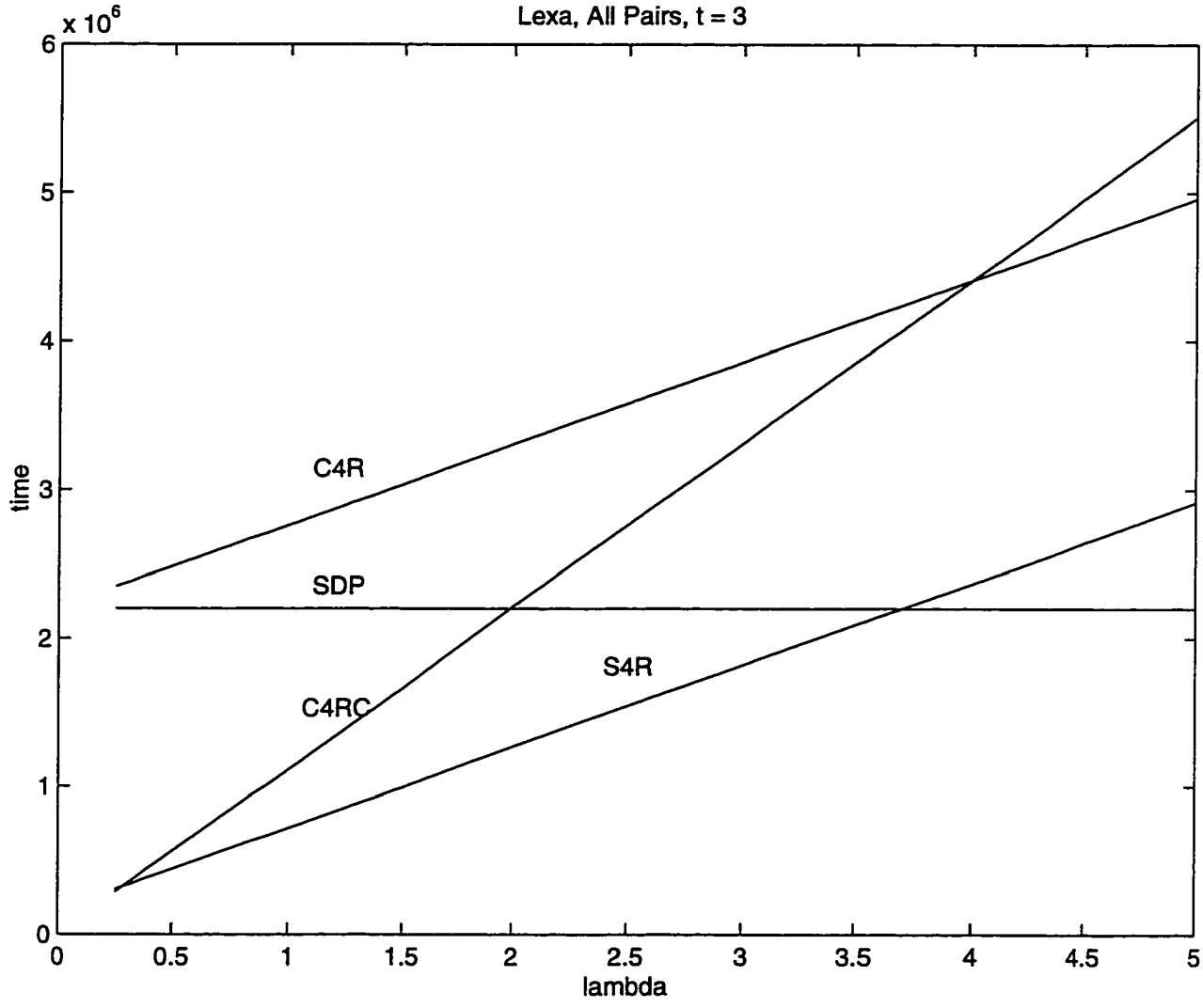


Figure 3.15: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The edit distance of all pairs of the LexA sequences was computed with $t = 3$.

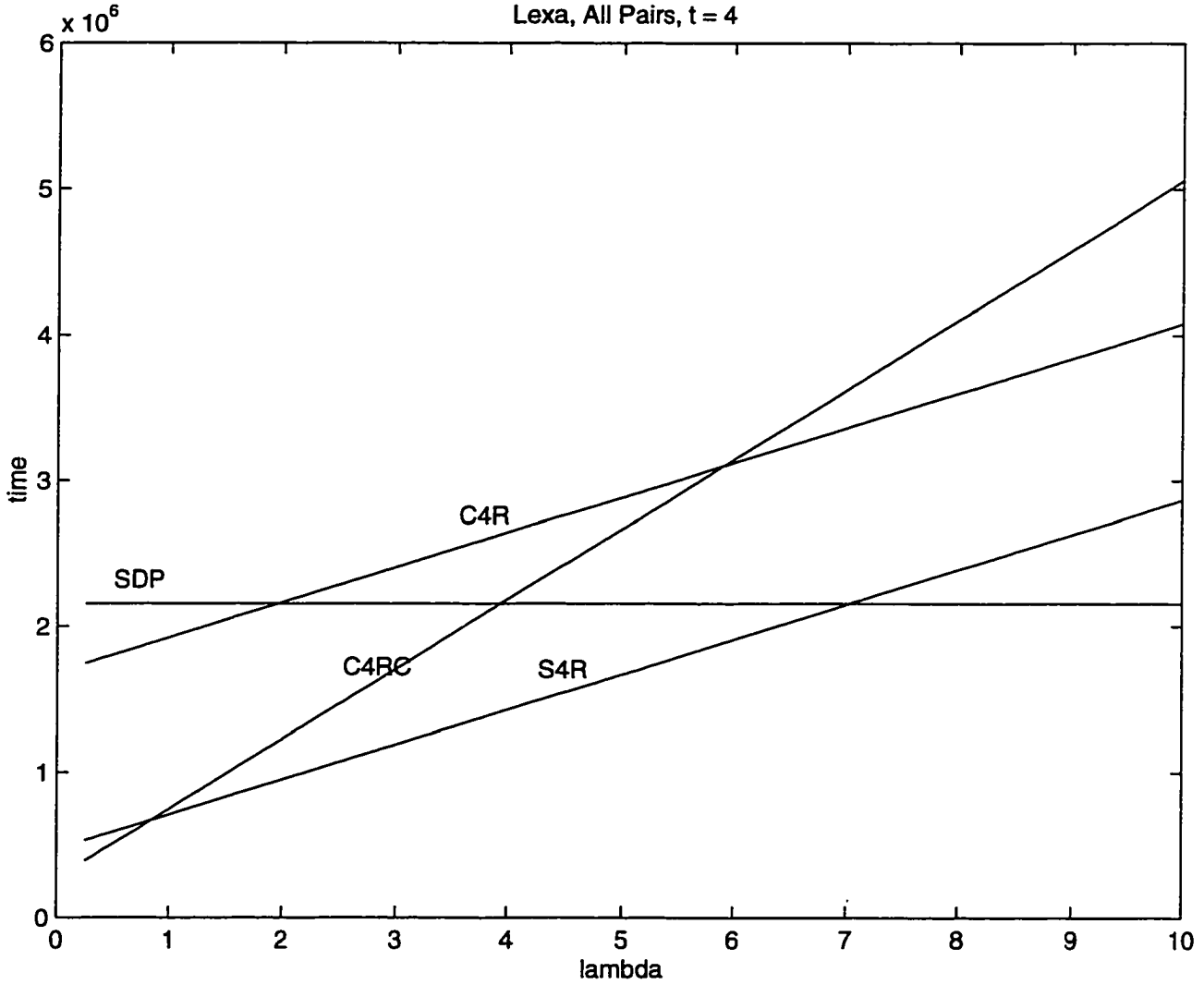


Figure 3.16: The time required for various values of λ is shown for dynamic programming, the standard four Russians algorithm, and the canonicalized four Russians algorithm with and without a separate cache for canonicalized strings. The edit distance of all pairs of the LexA sequences was computed with $t = 4$.

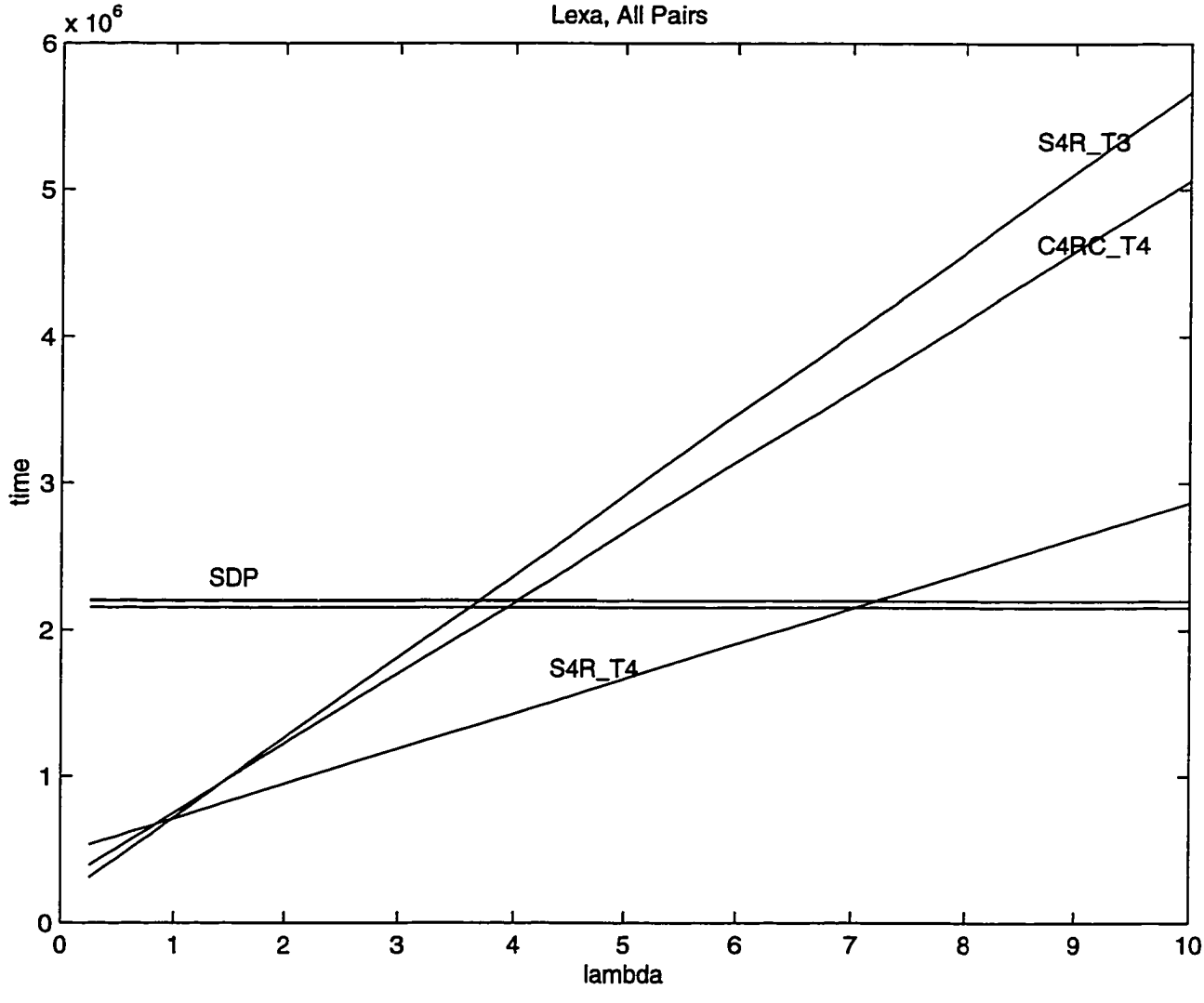


Figure 3.17: The time required to comput the edit distance of all pairs of the LexA seqeunces for various values of λ is shown for different algorithms and values of t. For example S4R_T4 denotes the standard four Russians algorithm with $t = 4$.

3.10 Summary

We have introduced a technique which uses a canonical representation of a pair of substrings to reduce the number of subproblems which a four Russians algorithm needs to precompute in order to compute the edit distance between two strings. For an alphabet of size of $\sigma = O(\log n)$ the technique saves a factor of up to $\sigma!$ space over the standard four Russians algorithm. For an alphabet size of $O(n)$ the technique makes it possible for a four Russians algorithm which is asymptotically faster than the standard dynamic programming algorithm, requiring $O(\frac{n^2(\log\log n)}{\log n})$ time.

We investigated the behavior of the canonicalizing and standard four Russians algorithm by generating empircal data. The data confirms that the canonicalizing algorithm does save space in practice. However for the problem sizes which our simulator could handle the potential time saving are not realized due to the overhead required in repeatedly computing or looking up canonical strings.

3.11 Acknowledgements

The author would like to thank Daniel S. Wilkerson and Saul Schleimer for pointing out that there is a simple relationship between the number of canonical strings of a given length and the Stirling number of the second kind.

Bibliography

[1] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dolk. Acad. Nauk SSSR*, 194:487–488, 1970.

[2] V. K. Balakrishnan. *Introductory Discrete Mathematics.* Prentice Hall, 1996.

[3] Dan Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge Press, 1997.

[4] W. J. Masek and M. S. Paterson. A faster algorithm for computing string-edit distances. *J. Comput. Syst. Sci.*, 20:18–31, 1980.

[5] D. Sankoff, R. Cedergren, and G. Lapalme. Frequency of insertion-deletion, tranversion and transition in the evolution of 5s ribosomal rna. *Journal of Molecular Evolution*, 7:133–149, 1976.

3.12 Appendix 1

In this appendix we prove:

Lemma 1 *The number of possible strings of length $t \geq \sigma$ using all of σ characters at least once is $\sigma! \cdot S(t, \sigma)$, where $S(t, \sigma)$ is the Stirling number of the second kind defined by:*

$$(\sigma!) \cdot S(t, \sigma) = \sigma^t - \binom{\sigma}{1}(\sigma - 1)^t + \binom{\sigma}{2}(\sigma - 2)^t - \cdots + (-1)^{\sigma-1}\binom{\sigma}{\sigma - 1}1^t \qquad (3.1)$$

Proof: Following [2] we prove this using the inclusion-exclusion principle. Let A_i, $i \leq \sigma$, represent the set of strings of length t in which the ith character appears. The quantity we are interested in is the cardinality of $A_1 \cap A_2 \cap \cdots \cap A_\sigma$. Let $U = A_1 \cup A_2 \cup \cdots \cup A_\sigma$ denote

our universe of strings. By the inclusion-exclusion principle we know that

$$\begin{aligned} |A_1 \cap A_2 \cap \cdots A_\sigma| &= |U| - |A_1' \cup A_2' \cup \cdots A_\sigma'| \\ &= |U| - \sum_i |A_i'| + \sum_{i<j} |A_i' \cap A_j'| - \sum_{i<j<k} |A_i' \cap A_j' \cap A_k'| + \cdots + (-1)^\sigma |A_1' \cap A_2' \cap \cdots \end{aligned}$$

The number of possible strings excluding any k characters is $(\sigma - k)^t$ for $1 \leq k < \sigma$. No strings are possible when all σ characters are excluded so the last term of equation 3.2 is zero. Combining these two facts with equation 3.2 gives equation 3.1, completing the proof.

3.13 Appendix 2

In this appendix we prove:

Theorem 2 $\Gamma^{-1}(n+1) = \omega(\frac{\ln n}{\ln \ln n})$

Proof: We prove that the inverse of n^n is $\omega(\frac{\ln n}{\ln \ln n})$. This is sufficient to prove the theorem because n^n grows faster than $n!$. Let $a = n^n$. We show that $n = \omega(\frac{\ln a}{\ln \ln a})$. This is done by taking logs and recursively substituting:

$$\log_n a = \frac{\ln a}{\ln n} = n.$$

Therefore

$$n = \frac{\ln a}{\ln n} = \frac{\ln a}{\ln \frac{\ln a}{\ln n}} = \omega(\frac{\ln a}{\ln \ln a}),$$

completing the proof.

3.14 Appendix 3

Theorem 3 $O((n)t^2 3^{2t}) = o(\frac{n^2}{t})$, *where* $t! = n$.

Proof Cancelling terms we see that it is enough to show that

$$O(t^3 3^{2t}) = o(n) = o(t!)$$

or equivalently,

$$l(t) = \lim_{t \to \infty} \frac{t!}{t^3 3^{2t}} = \infty$$

Since $t! < \frac{t}{e}^t$,

$$l(t) > \frac{(\frac{t}{e})^t}{t^3 3^{2t}} = \frac{t^{t-3}}{(9e)^t}$$

For $t > 81e^2$,

$$\frac{t^{t-3}}{(9e)^t} > \frac{(9e)^{2t-6}}{(9e)^t} = (9e)^{t-6}.$$

Thus the limit goes to infinity.

Chapter 4

Efficiency of the A^* Algorithm for Multiple String Alignment

Abstract

Multiple sequence alignment is an important tool for analyzing biological macromolecules. A well known dynamic programming algorithm can align n sequences of average length l in $O(l^n)$ time. Carrillo & Lipman introduced a bounding technique that eliminates portions of the n dimensional dynamic programming table from consideration and reported excellent speed-up from the use of those bounds. Those same bounds can be used in a different manner with the A^* algorithm. Kececioglu implemented a branch and bound algorithm which is very similar to the A^* algorithm in the well known MSA multiple sequence alignment program. Here we present an analysis which shows that the A^* algorithm dominates Carrillo & Lipman's preprocessing algorithm in the sense that the portion of the dynamic programming table which is eliminated by Carrillo & Lipman's algorithm is never expanded by an A^* algorithm using the same bounds. We also show that in general A^* expands fewer nodes than Kececioglu's branch and bound algorithm.

4.1 Introduction

Multiple alignment of sequences representing nucleic acid and protein sequences is useful in molecular biology [4]. An extension of the dynamic programming method for aligning two sequences (Needleman and Wunsch [9]) introduced by Murata *et al.* [8] can align n sequences of average length l in $O(l^n)$ time. However, dynamic programming has the drawback that its best case execution time is no better than its worst case execution time. Carrillo & Lipman [2] showed that by using a lower bound on the cost of aligning two sequences in the multiple alignment a preprocessing step can eliminate much of the n dimensional dynamic programming table from consideration. They reported a significant speed up of the best and average case execution time for the alignment of three or more sequences, up to a practical limit of around six sequences. Kececioglu [6], [5] used similar bounds with a branch and bound algorithm in his implementation of the MSA multiple sequence alignment program. In this paper we show that, while quite similar to Kececioglu's branch and bound algorithm, the well known A^* algorithm has a provable advantage over Kececioglu's algorithm. Furthermore we show that the A^* algorithm never expands a vertex that would be eliminated by the preprocessing step of Carrillo & Lipman's algorithm. Thus suggesting that, at least for some cases, the A^* algorithm provides a more efficient means of utilizing the bounds than Carrillo & Lipman's algorithm.

4.2 Algorithms for Multiple Alignment

In this section we describe what a multiple alignment is and introduce the notation that is needed to describe different kinds of alignments of strings and substrings and their costs. We then state the recursion that is the basis of the dynamic programming algorithm for multiple alignment. After that we describe the heuristic speed-up of Carrillo & Lipman. Finally we describe the application of the A^* algorithm to this problem and point out how A^* is superior to the similar branch and bound algorithm of Kececioglu [6], [5]. In the next section we present a proof that the A^* algorithm dominates the heuristic of Carrillo & Lipman.

4.2.1 Definitions and Notation

For simplicity, we will consider the case of aligning three sequences when presenting notation and proofs. The generalization of the proofs presented to the case of $n > 3$ sequences will always be straightforward. A formal definition of the multiple alignment problem is given by Carrillo & Lipman [2] so we will only briefly describe the problem. Let a projection of a multiple alignment onto two of its component sequences simply be the alignment of those two sequences within the multiple alignment. For illustration the projection of the alignment:

```
-eugene
marci-o
b-r-ice
```

onto the sequences "eugene" and "marcio" is simply:

```
-eugene
marci-o
```

The cost of a projection is the sum of the costs of its columns, with a *cost matrix* giving the cost of aligning any two characters in a column. The *sum of pairs* multiple alignment problem defines the cost of a multiple alignment as the sum of the costs of the projections it imposes. Note that the words "cost", "weight", and "distance" will be used synonymously throughout this chapter.

We will now introduce the notation necessary for the rest of this paper. The reader may want to skim this now and refer back to this section as necessary. Let I be a sequence of characters indexed from i_1 to i_{l_i}. Let the length q prefix of I be denoted (i_0, i_q) and the

length $l_i - q$ suffix, i.e. the suffix starting at the $i_q + 1$th character, be denoted (i_q, i_{l_i}). For aesthetic reasons the i, j, k are written in upper case when they occur alone. Also note that we have purposefully defined the first coordinate of the substring to be one less that the index of the first character in the string, for example the substring representing just the first character of I is denoted (i_0, i_1), this is done solely to avoid adding "+1" to half of the subscripts below. Let α denote an alignment of two or more sequences. When the alignment α has an optimal score over some projection of the sequences then α will be superscripted with an $*$ and subscripted with the sequences the alignment is being projected onto, e.g. $\alpha^*_{ij}[I, J, K]$ denotes an alignment of the sequences I, J, K in which the cost of the projection onto the sequences I and J is minimal. When the alignment α is optimal with respect to the total sum of pairs cost no subscript will be used. Analogously the cost of an alignment or a projection of an alignment will be denoted $c(\alpha)$ where the cost of a projection of an alignment is indicated by a subscript on the c. Again the subscript is dropped when the total sum of pairs cost is to be designated. For example, $c_{ij}(\alpha^*_{ij}[I, J, K])$ is the cost of the projection onto the sequences I and J of the alignment mentioned above, while $c(\alpha^*_{ij}[I, J, K])$ is the sum of pairs cost of the same alignment.

4.2.2 Multiple Sequence Alignment and Dynamic Programming

We can now use our notation to give the basic recurrence used for the dynamic programming algorithm. First consider aligning two prefixes (i_0, i_q) and (j_0, j_r). In the optimal alignment of these two prefixes there are three possible final columns; either characters i_q and j_r align, or i_q aligns against a space, or j_r aligns against a space. These three columns correspond to three possibilities: either the optimal alignment can be obtained by extending the optimal alignment of (i_0, i_{q-1}) and (j_0, j_{r-1}), or by extending the optimal alignment of (i_0, i_{q-1}) and (j_0, j_r), or by extending the optimal alignment of (i_0, i_q) and (j_0, j_{r-1}). Thus the recursion for two sequences becomes

$$c(\alpha^*[(i_0, i_q), (j_0, j_r)]) = \min \begin{cases} c(\alpha^*[(i_0, i_{q-1}), (j_0, j_{r-1})]) & + d(i_q, j_r) \\ c(\alpha^*[(i_0, i_{q-1}), (j_0, j_r)]) & + d(i_q, \text{`}-\text{'}) \\ c(\alpha^*[(i_0, i_q), (j_0, j_{r-1})]) & + d(\text{`}-\text{'}, j_r) \end{cases}$$

where $d(i_q, j_r)$ is the cost of aligning the characters i_q and j_r and $d(i_q, \text{`}-\text{'})$ is the cost of aligning the character i_q against an indel. The generalization to three sequences involves a similar recurrence but now there are seven possible columns to end the alignment of

three strings. For example the seven possible first columns of the alignment of "eugene", "marcio", and "brice" are:

e	e	e	-	e	-	-
m	m	-	m	-	m	-
b	-	b	b	-	-	b

The only remaining question is how to score a column that aligns more than two characters, e.g. what should $d(\text{'}e\text{'}, \text{'}m\text{'}, \text{'}b\text{'})$ be? More than one choice is possible but this papers deals exclusively with the *sum of pairs cost* model mentioned above, in which the cost of a column of a multiple alignment is defined as the sum of all the pairs of characters in the column. To illustrate with our running example, $d(\text{'}e\text{'}, \text{'}m\text{'}, \text{'}b\text{'})$ would be defined as $d(\text{'}e\text{'}, \text{'}m\text{'}) + d(\text{'}e\text{'}, \text{'}b\text{'}) + d(\text{'}m\text{'}, \text{'}b\text{'})$. The generalization to four or more sequences is straightforward.

Since each cell in the dynamic programming table represents one of the possible ways to choose a prefix from each sequence, the size of the table is equal to the product of the lengths of the sequences. Thus if the geometric mean length of n sequences is l, the size of the dynamic programming table would be approximately l^n. By evaluating the cells of the table in a topographical order each cell can be computed from its neighbors in $O(2^n)$ time. Consequently the execution time is exponential in the number of sequences to be aligned, making it generally impractical to align more than four sequences with a straightforward application of dynamic programming. This situation motivated Carrillo & Lipman to discover a useful way to reduce the work required.

4.2.3 Carrillo & Lipman's Algorithm

One weakness with the dynamic programming algorithm described above is that its best case running time is no better than its worst case running time. Since we have no reason to believe that nucleic acid or protein sequences would be designed by an adversary, one might hope to do better in practice than the worst case running time. Indeed, Carrillo & Lipman [2] succeeded in doing this. Their method identifies cells of the n dimensional dynamic programming table which can be eliminated from consideration in advance. In hindsight, their key observation is a simple one. First observe that a lower bound on the cost of aligning n sequences can be obtained by aligning the $\binom{n}{2}$ pairs of sequences *independently.* Recalling that we denote the cost of the optimal alignment of two sequences I and J by $c_{ij}(\alpha^*_{ij})$, we show a lower bound for the cost of any projection of an alignment as well as for the overall alignment. Namely, for any alignment of the n sequences the projection of

the alignment onto the sequences I and J has a cost of at least $c_{ij}(\alpha^*_{ij})$ and therefore the overall sum of pairs cost of the alignment is bounded from below by $L = \sum_{i<j} c_{ij}(\alpha^*_{ij})$. The general scheme in this exposition is to show that certain alignments can be eliminated from consideration. In particular all of the alignments which pass through certain cells of the dynamic programming table can be eliminated.

Theorem 4.2.1 (Carrillo & Lipman) $\forall i, j\ i \neq j\ c_{ij}(\alpha^*) - c_{ij}(\alpha^*_{ij}) \leq c(\alpha^*) - L.$

Proof: Note that $c(\alpha^*) - L$ is

$$\sum_{k<l} c_{kl}(\alpha^*) - c_{kl}(\alpha^*_{kl})$$

This sum includes the term $c_{ij}(\alpha^*) - c_{ij}(\alpha^*_{ij})$ and all terms in the sum are nonnegative. Thus the inequality holds.

Note that we do not need to know $c(\alpha^*)$ to use this theorem. We can use an upper bound U on the cost of aligning the n sequences, for example U might be the cost of a feasible alignment. We have $U \geq c(\alpha^*)$ and therefore

$$c_{ij}(\alpha^*) - c_{ij}(\alpha^*_{ij}) \leq U - L.$$

Hence we can eliminate any alignment α' for which

$$c_{ij}(\alpha') - c_{ij}(\alpha^*_{ij}) > U - L.$$

We use three sequences (I, J, and K) to illustrate the application of this inequality. To emphasize the three strings we write $c_{ij}(\alpha^*_{ij})$ as $c_{ij}(\alpha^*_{ij}[I, J, K])$. Let π_{qrs} denote three prefixes of I, J, K; i.e. let π_{qrs} replace the notation $(i_0, i_q), (j_0, j_r), (k_0, k_s)$. Similarly let σ_{qrs} denote the suffixes which complete the strings, i.e. let σ_{qrs} replace the notation $(i_q, i_{l_i}), (j_r, j_{l_j}), (k_s, k_{l_k})$. Furthermore let α' be any alignment which passes through the dynamic programming table cell which corresponds to characters i_q, j_r, and k_s, denoted cell (q, r, s). Equivalently α' is the concatenation of an alignment of π_{qrs} with an alignment of σ_{qrs}. This implies that

$$c_{ij}(\alpha') \geq c_{ij}(\alpha^*_{ij}[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}])$$

Thus we can eliminate any cell (q, r, s) if

$$c_{ij}(\alpha^*_{ij}[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) - c_{ij}(\alpha^*_{ij}[I, J, K]) > U - L$$

Rearranging terms and defining δ as $\delta \equiv U - L$ we obtain: eliminate if

$$c_{ij}(\alpha^*_{ij}[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) > c_{ij}(\alpha^*_{ij}[I, J, K]) + \delta$$

Where the analogous inequalities for the projections onto (I, K) and (J, K) are also sufficient conditions for elimination. Note that $c_{ij}(\alpha^*_{ij}[\pi_{qrs}])$ is independent of K and can be computed by filling in the two dimensional dynamic programming table of I versus J. Likewise $c_{ij}(\alpha^*_{ij}[\sigma_{qrs}])$ can be computed by filling in the dynamic programming table of the strings I and J *reversed.* This preprocessing is done for each pair of sequences in total time $O(n^2l^2)$.

We end this section by showing how the bound can be used to eliminate cells for two toy alignment problems. To keep things simple, we use the edit distance cost matrix to score our alignment, in this scheme a match costs zero and a mismatch or indel costs one. First we use our running example of the strings "eugene", "mario", and "brice". Using edit distance as our cost matrix, the eyeballed alignment shown previously can be verified to cost 18. This gives us an upper bound of $U = 18$. It can also be verified that the cost of independently aligning "eugene" and "mario" is 6, while the other two pairs cost 5. This gives us a lower bound of $L = 16$. This implies that the cost of the projection of "eugene" and "mario" in the optimal three-way alignment must be no more than 8 and the other two projections must cost no more that 7 each. The eliminated projections are shown in table 4.1. The table refers to two-way alignments, which correspond to possible projections in our three string multiple alignment.

It can be seen from the asterisks in table 4.1 that roughly half of the area of the two dimensional tables are eliminated. Moreover it turns out that the optimal alignment does in fact achieve the lower bound of 16. As shown in table 4.2, if we had know that at the start we could have eliminated much more.

Carrillo & Lipman's algorithm uses preprocessing to eliminate some regions of the multiple sequence dynamic programming table. Kececioglu [6] [5] described another algorithm that utilizes the same type of lower bound to reduce the work required. We will describe the A^* algorithm and its application to multiple sequence alignment and then describe the relationship between A^* and Kececioglu's branch and bound algorithm. The A^* algorithm is used as a search algorithm and is described in many artificial intelligence textbooks [10], [11]. However, for the reader's convenience we summarize the algorithm here.

		e	u	g	e	n	e
	0	1	2	3	4	5	6
m	1	1	2	3	4	5	6
a	2	2	2	3	4	5	6
r	3	3	3	3	4	5	6
c	4	4	4	4	4	5	6
i	5	5	5	5	5	5	6
o	6	6	6	6	6	6	6

		e	u	g	e	n	e
	0	1	2	3	4	5	6
b	1	1	2	3	4	5	6
r	2	2	2	3	4	5	6
i	3	3	3	3	4	5	6
c	4	4	4	4	4	5	6
e	5	4	5	5	4	5	5

		b	r	i	c	e
	0	1	2	3	4	5
m	1	1	2	3	4	5
a	2	2	2	3	4	5
r	3	3	2	3	4	5
c	4	4	3	3	3	4
i	5	5	4	3	4	4
o	6	6	5	4	4	5

		e	n	e	g	u	e
	0	1	2	3	4	5	6
o	1	1	2	3	4	5	6
i	2	2	2	3	4	5	6
c	3	3	3	3	4	5	6
r	4	4	4	4	4	5	6
a	5	5	5	5	5	5	6
m	6	6	6	6	6	6	6

		e	n	e	g	u	e
	0	1	2	3	4	5	6
e	1	0	1	2	3	4	5
c	2	1	1	2	3	4	5
i	3	2	2	2	3	4	5
r	4	3	3	3	3	4	5
b	5	4	4	4	4	4	5

		e	c	i	r	b
	0	1	2	3	4	5
o	1	1	2	3	4	5
i	2	2	2	2	3	4
c	3	3	2	3	3	4
r	4	4	3	3	3	4
a	5	5	4	4	4	4
m	6	6	5	5	5	5

		e	u	g	e	n	e
		7	8	*	*	*	*
m	7	6	7	8	*	*	*
a	8	7	6	7	8	*	*
r	*	8	7	6	7	8	*
c	*	*	8	7	6	7	8
i	*	*	*	8	7	6	7
o	*	*	*	*	8	7	6

		e	u	g	e	n	e
		5	6	7	*	*	*
b	6	5	5	6	7	*	*
r	7	6	5	5	6	7	*
i	*	7	6	5	5	6	*
c	*	*	7	6	5	5	7
e	*	*	*	*	6	6	5

		b	r	i	c	e
		6	7	*	*	*
m	5	5	6	7	*	*
a	6	5	5	6	*	*
r	7	6	5	5	7	*
c	*	7	5	5	5	6
i	*	*	7	5	5	5
o	*	*	*	6	5	5

Table 4.1: Dynamic programming tables for the three possible pairings of the strings "eugene", "marcio, and "brice" are shown. The top row shows edit distance for aligning prefices, the middle row shows edit distance for aligning suffices, and the bottom row gives the minimum cost of alignments which align the respective characters in the same column. In the last column an asterisk was used when the minimum cost was greater than a known upper bound on the cost of the two-way alignments.

		e	u	g	e	n	e
		*	*	*	*	*	*
m	*	6	*	*	*	*	*
a	*	*	6	*	*	*	*
r	*	*	*	6	*	*	*
c	*	*	*	*	6	*	*
i	*	*	*	*	*	6	*
o	*	*	*	*	*	*	6

		e	u	g	e	n	e
		5	*	*	*	*	*
b	*	5	5	*	*	*	*
r	*	*	5	5	*	*	*
i	*	*	*	5	5	*	*
c	*	*	*	*	5	5	*
e	*	*	*	*	*	*	5

		b	r	i	c	e
		*	*	*	*	*
m	5	5	*	*	*	*
a	*	5	5	*	*	*
r	*	*	5	5	*	*
c	*	*	5	5	5	*
i	*	*	*	5	5	5
o	*	*	*	*	5	5

Table 4.2: The last row of the previous table is shown here, modified to reflect the effect of knowing better upper bounds on the cost of the two-way alignments.

The A^* algorithm can be seen as a generalization of Dijkstra's shortest path algorithm [3]. The problem is to find the shortest path in a weighted graph $G(V, E)$ from a designated start vertex s to a designated goal vertex γ. The weights of G are constrained to be nonnegative. Additionally we are provided with a lower bound $\hat{h}_u$ on the distance from each vertex u to γ. The algorithm keeps a priority queue Q which initially contains only s. The key for a vertex u in Q is an estimate on the cost of a path from s to γ constrained to go through u. Specifically, let $c(u)$ be the length of the shortest path from s to u and $\hat{c}(u)$ be the length of the shortest *known* path from s to u, then the key associated with u is $\tilde{f}(u) \equiv \hat{c}(u) + \hat{h}_u$. The algorithm repeatedly removes the vertex t from Q with the smallest $\tilde{f}$ value and adds each of its adjacent vertices u to Q if extending a path from s to u through t improves on the shortest previously known path from s to u. This process is known as *expanding* t. A^* continues to expand the top vertex of Q until γ reaches the top of Q. Pseudocode for the A^* algorithm is shown in table 4.3. This pseudocode only computes the cost of the shortest path, but the path itself can easily be computed by keeping track of the neighbor which was responsible for the last update of $\hat{c}(u)$ for each vertex u.

The termination condition in table 4.3 assumes that the lower bounds satisfy the *consistency condition* which requires that for any two vertices u, v, $\hat{h}_u - \hat{h}_v$ is no more than the shortest path distance from u to v. If this condition holds, then when the algorithm expands a vertex t the shortest path to t will be known to the algorithm, i.e. $\hat{c}(t) = c(t)$ and $\hat{f}(t) = \tilde{f}(t)$, where $\hat{f}(t) \equiv c(t) + \hat{h}_u$. The proof is given in Nilsson [10] but the intuition comes from the observation that on any path from s to γ the $\hat{f}$ values never increase. Another important consequence of the consistency condition is that when it holds the A^* algorithm never expands vertices u such that $\hat{f}(u) > c(\gamma)$. See Cormen *et al*[3] and Russell

```
Q ⟵ s
ĉ(s) ⟵ 0
∀u ≠ s, ĉ(u) ⟵ ∞,
until termination
    t ⟵ vertex in Q with smallest f̃ value
    if t = γ then terminate with success.
    else
        Expand(t)


Subroutine Expand(t)
    foreach vertex u adjacent to t
        if ĉ(t) + w(t → u) < ĉ(u) then
            ĉ(u) ⟵ ĉ(t) + w(t → u)
            Add u to Q
    remove t from Q
```

Table 4.3: Pseudocode for the A^* algorithm (assuming the consistency condition holds). $w(t \rightarrow u)$ denotes the weight of the edge from t to u.

& Norvig[11] for details on efficient implementation and other issues regarding general applications of Dijkstra's algorithm and A^*.

How is A^* used for multiple alignment? The multiple alignment dynamic programming problem can be expressed as a shortest paths problem over a graph whose vertices correspond to entries in the dynamic programming table and whose edges and edge weights correspond to the possible transitions between neighboring vertices and their costs, respectively. Meyer [7] gives a nice description of this graph for the alignment of two sequences, which he refers to as the edit graph. Araki *et al.* [1] used the A^* algorithm on an edit graph to speed up the alignment of two sequences; they used completely different bounds than the ones used for multiple sequence alignment but they do show how one can add suitable constants to any cost matrix to ensure that all edges in the edit graph have non-negative weight, without changing the relative order of costs for different alignments. In the case of aligning three sequences, a vertex is associated with one character from each of the three sequences (except for vertices which lie on boundaries), and a path up to that vertex represents an alignment of the prefixes of each of the three strings which end with the respective character associated with the vertex. Each vertex (again excluding boundary vertices) has an in-degree and out-degree of seven. The seven incoming edges represent

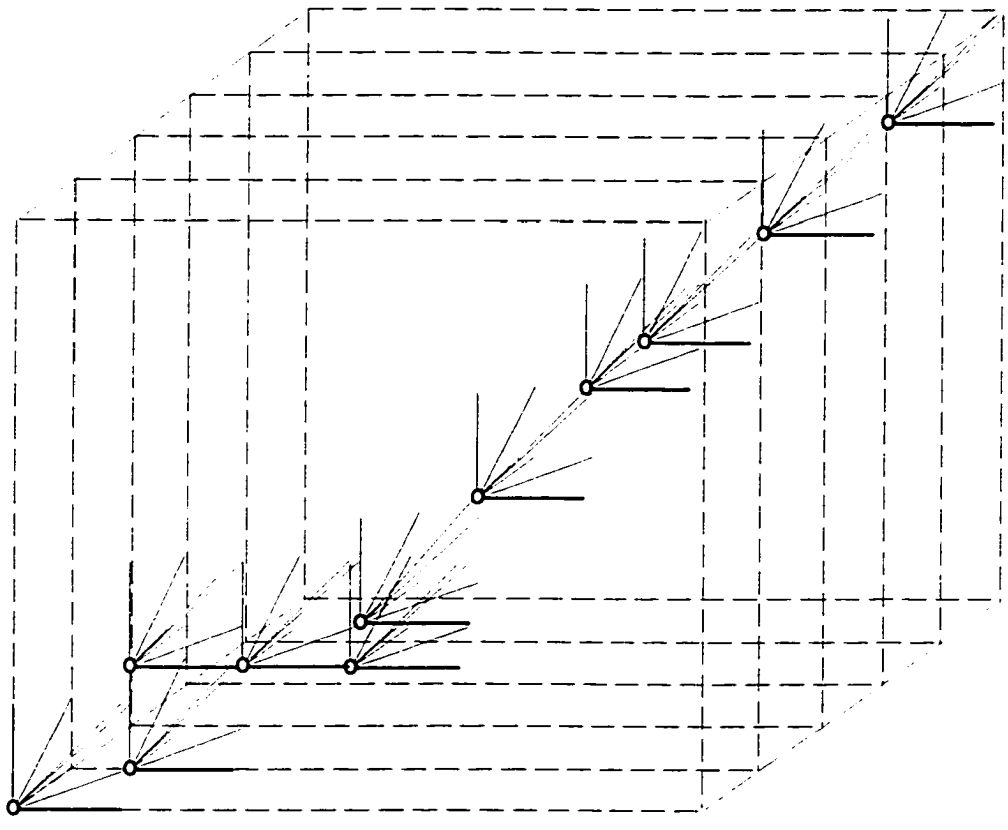


Figure 4.1: A schematic drawing of the vertices which might be explored using A^* with an edit graph of three sequences is shown. The start vertex is in the lower left, while the goal vertex is at the upper right

the seven possible ways to fill three positions with either characters or indels, where filling all three of the available positions with indels is disallowed. If the lower cost bounds are effective only a small portion of the edit graph will be explored. A schematic of how A^* might explore the edit graph of three sequences is shown in figure 4.1.

4.2.4 A^* versus Kececioglu's branch and bound algorithm

The branch and bound algorithm described by Kececioglu [6], [5] is almost identical to the A^* algorithm described in the previous section, and indeed shares many of the advantages of A^*. However there is a crucial difference between the two. The difference is in the way that the priority queue and the lower bounds on finishing an alignment are used. For Kececioglu's algorithm the key for the priority queue is just the distance from the start vertex. The lower bound is only used when expanding a vertex. When a vertex is

expanded any newly reached adjacent vertices u are added to the queue if and only if

$$\hat{f}(u) \equiv c(u) + \hat{h}_u \leq U.$$

where U is an upper bound on the cost of the optimal path. As stated above, the A^* algorithm only expands vertices for which $\hat{f}(u) \leq c(\gamma)$. Thus the two algorithms should have roughly the same efficiency if the upper bound is tight, i.e. $U = c(\gamma)$. If the upper bound is loose however, the A^* algorithm will be expand fewer vertices. This is illustrated by the extreme case in which the lower bounds are perfectly tight but U is set to infinity. In that case the A^* algorithm only expands vertices on an optimal path but Kececioglu's algorithm degenerates to Dijsktra's shortest path algorithm, completely unable to use the lower bounds to advantage.

4.3 A^* Dominates Carrillo & Lipman's Algorithm

In this section we show that A^* dominates Carrillo & Lipman's preprocessing algorithm in the sense that it expands no more vertices than the number of dynamic programming entries which must be computed by Carrillo & Lipman's algorithm. To do this we prove two things: first that A^* never expands a vertex twice, and second that A^* never expands a vertex which corresponds to a dynamic programming table entry that Carrillo & Lipman's algorithm could eliminate.

4.3.1 A^* Never Expands a Vertex Twice

To show that the A^* algorithm described here never expands a vertex twice it suffices to show that the lower bounds satisfy the consistency condition [10]. To do this we must prove that the difference in the lower bound between two vertices is never greater than the shortest path distance between those two vertices. More formally, let the first vertex in question be the vertex which represents aligning the prefixes ending in the qth, rth, and sth characters of the sequences I, J, and K respectively. Likewise let the second vertex in question represent the alignment of the prefixes ending in the uth, vth, and wth characters of the three sequences; where $q \leq u$, $r \leq v$, and $s \leq w$, i.e. where in the edit graph corresponding to aligning I, J, and K the second vertex is reachable from the first. We must prove that $\hat{h}_{qrs} - \hat{h}_{uvw} \leq c(\alpha^*[\rho])$, where ρ denotes the three substrings containing the $u-q$ characters of I starting with the $q+1$th character, and the analogous characters of

J and K, i.e. ρ replaces the notation $(i_q, i_u), (j_r, j_v), (k_s, k_w)$. Let σ_{qrs} be defined as before and defining σ_{uvw} analogously we make the following observation:

$$\begin{aligned} c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) &\leq c_{ij}(\alpha^*_{ij}[\rho]) + c_{ij}(\alpha^*_{ij}[\sigma_{uvw}]) \\ c_{ik}(\alpha^*_{ik}[\sigma_{qrs}]) &\leq c_{ik}(\alpha^*_{ik}[\rho]) + c_{ik}(\alpha^*_{ik}[\sigma_{uvw}]) \\ c_{jk}(\alpha^*_{jk}[\sigma_{qrs}]) &\leq c_{jk}(\alpha^*_{jk}[\rho]) + c_{jk}(\alpha^*_{jk}[\sigma_{uvw}]). \end{aligned}$$

The equalities hold when an optimal alignment of the suffixes σ_{qrs} ends in an alignment of the suffixes σ_{uvw}. In that case $\alpha^*_{ij}[\sigma_{uvw}]$ represents the least costly way to finish the alignment.

However the left hand sides of these inequalities sum to $\hat{h}_{qrs}$, while the second terms of the right hand sides sum to $\hat{h}_{uvw}$. Also from our earlier observations we know that the sum of the cost of independently aligning all possible pairs of three substrings gives a lower bound on the cost of aligning the three substrings. Thus the sum of the first terms of the right sides of the inequalities is a lower bound for $c(\alpha^*[\rho])$. This gives us: $\hat{h}_{qrs} \leq c(\alpha^*[\rho]) + \hat{h}_{uvw}$, finishing the proof.

4.3.2 A^* Expands Fewer Vertices than Carrillo & Lipman

In this section we will show that Carrillo & Lipman's condition for eliminating a cell in the dynamic programming table also implies that the A^* algorithm will never expand the corresponding vertex in the edit graph. Before we begin a formal proof we would like to give some intuition as to why A^* should expand fewer vertices than Carrillo & Lipman's algorithm calculates. The bounds used by the two algorithms are similar but there are three kinds of information that A^* gains by "deferring" the decision to eliminate a vertex instead of eliminating during preprocessing. First, the cost of a good feasible alignment is not needed as the A^* alignment will know the cost of the optimal alignment when it terminates. Second, when expanding an intermediate vertex the A^* alignment knows the exact cost of aligning the prefixes associated with that vertex and needs to use a lower bound estimate only for the cost of aligning the corresponding suffixes. Third, A^* may eliminate more vertices because it compares the sum of the deviations for all pairs of sequences from their lower bound estimate to δ rather than just looking at one pair at a time as the algorithm described by Carrillo & Lipman does. Note that in general A^* will eliminate more vertices that Carrillo & Lipman even if the upper bound U is perfectly tight.

More formally, recall from section 4.2.3 that Carrillo & Lipman's condition for the elimination of the dynamic programming table cell corresponding to the characters i_q, j_r and k_s may be written as:

$$c_{ij}(\alpha^*_{ij}[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) > c_{ij}(\alpha^*_{ij}[I,J,K]) + \delta. \tag{4.1}$$

Similarly the cell can be eliminated if the analogous condition holds when (I,J) is replaced by (I,K) or (J,K). To facilitate the discussion however, we will only consider one particular cell, note that we may name the strings so that if the cell is to be eliminated the above condition written with (I,J) must hold.

Now we show that Carrillo & Lipman's condition implies that the A^* algorithm will never expand the corresponding vertex. In section 4.2.3 we stated without proof that a vertex w will not be expanded if $\hat{f}(w) > c(\gamma)$. Again considering the vertex corresponding to the characters i_q, j_r, k_s and substituting the specific costs and lower bounds used for multiple sequence alignment we obtain:

$$c(\alpha^*[\pi_{qrs}]) + \hat{h}_{qrs} \equiv c(\alpha^*[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) + c_{ik}(\alpha^*_{ik}[\sigma_{qrs}]) + c_{jk}(\alpha^*_{jk}[\sigma_{qrs}]) > c(\alpha^*) \tag{4.2}$$

Theorem 4.3.1 (Horton & Lawler) *The A^* algorithm will never expand a vertex which would be eliminated by Carrillo & Lipman's preprocessing step. Or equivalently, equation 4.1 implies equation 4.2.*

Proof: from equation 4.1, expanding δ and cancelling terms gives:

$$c_{ij}(\alpha^*_{ij}[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) > U - c_{ik}(\alpha^*_{ik}[I,J,K]) - c_{jk}(\alpha^*_{jk}[I,J,K]),$$

Adding $c_{ik}(\alpha^*_{ik}[\pi_{qrs}]) + c_{jk}(\alpha^*_{jk}[\pi_{qrs}])$ to both sides gives:

$$\begin{aligned} c_{ij}(\alpha^*_{ij}[\pi_{qrs}]) + c_{ik}(\alpha^*_{ik}[\pi_{qrs}]) + c_{jk}(\alpha^*_{jk}[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) > \quad & U + c_{ik}(\alpha^*_{ik}[\pi_{qrs}]) + c_{jk}(\alpha^*_{jk}[\pi_{qrs}]) \\ & - c_{ik}(\alpha^*_{ik}[I,J,K]) - c_{jk}(\alpha^*_{jk}[I,J,K]). \end{aligned}$$

Again however, we know that the cost of aligning the prefixes independently gives a lower bound on the cost to align them, i.e. $c(\alpha^*[\pi_{qrs}]) \geq c_{ij}(\alpha^*_{ij}[\pi_{qrs}]) + c_{ik}(\alpha^*_{ik}[\pi_{qrs}]) + c_{jk}(\alpha^*_{jk}[\pi_{qrs}])$. This gives us

$$c(\alpha^*[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) > U + c_{ik}(\alpha^*_{ik}[\pi_{qrs}]) + c_{jk}(\alpha^*_{jk}[\pi_{qrs}]) - c_{ik}(\alpha^*_{ik}[I,J,K]) - c_{jk}(\alpha^*_{jk}[I,J,K]) \tag{4.3}$$

Here we observe that

$$c_{ik}(\alpha^*_{ik}[I,J,K]) \leq c_{ik}(\alpha^*_{ik}[\pi_{qrs}]) + c_{ik}(\alpha^*_{ik}[\sigma_{qrs}])$$

or equivalently,

$$c_{ik}(\alpha^*_{ik}[\pi_{qrs}]) - c_{ik}(\alpha^*_{ik}[I,J,K]) \geq -c_{ik}(\alpha^*_{ik}[\sigma_{qrs}]).$$

This is clear because the optimal prefix and suffix alignments can be concatenated to produce a feasible alignment of the strings I and K. Combining this inequality and the analogous inequality for (J,K) with equation 4.3 we obtain:

$$c(\alpha^*[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) > U - c_{ik}(\alpha^*_{ik}[\sigma_{qrs}]) - c_{jk}(\alpha^*_{jk}[\sigma_{qrs}]).$$

$$c(\alpha^*[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) + c_{ik}(\alpha^*_{ik}[\sigma_{qrs}]) + c_{jk}(\alpha^*_{jk}[\sigma_{qrs}]) > U.$$

We have by definition $U \geq c(\alpha^*)$. This gives

$$c(\alpha^*[\pi_{qrs}]) + c_{ij}(\alpha^*_{ij}[\sigma_{qrs}]) + c_{ik}(\alpha^*_{ik}[\sigma_{qrs}]) + c_{jk}(\alpha^*_{jk}[\sigma_{qrs}]) > c(\alpha^*),$$

which completes the proof.

4.4 Conclusion

We have shown that for a reasonable measure of work, namely the number of vertices expanded, the A^* algorithm is in general more efficient that the algorithm presented by Carrillo & Lipman [2]. A^* also has a smaller but provable advantage over the branch and bound algorithm of Kececioglu [6] [5]. Carrillo & Lipman's algorithm does not require a priority queue and thus it may still be faster for some problems. However Kececioglu reports that by implementing the priority queue as a bucketed heap for his algorithm, the overhead of the queue is more than compensated for by the reduced number of vertices that need to be visited. This empirical result should apply directly to the A^* algorithm described here as well.

Bibliography

[1] S. Araki, M. Goshima, S. Mori, H. Nakashima, S. Tomita, Y. Akiyama, and M. Kanehisa. Application of parallelized dp and a^* algorithm to multiple sequence alignment. In *Proceedings of the Genome Informatics Workshop IV*, pages 94–102, Tōkyō, 1993. Universal Academy Press.

[2] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math*, 49:1073–1082, 1988.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[4] M. O. Dayhoff, editor. *Atlas of Protein Sequence and Structure*, volume 5. Natl. Biomed. Res. Fournd., Washington, DC, 1978.

[5] Sandeep K. Gupta, John D. Kececioglu, and Alejandro Shäffer. Making the shortest-paths approach to sum-of-pairs multiple sequence alignment more space efficient in practice. In *Proceedings of Combinatorial Pattern Matching*, 1995.

[6] John Kececioglu. A branch-and-bound algorithm for minimum sum of pairs multiple sequence alignment. May 1992.

[7] Eugene Meyer. An overview of sequence comparison algorithms in molecular biology. Technical Report TR 91-29, Computer Science, University of Arizona, December 1991.

[8] M. Murata, J. S. Richardson, and J. L. Sussman. Simultaneous comparison of three protein sequences. *Proc. Natl. Acad. Sci., USA*, 82:3073–3077, 1985.

[9] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:444–453, 1970.

[10] N. J. Nilsson. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[11] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall, 1995.

Part II

Application of Machine Learning

Chapter 5

Learning to Classify Protein Sequences by their Cellular Localization Sites

Abstract

This Chapter describes the first application of machine learning to the problem of classifying protein sequences by their cellular localization sites. The results of three studies are consolidated and presented: [10], [11], and [9] respectively. The first two studies used expert defined features while the third study attempted to discover new features. The first study developed a probabilistic model which is easy to interpret and classifies reasonably well. The second study compared the accuracy of the probabilistic model to the standard k nearest neighbor (kNN), binary decision tree, and naïve Bayes classifiers. The result being that kNN performed best with an accuracy of approximately 60% for 10 yeast classes and 86% for 8 *E.coli* classes. That study also compared the effectiveness of using kNN with distances defined by sequence similarity with the results being highly favorable to the problem specific expert defined features. Lastly, the third study describes an attempt to discover interesting substring features, using the suffix tree data structure for efficiently calculating the correlation between substrings in the protein sequences and their localization sites. As implemented this approach did not yield higher classification accuracy but was anecdotally successful in automatically finding a meaningful biological feature that was unknown to us at the time we created the program.

Keywords: Protein Localization, k Nearest Neighbor Classifier, Classification, Feature Discovery, Yeast, *E.coli*

5.1 Introduction

In order to function properly, proteins must be transported to various localization sites within the cell. Conversely, the cellular localization site of a protein affects its potential functionality as well as its accessibility to drug treatments. Fortunately the information needed for correct localization is generally found in the protein sequence itself. We begin this chapter with a description of the process of protein localization. The description provided is a simplified one which is only intended to motivate and clarify the work and discussion presented in this chapter. We then describe the classes (localization sites), features, and datasets (protein sequences) used throughout the chapter.

Our motivation for applying machine learning to this problem came both from the success of an earlier system and its drawbacks. The first integrated system for predicting the localization sites of proteins from their amino acid sequences was a production type expert system [18], [19]. That system is still useful and popular but it requires hand tuning of certainty factors for each rule to obtain maximal performance. This makes it very time

consuming to update or adapt to new organisms, as well as making cross-validation studies impossible. Thus we were in the favorable position of knowing that even if we could only roughly match the accuracy of the expert system with a program that learned how to classify automatically, we would be able to make a significant contribution.

We succeeded in doing this with the development of a structured probabilistic model [10]. The structure of the model reflects the beliefs of a human expert but, once the structure is defined, parameters of the model are estimated from the data without any need for hand tuning. The input for the model is a vector of real valued features calculated directly from the protein sequence and the output is a probability vector giving the estimated probability that the protein belongs to any class. The model itself can either be viewed as a probabilistic analog to decision trees or as a restricted form of Bayesian network. We also report the results of comparing three different schemes for dealing with continuous variables.

The next section reports the results of comparing different classifiers on our protein localization dataset [11]. Our probabilistic model had two goals. The main goal was to obtain a high classification accuracy, but there was also a secondary goal of creating a model that could be interpreted in terms of what is known about the process of protein localization. However for many practical uses, the classification accuracy is all that matters. This situation motivated us to compare three standard classification algorithms, kNN, Naïve Bayes, and the binary decision tree to our probabilistic model. The difference in accuracies between the algorithms were not dramatic but the kNN algorithm did outperform the others on both the yeast and the *E.coli* dataset. In particular, using a paired-differences t test, the accuracy of kNN on the yeast dataset was significantly higher than the other classifiers. Another important comparison reported in this section is the comparison between using the expert identified features with kNN versus simply using sequence similarity with kNN. The results show that sequence similarity is not as effective as the expert identified features.

The last study in this chapter describes a method for discovering interesting substring features. The ultimate aim being to automatically find features which could complement the expert identified features. The method works directly with the protein sequences, instead of using the expert identified features. A common approach to finding substring features is to examine the frequency distribution of substrings of a fixed length. However the restriction of substrings to a fixed length is unnecessary, since by using the suffix tree data structure the frequencies of all substrings can be examined (indirectly for most substrings)

efficiently. We uses the suffix tree data structure combined with a χ^2 statistic test to identify substrings which correlate closely to specific classes. Finally those substrings were used as input to induce a decision tree. The method was able to find features with enough power to classify significantly better than the baseline majority class classifier but not enough power to classify as well as sequence similarity, let alone the expert identified features. However the method did consistently identify a substring which has known biological significance, (but is not at all obvious when examining the data manually).

We conclude this chapter with a discussion of possible extensions to this work. Both in terms of protein localization, and biosequence classification problems in general.

5.2 Protein Localization

5.2.1 Membranes and Compartments

Part of the structure of biological cells is due to the presence of membranes, which are normally impermeable to large molecules such as proteins. These membranes separate the inside of the cell from the exterior environment as well as subdividing most cells into different compartments. The structure of a Gram-negative bacteria such as *E.coli* is shown in figure 5.1. Each membrane and each compartment has its own functions and requires its own repertoire of proteins.

5.2.2 Localization in *E.coli*

In bacteria all proteins are made in the cytoplasm. Proteins which perform their functions in membranes or in other compartments, such as the periplasmic space, must be recognized and transported across (or into) the appropriate membranes. The presence of a length ≈ 10 substring of hydrophobic amino acids known as a *signal sequence* allows proteins to enter the inner membrane. When the signal sequence is found near the N-terminus of the protein it is often cleaved after the protein enters the inner membrane. The mechanism for the localization of lipoproteins is somewhat different as the lipid portion, rather than the protein portion, often serves to anchor the protein to the appropriate membrane. It has been empirically observed that lipoproteins with a positively charged residue at position 2 or 3 of the mature protein are usually localized to the inner membrane while other lipoproteins are usually localized to the outer membrane.

5.2.3 Localization in Yeast

Eukaryotic cells, such as yeast, have many more compartments than bacteria, as shown in figure 5.2. In eukaryotic cells most proteins are coded in DNA found in the nucleus and synthesized in the cytoplasm. Mitochondria also have their own DNA and the proteins that they code for are made inside the mitochondria (and stay within the mitochondria). However, the majority of mitochondrial proteins are imported from the cytoplasm. This work only considers nuclear encoded proteins so we will simplify the discussion by hereafter ignoring the existence of proteins made in the mitochondria.

The Secretory Pathway.

Many proteins in eukaryotic cells are localized as part of the *secretory pathway.* Proteins whose final destination is the lumen of the Endoplasmic Reticulum (ER), the Golgi body, the membranes of either of those organelles, the plasma membrane, the cell wall, or outside of the cell itself all use the secretory pathway. In the first step of the secretory pathway a nascent protein, i.e. one that is in the process of being synthesized, is passed into the membrane of the ER. Membrane proteins do not pass all the way through but are retained in the membrane. A eukaryotic signal sequence, a length 6-12 substring of hydrophobic amino acids similar both in function and composition to the bacterial signal sequence, is generally found within the first 25 amino acids of the protein. The constraints on the signal sequence are loose enough that many variations are possible. This variation is illustrated in the example signal sequences found in table 5.1. Most signal sequences are cleaved by enzymes in the ER to form the mature protein. However some membrane proteins have internal signal sequences, i.e. signal sequences which are further from the N-terminus, which are not cleaved. Once localized to the ER, the secretory pathway defines a kind of "default" pathway for proteins. Unless specific retention signals are present, membrane proteins flow from the ER to the Golgi body membrane and then on to either the plasma membrane or the membrane of the vacuole. Likewise proteins transported to the ER lumen tend to flow to the lumen of the Golgi body and then out of the cell or into the inside of a vacuole. Some retention signals have been fairly well characterized, for example although they are temporarily transported to the lumen of the Golgi body, proteins with a C-terminal suffix of HDEL (Histidine-Aspartic Acid-Glutamic Acid-Leucine) are specifically returned to the ER (HDEL is the suffix in yeast but other eukaryotic cells generally use a

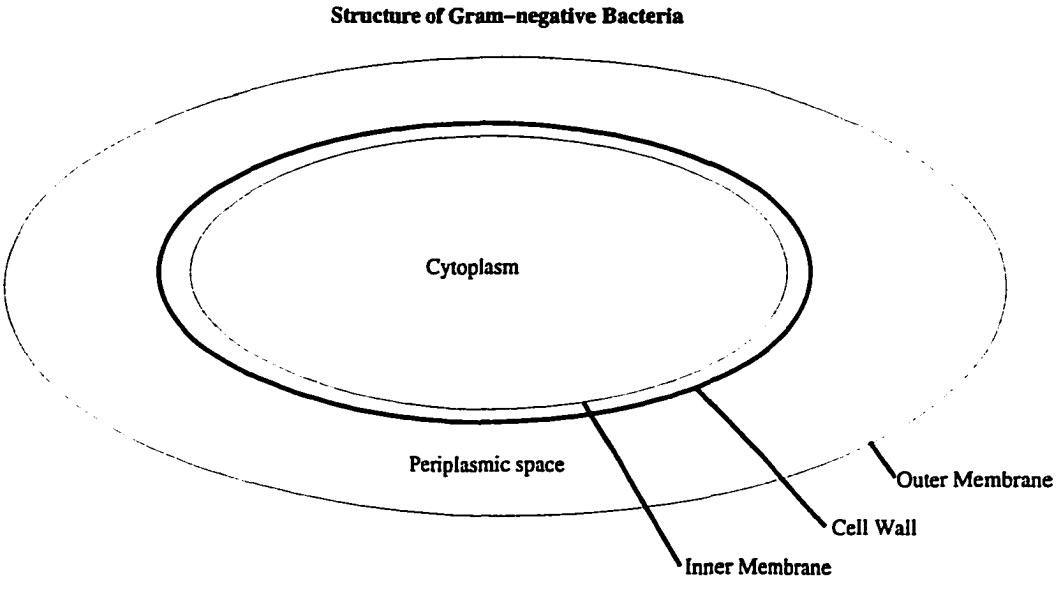


Figure 5.1: A schematic depiction of the membranes and compartments of Gram-negative Bacteria is shown.

Lysine in place of the Histidine, yielding KDEL). Other signals are partially characterized, for example a single membrane spanning α helix is known to cause some proteins to be retained in membrane of the Golgi body. Still other targeting signals are not well understood in terms of their sequence requirements.

Protein	N-terminal prefix
Preproalbumin	MKWVT**FLLLLFISGSAFSR**
Pre-IgG light chain	MDMRAPAQ**IFGFLLLLFPGTRCD**
Prelysozyme	MR**SLLILVLCFL**PLAALGK
Preprolactin	MNSQVSARKAGT**LLLLMMSNLL**
VSV glycoprotein	MK**CLLTLAFLFI**HVNCK
Rat proinsulin-1	MA**LWMRFLPLLALLVLWEPKPAQAF**
Acetylcholine receptor γ subunit precursor	**MVLTLLLIICLALEVRSE**

Table 5.1: Adapted from [4]. The first (i.e. N-terminal) few amino acids of several proteins which function as eukaryotic signal sequences are shown. Runs of hydrophobic acids are in boldface.

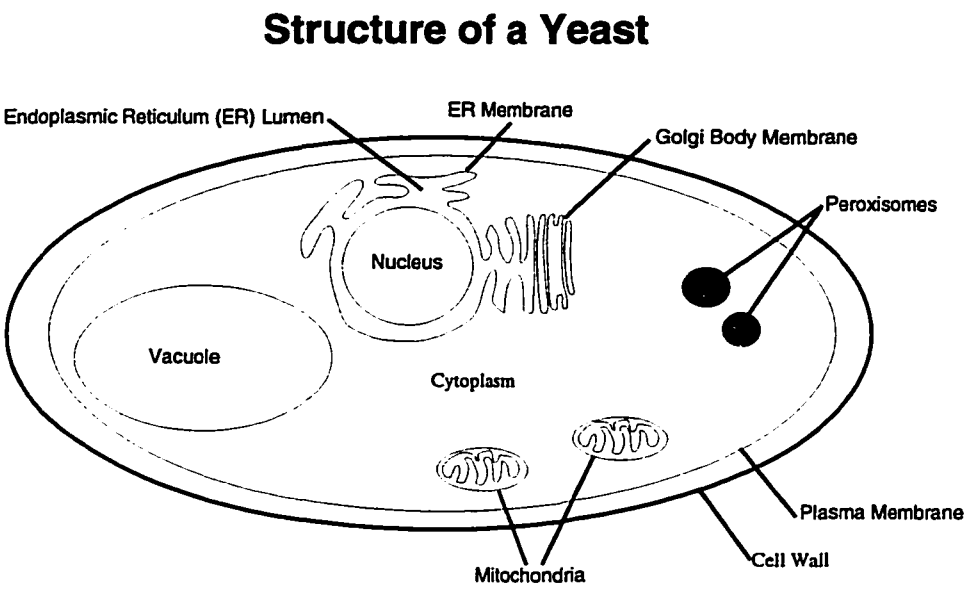


Figure 5.2: A schematic depiction of the membranes and compartments of a yeast cell is shown.

Other Pathways.

Proteins localized to the mitochondria, peroxisomes, and the nucleus do not use the secretory pathway. Instead their proteins are directly imported from the cytosol. Like the signal signals in the secretory pathway, partially characterized sequence motifs are known to cause proteins to be imported into mitochondria and peroxisomes. Many (but not all) proteins imported into the mitochondria have a targeting signal near their N-terminal. The signal is a loosely defined pattern of 3-5 nonconsecutive arginine or lysine residues typically surrounded by serine or threonine residues and excluding any acidic amino acid residues. Proteins which are imported into the peroxisomes often have the substring "serine-lysine-leucine" near their C-terminus.

Import to the Nucleus.

In contrast to other membranes, the import of proteins across the nuclear membranes occurs through relatively large *nuclear pores*. These pores make it possible for proteins to pass through the nuclear membrane in a folded state, in some cases even while complexed with another protein. However, it is still not possible for medium or large sized proteins to passively diffuse through the pores, and like other organelles, a signal is recog-

nized in proteins before they are carried through the pores. In several proteins a length 5 substring of basic residues or two shorter substrings of basic residues separated by 10 residues has been experimentally shown to be required for nuclear localization. This *nuclear localization signal* must occur on the surface of the folded protein but, except for that constraint, may occur anywhere in the amino acid sequence. Only one protein in a protein complex needs to have a nuclear localization signal for the complex to be imported into the nucleus.

5.3 Datasets

5.3.1 Sequences

We used two datasets: an *E.coli* dataset with 336 proteins sequences labeled according to eight classes (localization sites) and a yeast (specifically *Saccharomyces cerevisiae*) dataset with 1484 sequences labeled according to ten classes. In some of our experiments a few of the yeast sequences were discarded. We defined two additional datasets: a non-identical dataset of 1462 sequences which contained no identical sequences, and a "non-redundant" dataset of 1386 sequences which contained no pair of sequences with more than 50% identity (after alignment). The class information was taken from the annotations of SWISS-PROT[1] for the *E.coli* sequences and from the YPD[27] for the yeast sequences.

5.3.2 *E.coli* classes and features

Proteins from *E.coli* were classified into eight classes shown with their frequencies in table 5.2. Seven expert identified features were used and are summarized in table 5.3.

5.3.3 Yeast classes and features

Proteins from yeast were classified into 10 classes shown with their frequencies in the non-identical dataset in table 5.4. Eight expert identified feature were used, three of which (*alm*, *gvh*, and *mcg*) were the also used for the *E.coli* classification. A brief description of the other five features is shown in table 5.5

Classes for the *E.coli* Dataset

Class	Abbr.	Number
Cytoplasm	cp	143
Inner membrane, no signal sequence	im	77
Periplasm	pp	52
Inner membrane, uncleavable signal sequence	imU	35
Outer membrane non-lipoprotein	om	20
Outer membrane lipoprotein	omL	5
Inner membrane lipoprotein	imL	2
Inner membrane, cleavable signal sequence	imS	2

Table 5.2: The names, abbreviations and number of occurrences of each class for the *E.coli* dataset are shown.

5.3.4 Dataset Issues

An inspection of the class definitions makes it clear that some classes are actually localized to the same part of the cell, for example *im*, *imU*, *imL*, and *imS* are all in fact localized to the inner membrane. The rationale behind dividing the inner membrane proteins into several classes is that the different classes are localized by different mechanisms. This is useful if your goal is to model the process but not necessarily useful for increasing prediction accuracy. This issue is explored further in the discussion section of study 2. In other cases several distinct localization sites have been lumped into one class. For example although all mitochondrial proteins are considered here as class "MIT", it is known that mechanisms exist to further localize those proteins to the inner or outer membrane of the mitochondria or the matrix or intermembrane space of the mitochondria. Thus the choice of classes is somewhat arbitrary but in general reflects what is known about the localization process as well as the availability of sufficient data for the localization sites. The datasets used are available from the UCI Machine Learning Data Repository [17] and are described in more detail in [18], and [19].

Features for the *E.coli* Dataset

Feature	name
Modification of McGeoch's signal sequence detection parameter[16]	mcg
The presence or absence of the consensus sequence[26] for Signal Peptidase II	lip
The output of a weight matrix method for detecting cleavable signal sequences by von Heijne's method[25]	gvh
The output of the ALOM program[12] for identifying membrane spanning regions on the whole sequence	alm1
The output of the ALOM program on the sequence excluding the region predicted to be a cleavable signal sequence by von Heijne's method[25]	alm2
The presence of charge on the N-terminus of predicted mature lipoproteins	chg
The result of discriminant analysis on the amino acid content of outer membrane and periplasmic proteins	aac

Table 5.3: A description of the *E.coli* features and their names are shown.

Classes for the Yeast Dataset

Class	Abbr.	Number
Cytoplasm	CYT	444
Nucleus	NUC	426
Mitochondria	MIT	244
Membrane protein, no N-terminal signal	ME3	163
Membrane protein, uncleaved signal	ME2	51
Membrane protein, cleaved signal	ME1	44
extracellular	EXC	35
Vacuole	VAC	30
Peroxisome	POX	20
Endoplasmic Reticulum	ERL	5

Table 5.4: The names, abbreviations and number of occurrences of each class for the yeast dataset are shown.

Features for the Yeast Dataset

Feature	name
The presence or absence of HDEL suffix	erl
The result of discriminant analysis of the amino acid content of vacuolar proteins vs. extracellular proteins	vac
The result of discriminant analysis of the amino acid content of the 20-residue N-terminal region of mitochondrial and non-mitochondrial proteins	mit
The presence or absence of nuclear localization consensus patterns combined with a term reflecting the frequency of basic residues	nuc
The presence or absence of a short sequence motif combined with the result of discriminant analysis of the amino acid composition of the protein sequence	pox

Table 5.5: A description of the yeast features and their names are shown. Features found in the table for *E.coli* were omitted.

5.4 Probabilistic Model

5.4.1 Model Definition

Proteins are synthesized at a common location, namely the cytosol, and then cross a series of membranes, one by one, before reaching their appropriate destination. Thus the path a protein takes can be visualized as a rooted tree where the branches represent binary events such as whether or not the protein enters a membrane or whether or not the protein exits a membrane it has entered. With this image in mind, we defined a simple model for the probabilistic classification of objects. The model consists of a rooted binary tree with a feature variable associated with each non-leaf node of the tree, as in figure 5.3a. The "classification variables", or nodes of the tree, are boolean variables which represent membership in some set of classes. The leaves of the tree represent the possible classes into which an object can be classified. A non-leaf node n represents the union of all the classes which belong to leaves that are descendants of n (in this section we will refer to that set of classes as the class of node n). When performing inference, each node has a probability associated with it, the probability of n being true represents the probability that an object belongs to n's class. Since the children of a node represent a partitioning of the node's class, it follows that the probability that a node is true must equal the sum of the probabilities that its children are true. For example in figure 5.3, $Pr[C_1] = Pr[C_{10}] + Pr[C_{11}]$. Each non-leaf node n has a feature variable F_n and a conditional probability table (or function) associated with it. The influence of the features of an object on whether it should be

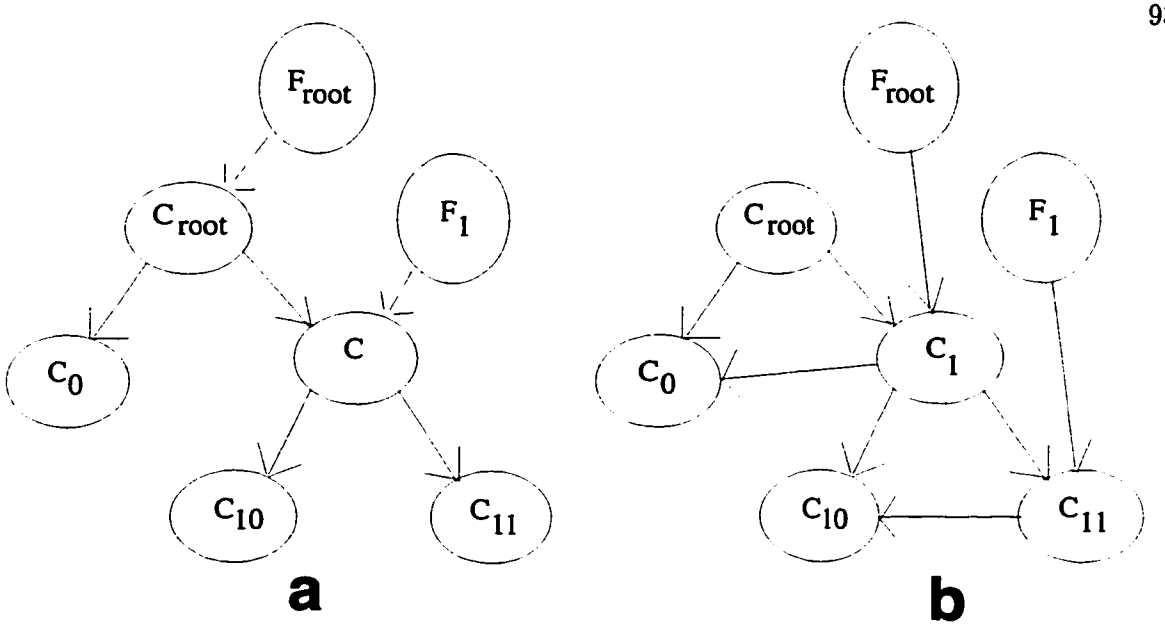


Figure 5.3: (a.) An example of a classification tree with its feature variables. The dotted edges represent feature variables associated with classification variables. (b.) One possible equivalent Bayesian network; where F_1 and F_{root} are clamped variables.

classified as being a left descendant of n versus being classified as a right descendant of n is assumed to be completely summarized by F_n. For example in figure 5.3 this would imply that $Pr[C_{11}|C_1, F_1] = Pr[C_{11}|C_1, F_1, F_{root}]$. This conditional independence allows us to calculate the probability of each node given a set a values for the feature variables, and the appropriate conditional probability tables, with one traversal of the tree. The traversal starts with the root which always has a probability of 1. Although we did not originally conceptualize this model as a family of Bayesian networks, it can be expressed as such. For readers who prefer to think in terms of Bayesian networks, the translation of our example to a Bayesian network is shown in figure 5.3b.

5.4.2 Classification Trees for *E.coli* and Yeast

With the help of a human expert, we built one classification tree each for *E.coli* and yeast shown in figures 5.4 and 5.5. The structure of these classification trees roughly reflects the localization pathways. For example the left subtree of both classification trees reflects proteins with signal sequences. Furthermore the left subtree of the yeast classification tree (figure 5.5) represents the secretory pathway with downstream locations corresponding to nodes of greater depth in the tree. Of course some paths in the classification trees do not

reflect biological pathways. For example, the order of the nodes in the right subtree of the yeast classification tree is essentially arbitrary from a biological point of view, since proteins are transported directly from the cytosol to the mitochondria, peroxisomes, and the nucleus.

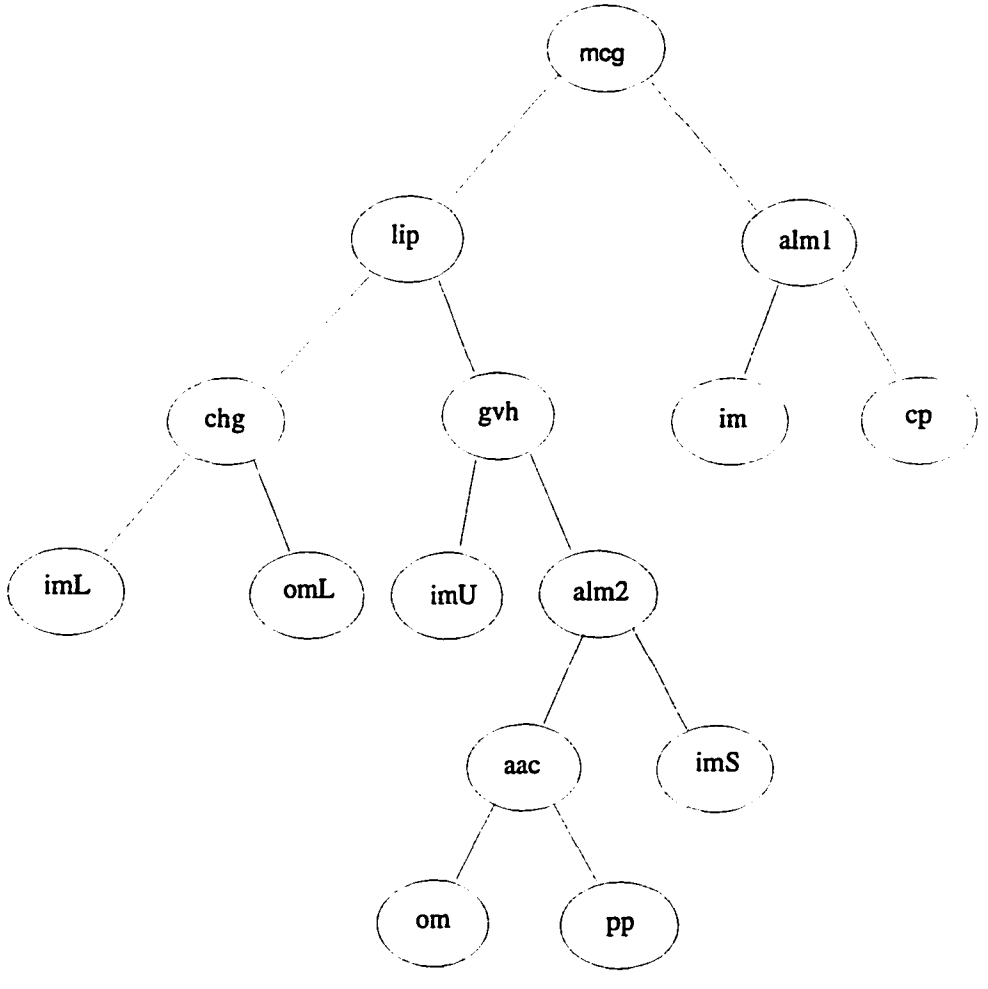


Figure 5.4: The classification tree used for *E.coli* protein localization is shown. The leaf nodes are labeled with the class that they represent, while the other nodes are labeled with their feature variable. All the edges shown are directed downward and the edges connecting feature variable to their classification variables are not shown explicitly. The labels are abbreviations (see tables 5.2 and 5.3).

5.4.3 Conditional Probabilities from Continuous Variables

If the feature variables are discrete variables then the conditional probabilities, e.g. $Pr[C_{11} = 1|C_1 = 1, F_1 = 1]$ may be estimated by counting the frequency of the examples in the training data for which $F_1 = 1$ & $C_{11} = 1$ and dividing by the frequency of examples for which $F_1 = 1$ & $C_1 = 1$. Indeed the features: *lip*, *chg*, and *erl*, were discrete variables and required no further processing. However, the other features in this application were "continuous" variables, i.e. variables with a large or infinite number of possible values. We simplified the problem somewhat by using a linear transformation to normalize the feature values to fall within the range of $[0, 1]$. We then tried three approaches for dealing with the resulting values: uniform binning, Fayyad-Irani binning, and fitting a sigmoid curve.

Uniform Binning

In the first two method we discretized the values by dividing $[0, 1]$ into intervals and treated each interval as a single value. The intervals were chosen such that a roughly equal number data points (i.e. values of the feature variable in question for the sequences in our training data), fell into each interval. Unfortunately, we were not able to derive a well principled criterion for how many intervals the range $[0, 1]$ should be divided into. Instead we somewhat arbitrarily tried making a number of intervals equal to either the log to the base 2 of the number of relevant examples, or the square root of the number of those examples. Here relevant means that the examples belong to the class of the node whose feature value we were discretizing.

5.4.4 Fayyad-Irani binning

The second method we employed is known as Fayyad-Irani binning[5]. This method first chooses a threshold which splits the interval $[0, 1]$ into two bins such that the weighted sum of the entropy of the class frequencies in the training examples in each bin is minimal; with the entropy of each bin being weighted by the fraction of the examples it contains. The algorithm is then recursively applied to each bin until further splitting is deemed unjustifiable. We quit splitting when a χ^2 test failed to show a significant difference between the class frequencies found in a bin versus the frequencies which would result from splitting the bin. Fayyad-Irani binning is a special case of binary decision tree induction which is described in study 2. This method seems less arbitrary than uniform binning but we did

not learn of it until after our first study had been completed.

5.4.5 Sigmoid Conditional Probability Function

The third method we employed does not discretize the values but learns a conditional probability function in the form of the sigmoid function $G(F_i) = \frac{1}{1+e^{(aF_i+b)}}$. More specifically suppose that we want to learn the conditional probability function $Pr[C_1|F_1]$ from figure 5.3. Let F_{1i} denote the value of F_1 for the ith example of the training data. We used gradient descent to choose values for a_1 and b_1 which minimize

$$\sum_i (G(F_{1i}) - C_{11})^2$$

where C_{11} equals one when true, i.e. for examples of class C_{11}, and zero otherwise. The summation is over all the examples of class C_1. We subscripted a and b here to indicate that a separate pair of a and b parameters are learned if a feature variable is used more than once in the tree. This sigmoid function does not have a local minimum and therefore gradient descent is sufficient for learning optimal values for a and b. The reader may observe that this procedure is equivalent to using a feed-forward neural network with just one input node and one output node to learn the mapping from feature variable values to conditional probabilities.

5.5 Study 1: Different Binning Strategies with the Classification Trees

This section reports the results of the first application of machine learning to the protein localization problem[10].

5.5.1 Results of Study 1

We report the results of 10-fold cross-validation tests using our probabilistic model. Tables 5.6 and 5.7 show the overall accuracy the *E.coli* and yeast classification trees with three different binning strategies. For yeast we ran the experiment for both the original dataset and the "non-redundant" dataset. For the sigmoid function, the accuracy for each class is broken down in tables 5.8 and 5.9.

Results of *E.coli* Protein Classification Using 3 Strategies

	sigmoid	log	square root
all data	84.2%	79.8%	82.7%
X-valid	81.1%, 7.7	79.1%, 10.1	80.6%, 7.1

Table 5.6: The accuracy of classifying *E.coli* proteins by three different strategies for learning conditional probabilities of continuous variables is shown. The cross-validation row gives the average accuracy and its standard deviation for each strategy.

Results of Yeast Protein Classification Using 3 Strategies

	sigmoid	log	square root
all data	54.5%	54.2%	56.5%
X-valid	54.9%, 4.9	53.9%, 4.1	54.3%, 4.4
non-red.	54.4%	55.6%	55.9%
X-valid	54.1%, 4.9	53.9%, 5.0	55.0%, 4.2

Table 5.7: The accuracy of classifying yeast proteins by three different strategies for learning conditional probabilities of continuous variables is shown. The third line reports the accuracy for training on all of the non-redundant dataset (described in the text), and the fourth line reports the cross-validation accuracy for that dataset. The cross-validation rows show the average accuracy and its standard deviation for each strategy.

Results of *E.coli* Protein Classification Using Sigmoid Conditional Probability Functions

Examples	Class	High	Top2
77	im	77.9%	89.6%
143	cp	96.5%	100%
2	imL	50.0%	50.0%
5	omL	100%	100%
35	imU	71.4%	91.4%
2	imS	0.0%	0.0%
20	om	65.0%	85.0%
52	pp	78.9%	94.2%

Table 5.8: The accuracy of classification of *E.coli* proteins is displayed for each class when all of the data was used for training. For each class the number of examples in the training data, the percentage of sequences for which the correct class matched the class with the highest computed probability, and the percentage which matched one of the classes with the 2 highest computed probabilities is shown.

Results of Yeast Protein Classification Using Sigmoid Conditional Probability Functions

Examples	Class	High	Top2
44	ME1	63.6%	81.8%
5	ERL	60.0%	80.0%
30	VAC	10.0%	13.3%
35	EXC	45.7%	60.0%
51	ME2	15.7%	52.9%
244	MIT	47.1%	56.6%
429	NUC	35.7%	90.2%
20	POX	0.0%	0.0%
163	ME3	85.3%	92.0%
463	CYT	74.3%	93.1%

Table 5.9: The accuracy of classification of yeast proteins is displayed for each class when all of the data was used for training. For each class the percentage of sequences for which the correct class matched the class with the highest computed probability, and the percentage which matched one of the classes with the 2 highest computed probabilities is shown.

5.5.2 Attempted Extensions to the Probabilistic Model

The probabilistic model is very restrictive in allowing only one feature variable to be used at each node. If the feature variables are perfectly effectively in detecting what they are designed to detect than that restriction is perhaps a desirable one, but with biology we cannot expect that to be the case very often. The assertions of conditional independence implied by the tree structure, for example

$$Pr[C_{11}|C_1, F_1] = Pr[C_{11}|C_1, F_1, F_{root}]$$

can be tested with a χ^2 test. Indeed there are several combinations of nodes and their associated feature variables for which this hypothesis could be statistically rejected. We attempted to use this observation by allowing more than one feature variable to be used for the conditional probability function of a given node. We started with the expert defined classification trees of figures 5.4 and 5.5 and added the feature variable for which conditional independence could be rejected at the highest confidence level until no conditional independence assertions could be rejected with a confidence level of 0.95. We then used a feed-forward neural network with no hidden layer at each node to learn the condition probability functions, for nodes with only one feature variable this reduced to simply using the sigmoid function. Unfortunately the resulting classifier did not do any better in

cross-validation tests than the original classification trees. This was a disappointing result. However it is often difficult to use statistically significant but relatively weak correlations to increase the accuracy of a classifier. It is possible to conceive of other attempts to relax or extend our probabilistic model but those extensions would take us further away from directly representing what is known about the biology. Rather than do that we opted to use several general purpose classification algorithms described in study 2.

5.5.3 Discussion of Study 1

This study showed that machine learning could be successfully applied to predicting protein localization sites. The most common class of protein represents 41% and 32% of the *E.coli* and yeast data respectively. Thus the obtained classification accuracies of 81.1% and 54.9% are dramatically superior than that obtained by simply choosing the most common class. A direct comparison of the classification accuracies of this system and the expert system of Nakai and Kanehisa [18, 19] was impossible because the difficulty of tuning the certainty factors of the rules makes cross-validation with their system infeasible. However the classification accuracy of our probabilistic model appeared roughly comparable to theirs. Indeed the reason for the difficulty of this comparison underscores the utility of using machine learning. Given a classification tree, our program could compute everything it needed from the training data.

We empirically evaluated three strategies for handling continuous variables. Of these three, the sigmoid conditional probability function performed slightly better on the *E.coli* dataset and uniform binning with a square root number of intervals performed slightly better on the yeast dataset. The relatively low performance of the sigmoid function with the larger yeast dataset was consistent with the fact that the sigmoid function has only two free parameters. In fact the sigmoid function is so restrictive that it cannot model many distributions, such as bimodal ones. Thus we were somewhat surprised at the relatively strong performance obtained with the sigmoid function. However when one considers that the sequence features used for our classification are imperfect sensors of essentially binary conditions (e.g. either a signal is cleaved or it is not), then one would expect the probability distribution to basically look like a fuzzy step function. The sigmoid function is well suited for use as a fuzzy step function and, apparently, therefore also well suited for this application. Interestingly, the sigmoid function actually showed a higher accuracy on the cross-validation

test for the complete yeast dataset then when trained on all of the data. While this is surprising, it is not a contradiction because the sigmoid functions were fit to minimize the root mean squared error between the function and the data points rather then to directly minimize the number of misclassified sequences. This example demonstrates the favorable property of being resistant to overfitting but also shows that the limited representational power of the sigmoid curve is completely saturated with a dataset as large as the yeast dataset. Thus although we would highly recommend the sigmoid curve for small datasets, we switched to Fayyad-Irani binning for study 2.

Although we were generally happy with the classification results we were disappointed by the fact that none of the 20 POX (peroxisomal) proteins were predicted correctly. In addition to the pox feature variable used when generating table 5.9, we also tried two other pox feature variables, neither of which enabled the program to correctly predict any of the POX proteins. In the discussion of Horton & Nakai[10] that failure was mainly attributed to the under-representation of POX examples in the dataset. However, as will be seen in study 2, another classification algorithm was able to predict about half of the POX examples correctly. Thus much of the blame must be attributed to the probabilistic model, or at least the particular classification tree used. We defer our interpretation of this discrepancy until the discussion of modeling versus leveraging correlations.

5.6 Study 2: Comparison of Four Classifiers

The section describes the work of Horton & Nakai[11]. To improve the classification accuracy and to provide a baseline for our probabilistic model we implemented three other classification algorithms. For this study we used Fayyad-Irani binning to handle continuous variables but otherwise the probabilistic model was left unchanged. The three other classifiers are all standard classifiers from the fields of machine learning and pattern recognition, however we describe each one here for the convenience of the reader.

5.6.1 k Nearest Neighbors

The k nearest neighbors classifier [3] stores the complete training data. New examples are classified by choosing the majority (or plurality) class among the k closest examples in the training data. For our particular problem, we first used a linear transformation to normalize the feature values to lie within the interval [0,1] and then used the Euclidean,

i.e. sum of squares, distance to measure the distance between examples. Figure 5.6 gives a graphical interpretation of using kNN to classify objects with two features.

5.6.2 Binary Decision Tree

Binary decision trees [20] recursively split the feature space based on tests that test one feature variable against a threshold value. Figure 5.7 gives a graphical depiction of the binary decision tree classifier which can be compared to the graphical depiction of kNN. It can be seen that a decision tree partitions the feature space into regions whose boundaries are hyperplanes. The form of the tests at each node of the decision tree constrains the hyperplanes to be perpendicular to one of the axes of the feature space.

Table 5.10 shows pseudocode for inducing a binary decision tree from a set of labeled training examples. The tree is constructed by choosing a test consisting of a feature and threshold value that splits the training data into two partitions that have the most biased distribution of class labels possible. We used the weighted entropy, usually known as the information gain criteria, for measuring the amount of bias of the class distribution. Once the test is selected a χ^2 statistic is used to check if the class distribution in the two partitions induced by the test are significantly different from the class distribution of the training data, i.e. the union of the two partitions. If they are not significantly different the test is not added and the node becomes a leaf in the decision tree labeled by the most common class label amonst its training data. If however the χ^2 test shows a significant difference, then the procedure is repeated recursively using the induced partitions as training data. As applied here this statistical test is known as top-down pruning. Note that when used with continuous variables the decision tree inference algorithm can be viewed as a strict generalization of the Fayyad-Irani binning described above.

5.6.3 Naïve Bayes Classifier

The naïve Bayes classifier [7], [14] is an approximation to an ideal Bayesian classifier which would classify an example based on the probability of each class given the example's feature variables. For features $F_1, F_2, \cdots, F_n$ an ideal Bayesian classifier would classify according to

$$P[C|F_1, F_2, \cdots, F_n].$$

```
// classes is the set of class labels, c is the cardinality of classes.
main{
    Create root node
    split_if_justified( root )
}

split_if_justified( node n, set relevant_examples ){
    feature f, threshold t
    chose a pair ( f, t ) such that weighted_entropy( f, t, relevant_examples ) is minimal
    left_examples ⟵ examples ∈ relevant_examples such that feature f < t
    right_examples ⟵ examples ∈ relevant_examples such that feature f ≥ t
    if null_hypothesis_rejected( relevant_examples, left_examples, right_examples )
        Create new nodes left_child, right_child
        Add left_child, right_child to n
        split_if_justified( left_child, left_examples )
        split_if_justified( right_child, right_examples )
    else
        label n with the most common class in relevant_examples
}

weighted_entropy( feature f, threshold t, set relevant_examples ){
    left_examples ⟵ examples ∈ relevant_examples such that feature f < t
    right_examples ⟵ examples ∈ relevant_examples such that feature f ≥ t
    return (|left_examples|*H(left_examples)+|right_examples|*H(right_examples)) / |relevant_examples|
}

H( set examples ){
    P_1, P_2, ..., P_c ⟵ proportion of each class in examples
    return -Σ_{i=1}^{c} P_i log P_i
}

null_hypothesis_rejected( set relevant_examples, left_examples, right_examples )
    deviation ⟵ 0
    for each class in classes
        expected ⟵ proportion of class in relevant_examples
        for each partition in { left_examples, right_examples }
            observed ⟵ proportion of class in partition
            deviation ⟵ deviation + (observed - expected)^2 / expected
    if deviation > χ^2_{confidence,c-1} return true
    else return false
}
```

Table 5.10: Pseudocode for binary decision tree learning is shown.

That is the probability of the class given all of the features. Unfortunately it is generally not feasible to obtain enough training data to effectively sample the exponential number of combinations of values for F_1 though F_n. The naïve Bayes classifier deals with this by assuming that the probability of a feature given its class is independant of the probability of any other feature. More precisely, we assume

$$P[F_1, F_2, \cdots, F_n|C] = \prod_i P[F_i|C].$$

This assumption can be combined with Bayes law to obtain an approximation to the ideal Bayes classifier:

$$P[C|F_1, F_2, \cdots, F_n] = \frac{P[C]P[F_1, F_2, \cdots F_n|C]}{P[F_1, F_2, \cdots, F_n]} \propto P[C] \prod_i P[F_i|C]$$

The advantage of using this approximation is that we only need to estimate statistics for each possible feature value and not for combinations of feature values.

5.6.4 Evaluation Methodology

For this study we used stratified cross-validation to estimate the accuracy of the classifiers. In this procedure the dataset is randomly partitioned into equally sized partitions subject to the constraint that the proportion of the classes in each partition is equal. Empirical tests have indicated that this procedure provides more accurate estimates than plain cross-validation [13].

We employed a cross-validated paired-differences t test to establish the statistical significance of the difference in performance between two classifiers [13] (a general description of the paired t test for hypothesis testing can be found in introductory textbooks on statistics, for example [15]). This test makes two assumptions. One assumption is that the difference of the performance of the two algorithms is normally distributed. The second assumption is that the performance difference on different test partitions of the cross-validation is independent. In general both of these assumptions may be violated, in particular the training partitions of the cross-validation overlap heavily and thus the trials are not independent [21]. Despite these observations, the t test has been shown empirically to discriminate adequately [2].

Results with the *E.coli* Dataset for 4 Classifiers

Partition	*k*NN	Dec. Tree	Naïve Bayes	HN
0	89.28	83.33	82.14	84.42
1	95.24	80.95	84.52	88.10
2	84.52	88.10	82.14	88.10
3	76.19	69.05	75.00	69.05
mean	86.31	80.36	80.95	82.44
std. dev.	8.04	8.10	4.12	9.08

Table 5.11: The results of cross-validation are shown in units of percent accuracy, including the mean and sample standard deviation. HN is the probabilistic model of Horton & Nakai. All trials of *k*NN are for $k = 7$.

5.6.5 Results of Study 2

A summary of the accuracies of the different classifiers is given in table 5.11 for *E.coli* and table 5.12 for yeast. Accuracies for the smaller *E.coli* dataset were estimated with 4-fold cross-validation to keep the test partitions reasonably large. It can be seen that the mean accuracy of *k*NN is higher than the other 3 classifiers for both datasets. Using the cross-validated paired t test to test whether the mean accuracy of *k*NN is different than the other classifiers gives t values of 2.86, 2.59, and 2.88 against the binary decision tree, Naïve Bayes, and HN respectively. For a two-sided t test with nine degrees of freedom the t value corresponding to a confidence level of 0.95 is 2.2622. By the same t test the only significant difference for the *E.coli* dataset is the difference between *k*NN and Naïve Bayes which has a t value of 3.3169 and is significant at a confidence level of 0.95.

5.6.6 *k* Parameter

For accuracy estimation we used k values for *E.coli* and yeast datasets of 7 and 21 respectively. We determined those values by doing leave-one-out cross-validation on each training partition (this is a nested cross-validation) and taking the best overall value. Since this procedure averages over all the data and therefore indirectly uses the test data, it is important to know how sensitive the classification accuracy is to the choice of k. figure 5.8 shows the relationship between the k value and accuracy estimated by cross-validation for the *E.coli* dataset. The accuracy is highest for k values of 5 and 7 but is higher than the other three classifiers from $k = 3$ to $k = 25$. Figure 5.9 shows the corresponding graph for yeast. With the larger yeast dataset the highest accuracy is achieved for k values from 21

Results with the Yeast Dataset for 4 Classifiers

Partition	kNN	Dec. Tree	Naïve Bayes	HN
0	55.78	55.10	53.74	55.10
1	59.18	51.02	57.82	55.78
2	60.96	56.16	56.16	58.22
3	65.75	58.22	58.22	55.48
4	48.63	50.00	45.21	47.95
5	62.33	57.53	54.11	53.42
6	68.49	65.75	60.27	67.81
7	58.90	57.53	61.64	56.16
8	56.85	56.85	56.16	55.48
9	58.22	57.53	59.59	57.53
mean	59.51	56.57	56.29	56.29
std. dev.	5.49	4.30	4.66	4.93

Table 5.12: The results of cross-validation are shown in units of percent accuracy, including the mean and sample standard deviation. HN is the probabilistic model of Horton & Nakai. All trials of kNN are for $k = 21$.

to 25, but the accuracy for kNN is higher than the other classifiers for values of k from 9 to 99. We did not calculate the accuracy for $k > 99$.

5.6.7 Local Alignment Distance with kNN

To provide a baseline comparison for the effectiveness of the expert identified features used we calculated the accuracy of kNN with the local alignment distances calculated using the PAM120 matrix. Using the same cross-validation partitions and criteria for choosing k we obtained an accuracy of 67.86% on the *E.coli* dataset. This is much higher than the 42.9% accuracy that the majority class classifier achieves but is much lower than the 86.31% accuracy achieved by kNN using the expert identified features.

On the yeast dataset the local alignment distances did relatively better, yielding an accuracy of 52.1%. However, the t test still shows this accuracy to be significantly lower than the accuracy of the four classifiers with the expert identified features at a confidence level of 0.97.

5.6.8 Confusion Matrices

In order to identify common misclassifications we calculated the confusion matrix for both datasets using kNN with the expert identified features. These results are shown in

Confusion Matrix for *E.coli* dataset with *k*NN

	cp	imL	imS	imU	im	omL	om	pp
cp	141	0	0	0	0	0	0	2
imL	0	0	0	0	1	1	0	0
imS	0	0	0	1	0	0	0	1
imU	1	0	0	23	11	0	0	0
im	3	0	0	14	58	0	0	2
omL	0	0	0	0	0	4	1	0
om	0	0	0	0	0	0	18	2
pp	4	0	0	0	1	0	0	47

Table 5.13: The actual class labels are shown in the vertical column. The predicted class labels are shown in the row across the top. Thus 2 proteins that localize to the cytoplasm were incorrectly predicted to be localized to the periplasm.

Confusion Matrix for yeast dataset with *k*NN

	cyt	erl	exc	mel	me2	me3	mit	nuc	pox	vac
cyt	314	0	1	0	2	3	32	91	1	0
erl	0	0	3	1	1	0	0	0	0	0
exc	4	0	22	4	2	0	2	1	0	0
mel	0	0	8	33	0	1	2	0	0	0
me2	9	0	7	10	11	3	7	4	0	0
me3	18	0	0	0	1	122	6	16	0	0
mit	62	0	4	2	5	8	141	19	3	0
nuc	171	0	0	0	2	10	27	216	0	0
pox	4	0	1	1	0	0	1	2	11	0
vac	13	0	3	1	1	6	1	5	0	0

Table 5.14: The actual class labels are shown in the vertical column. The predicted class labels are shown in the row across the top.

tables 5.13 and 5.14.

5.7 Discussion of Study 2

The confusion matrix for *E.coli* is very encouraging in that most of the mistakes can be seen to result from confusing inner membrane proteins without a signal sequence with inner membrane proteins with an uncleavable signal sequence and *vice versa*. We consider this to be a relatively minor error for two reasons. First, for some uses the distinction between different types of inner membrane proteins may be immaterial. Second, the defi-

nition of the presence or absence of an uncleavable signal sequence is somewhat arbitrary and thus the labels of the training examples include some uncertainty. If we collapse the two classes to form a class "inner membrane protein without a cleavable signal sequence" we attain a surprisingly high accuracy of 94%!

The confusion matrix for the yeast dataset shows that most of the error is due to confusing cytoplasmic proteins with nuclear proteins and *vice versa*. This reflects a fundamental difficulty in identifying nuclear proteins. One component of that difficulty comes from the fact that unlike other localization signals, the nuclear localization signal does not appear to be limited to one portion of a protein's primary sequence [6]. Another component is the fact that in some cases a protein without a nuclear localization signal may be transported to the nucleus as part of a protein complex if another subunit of the complex contains a nuclear localization signal [28].

Another interesting result is the relatively low accuracy of using *k*NN with the local alignment distance. This is interesting because the common practice of inferring protein function by sequence similarity search of the databases is essentially a variant of *k*NN with local alignment distance. Our results show that localization site prediction is an example of a protein classification problem where domain specific features are much more effective than sequence similarity alone.

One question we would like to answer is why *k*NN was more effective than the other classifiers. It is easy to point out some shortcomings with the other classifiers, the binary decision tree and HN suffer from data fragmentation as the data is repeatedly partitioned. Naïve Bayes has a fixed number of parameters and does not asymptotically approach an optimal classifier as the number of training examples increases. However we do not have a solid answer as to why *k*NN performs better on this task.

In summary, in this study we demonstrated that *k*NN with expert identified features is superior to three other classifiers for classifying proteins based on their cellular localization sites. For the yeast dataset this difference can be shown to be statistically significant. We also showed that the expert identified features are much more effective than local alignment distance and that most of the classification errors on the *E.coli* dataset are relatively minor errors. The use of *k*NN and better testing methodology allowed us to achieve estimated accuracies of 60% and 86% for the yeast and *E.coli* datasets respectively, exceeding the accuracies of 55% and 81% from study 1.

5.8 Study 3: Finding Substring Features from Protein Sequence Data

This section describes a method[9] for generating and selecting substrings that correlate to specific classes and preliminary application of the method to the prediction of protein localization sites in *E.coli*. The method uses a generalized suffix tree to efficiently compute correlations between substrings and classes, and a statistical test to select a few promising substrings to use as features.

5.9 System Overview

Our basic goal was to automatically generate features that can be used to *augment* features identified by a human biologist. Our approach to this is shown schematically in figure 5.10. The substrings present in the database are automatically organized and counted by the generalized suffix tree. These features then pass through a simple filter which reduces their number to a manageable level. At this point a decision tree is induced based on those features. The features selected to be part of that decision tree are then combined with the human expert defined features to produce the final feature set. Finally we induce a decision tree using those features.

5.9.1 Counting Substring Occurrences with the Suffix Tree

Classifier systems designed for biomolecular sequences often exploit correlations between the distribution of fixed length substrings in a set of sequences and the function of those sequences, for example [23]. It is a known but relatively unexploited fact that the generalized suffix tree data structure can be used to efficiently examine the distribution of *all* the substrings occurring in a collection, i.e. the substrings of any length. We implemented Ukkonen's suffix tree construction algorithm [24], [8] in C++. The algorithm was slightly modified to add a class count for each suffix. We then built a generalized suffix tree from the protein sequences in the training set. We prepended an "n" and appended a "c" (for N and C-terminus) to each protein sequence so that a pattern such as the yeast C-terminal HDEL could be represented. After the tree construction was complete our software propagated class counts for suffices up the tree. The result was a tree with a vector of class counts for each node, and therefore for each substring.

5.9.2 Feature Selection

As shown in figure 5.10 we used a two stage process for feature selection. The first stage used a statistical test to identify promising substrings. Specifically we compared the fraction of occurences of class i within the training sequences which contained a substring feature f to the fraction of class i within the overall training set. In other words we look for substring features f such that for some class C_i we can reject the null hypothesis:

$$Pr[C_i|f] = Pr[C_i]$$

We used a confidence level of 0.9995 which resulted in a manageable number ($\approx$ 400) of preliminary features. At this stage we added the length of the protein sequence as an additional preliminary feature. We then induced a binary decision tree with those features using a strict confidence of 0.995 for the χ^2 test. The substring features chosen by the decision tree induction algorithm were then combined with the human expert identified features as the final feature set. The confidence level for the χ^2 test was then relaxed to 0.95 and a decision tree was induced on this final feature set.

5.9.3 Classifier for Study 3

Although study 2 showed that kNN gives a somewhat higher prediction accuracy we chose to use the binary decision tree for this study. There were two reasons for our choice: first, the binary decision tree has a nice visual representation which makes it relatively easy to interpret how features are being used and second the decision tree inference algorithm has a form of feature selection built into it which we used to further cull the candidates which passed our χ^2 test.

5.9.4 Distribution of Significant Substrings

First we investigated whether the ability to find substrings of arbitrary length was in fact useful. Figure 5.11 shows the distribution of the lengths of substrings in the dataset which had a significant correlation with at least one class as determined by the first stage of the selection procedure shown in figures 5.10. Most of the substrings are of length two to four but every length from one to eighteen has at least one representative. This data supports the claim that substrings of various lengths should be considered as candidate features. Another interesting point is that all of the possible length one substrings passed

the first stage of selection. This underscores the fact that the substring features described here are a strict generalization of using amino acid composition as a feature.

5.9.5 Results and Discussion of Study 3

In this section we present the results of 4-fold stratified cross validation tests using the binary decision tree classifier for classification. Table 5.15 shows the accuracy of different feature sets. Unfortunately the accuracy obtained is not especially encouraging. Although the substring features alone perform better than the majority class (54.2% *vs.* 42.9%) they did not perform as well as using sequence similarity with *k*NN with which we obtained an accuracy of 67.9% in study 2. Furthermore the use of the combination of the substring features with the expert identified features did not significantly raise the classification accuracy above that obtained using the expert identified features alone. However we did have a surprising anecdotal success in terms of automatically identifying a biologically meaningful feature. Figures 5.12, 5.13, 5.14, and 5.15 show four examples of decision trees induced with the preliminary substring features that passed the first stage statistical filter. Upon inspecting those trees we noticed that three out of the four trees test for the presence of a C-terminal phenylalanine to classify an example as an outer membrane protein. The author was unaware of the significance of this feature but a literature search revealed that this feature has been experimentally determined to be required for some proteins to localize to the outer membrane and was deemed important enough to be the main result of a journal article[22]. Thus we consider this method promising but concede that it must be refined before it can be used to increase the classification accuracy on this problem. Part of the problem may come from counting occurrences of *exact* matches to substrings rather than allowing approximate matches, since many localization signals are rather loosely defined. One possible scheme would be to add a stage to our selection process which uses the substrings which scored well with the χ^2 test as "consensus sequences" and counting the occurrences of substrings which match each consensus sequence when a few substitutions are allowed. The resulting features would then reflect approximate occurrences as well as exact occurrences.

Classification Accuracy with Different Feature Sets

Feature Set	χ^2 value	Mean	S.D.
Majority Class	NA	42.9%	0.94%
Preliminary Substrings	0.995	54.2%	4.51%
Expert Features	0.95	80.4%	8.10%
Expert Features + Preliminary Substrings	0.95	78.8%	6.96%
Expert Features + Preliminary Substrings	0.995	78.6%	6.07%
Expert + Select Substrings	0.95	80.7%	8.15%

Table 5.15: The accuracy obtained with 4-fold stratified cross-validation for different feature sets is shown. The first entry shows the accuracy of simply choosing the majority class. The second column is the χ^2 value used for pruning the decision trees. The last column reports the standard deviation of the accuracy for different partitions of the cross-validation. The "preliminary substrings" are the substrings which survive the first stage of feature selection described in the text. The "select substrings" are the substrings which survive the second stage as well.

5.10 Chapter Discussion

5.10.1 Modeling *vs.* Leveraging All Correlations

Throughout this chapter there is a sort of tension between two separate goals. One goal is to model the mechanism of protein localization, while the other is to maximize the classification accuracy. To some extent the two goals are complementary, certainly if we had a perfect model of the mechanism of protein localization, we could use that as an optimal classifier simply by simulating the localization process for any protein sequence of interest. However, the converse is not true. It is possible to use correlations to improve the classification accuracy that have no causal relationship to the actual process of localization.

The work in this chapter makes some effort to achieve both goals but the emphasis shifts towards classification accuracy as the chapter progresses. In study 1 the classification trees extended the biological knowledge reflected in the expert identified features by dictating that the features be used in the way in which they were originally intended. However it should be noted that a few of the "expert identified features" such as *aac* and *vac* (see tables 5.3, 5.5) were actually the results of discriminant analysis. By using general purpose classifiers in study 2 we were able to raise the classification accuracy somewhat, but in the process allowed features to be used in unintended ways. For example, despite the fact

that the results of study 1 indicated that the *pox* feature variable did not discriminate well enough to correctly predict proteins which where localized to the peroxisomes, in study 2 *k*NN was able to correctly predict 11 out of 20 examples. It seems likely that *k*NN used correlations from other feature variables to accomplish that. This trend was taken a step further in study 3 where correlation was explicitly used as a criterion for choosing features.

We believe that at this point it may be best to develop two systems: one which is solely concerned with achieving a high classification accuracy and one which tries to model the protein localization process as closely as possible. For the first system it is easy to imagine taking measures to increase the classification accuracy once all pretenses of modeling are abandoned. For example, proteins with known DNA-binding motifs can usually be expected to be localized to the nucleus, one of the localization sites which our current system has the most trouble with. Use of this kind of prior knowledge about the correlations between sequence and function and between function and localization site may substantially improve the prediction accuracy. The second system could be improved as well. One interesting direction would be to attempt to model the dynamics of protein localization. On the biology side, chase studies which combine immunoflouresence with the timed suppression of protein synthesis could provide an experimental technique for gathering data. While on the computer science side the use of dynamic Bayesian networks may provide useful as a framework for a model which is learned from training data. It should be noted that besides the scientific interest in modeling biological processes, there is another advantage to the approach of emphasizing relationships which reflect the localization mechanism. Sequence similarity based sequence comparisons, which are routinely done on new protein sequences, can be expected to primarily reflect functional relationships rather than localization signals. Thus a model of protein localization could provide additional, relatively independent evidence to the biologist.

5.10.2 Other Applications

This chapter can be considered to be a case study on the effective application of machine learning to a biosequence classification problem. Features were chosen primarily by using prior domain knowledge but machine learning was applied to discover features which complement those based on prior knowledge. A classifier which reflected the structure of the problem was then constructed by hand and compared to several standard classification

algorithms. The result is a system which effectively leverages both biological knowledge and machine learning. We believe that this approach can be successful for other classification problems in biology. We should note that hidden Markov models (HMM's) have enjoyed much success in the area of protein classification, however it is not clear how to apply them to problems which are not naturally viewed as generating sequences from a distribution. The methodology applied in this chapter offers an alternative to HMM's which clearly separates the features used for classification and the classification algorithm. Interestingly it would be quite plausible for the output of an HMM to be used as a feature.

5.11 Software

We have developed a C program to implement the computations necessary to perform the probabilistic inference described above in the probabilistic model section. The program also learns the conditional probability tables (function) using any of the three methods described in the section on conditional probabilities. Alternatively the program can read in conditional probability tables from a file and use them when doing inference. The program inputs a file which describes the topology and feature variables of the classification tree in a simple language and another file which contains the values of the feature variables for the objects. The output of the program is the probability of each leaf class for each input object. The other three classifiers were implemented in C, or both C and Perl for kNN. Ukkonen's linear time suffix tree construction algorithm was implemented in C++. All of the software described in this chapter is available upon request.

5.12 Acknowledgments

I would like to thank Kenta Nakai showing me this wonderful application and working with me on study 1 and 2, and also for his helpful comments on study 3, particularly for pointing out the paper on the C-terminal phenylalanine. Geoff Zweig and Kevin Murphy made helpful comments on the machine learning aspects of study 1 and 2.

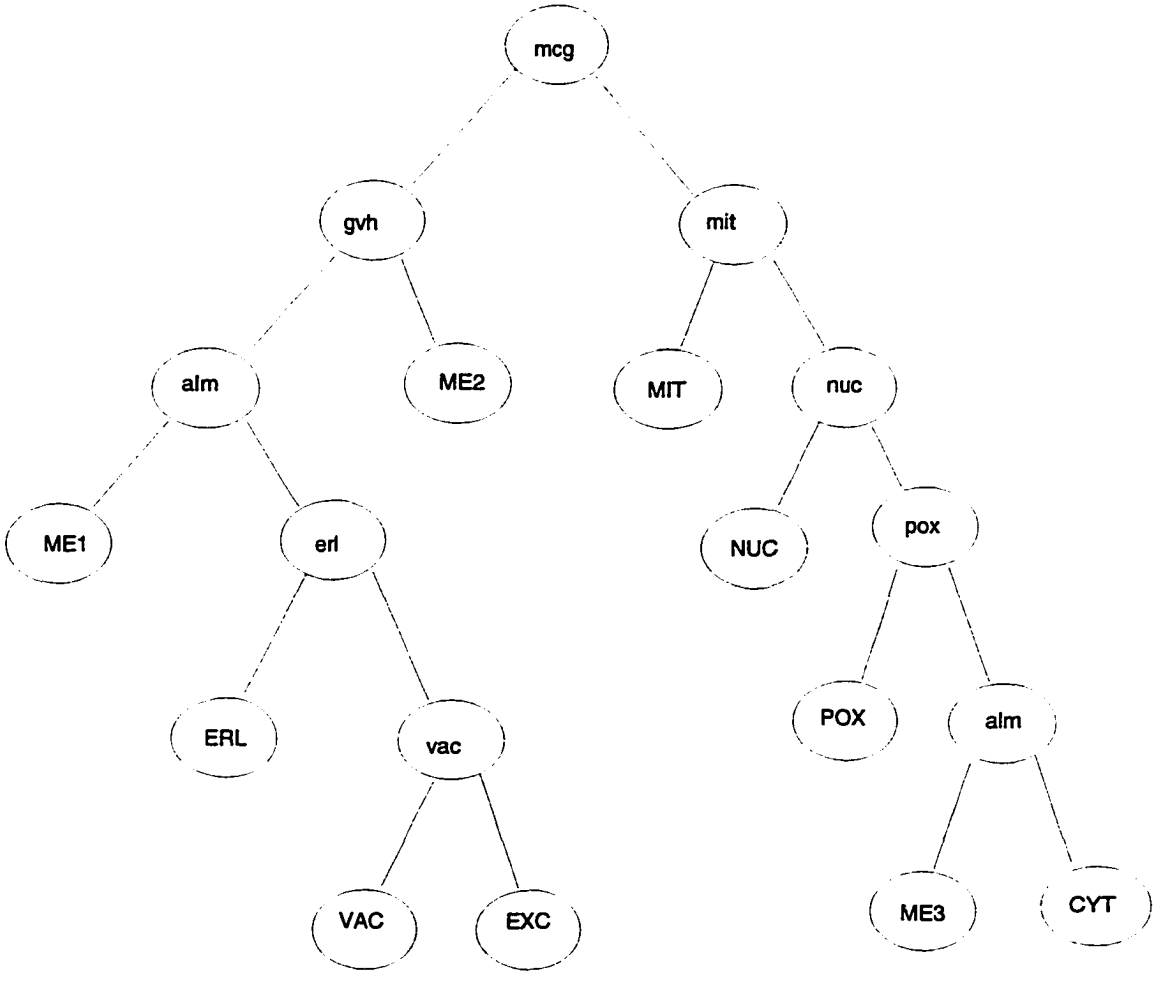


Figure 5.5: The classification tree used for yeast protein localization is shown. The leaf nodes are labeled with the class that they represent, while the other nodes are labeled with their feature variable. All the edges shown are directed downward and the edges connecting feature variable to their classification variables are not shown explicitly. The labels are abbreviations (see tables 5.4 and 5.5).

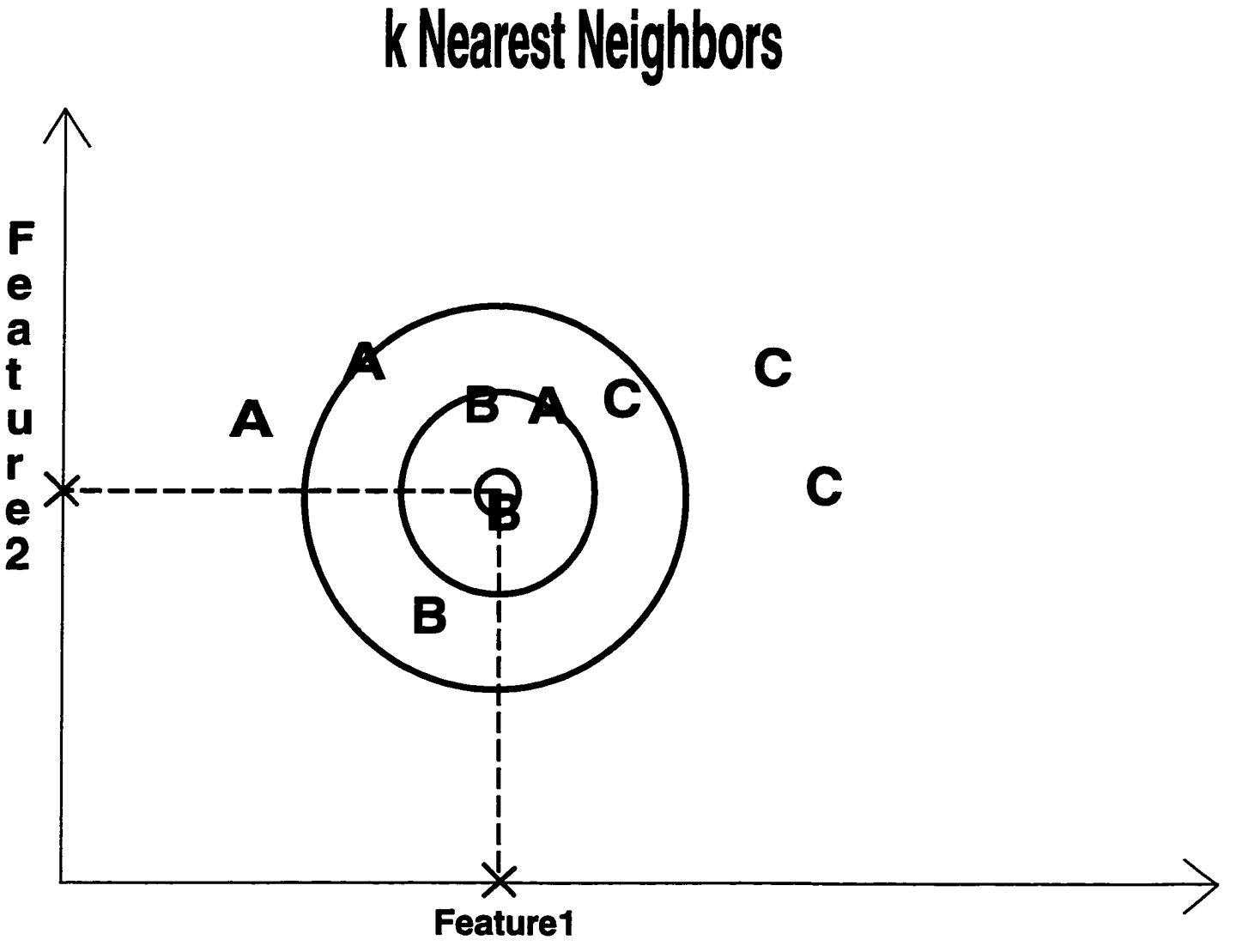


Figure 5.6: The use of the kNN classifier to classify objects with two features is shown. The coordinates labeled A, B, or C represent examples in the training set labeled as class A, B, or C. The three concentric cirles represent k values of 1, 3, and 5. In each case the object at the center of the cirle would be classified as belonging to class B.

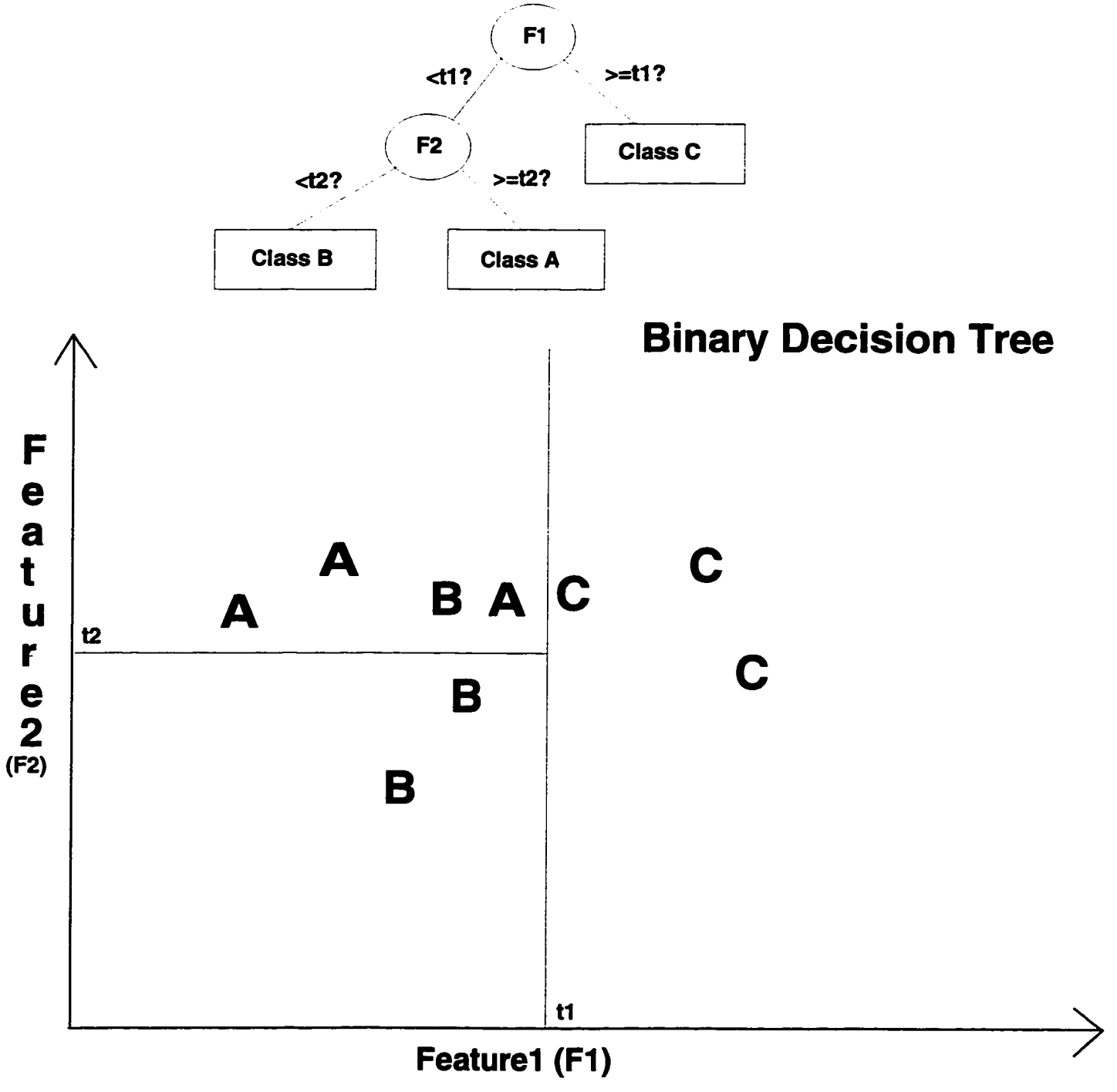


Figure 5.7: The use of the binary decision tree classifier to classify objects with two features is shown. The coordinates labeled A, B, or C represent examples in the training set labeled as class A, B, or C. The decision tree shown in the upper part of the figure corresponds to dividing the space with the line $f1 = t1$ and the ray $f2 = t2, f1 < t1$ as shown in the lower part of the figure.

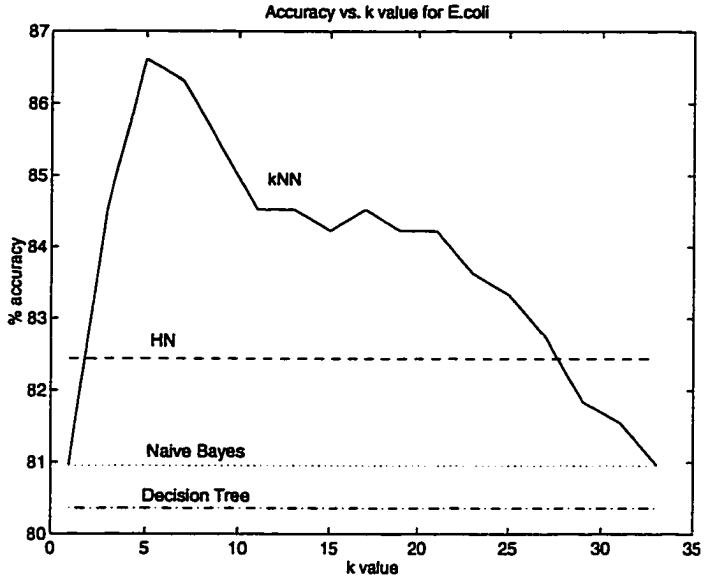


Figure 5.8: The accuracy of kNN for the *E.coli* dataset is shown for odd k from 1 to 33. The accuracy of the decision tree, Naïve Bayes, and HN is also shown.

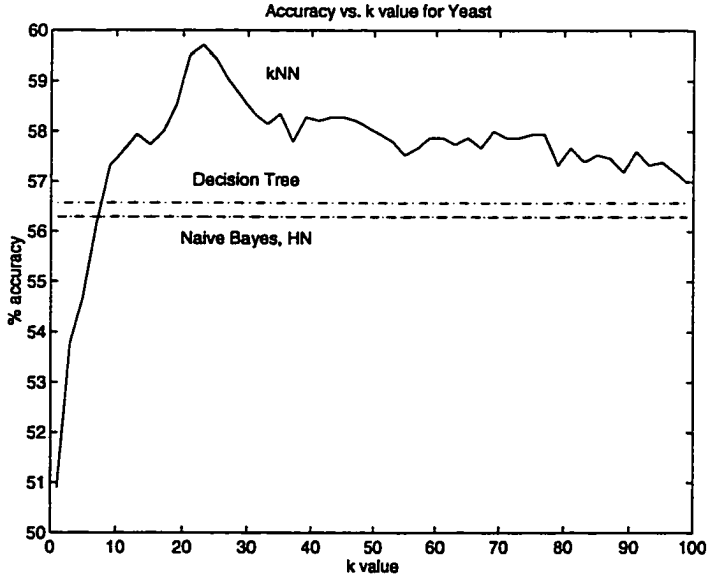


Figure 5.9: The accuracy of kNN for the yeast dataset is shown for odd k from 1 to 99. The accuracy of the decision tree, Naïve Bayes, and HN is also shown.

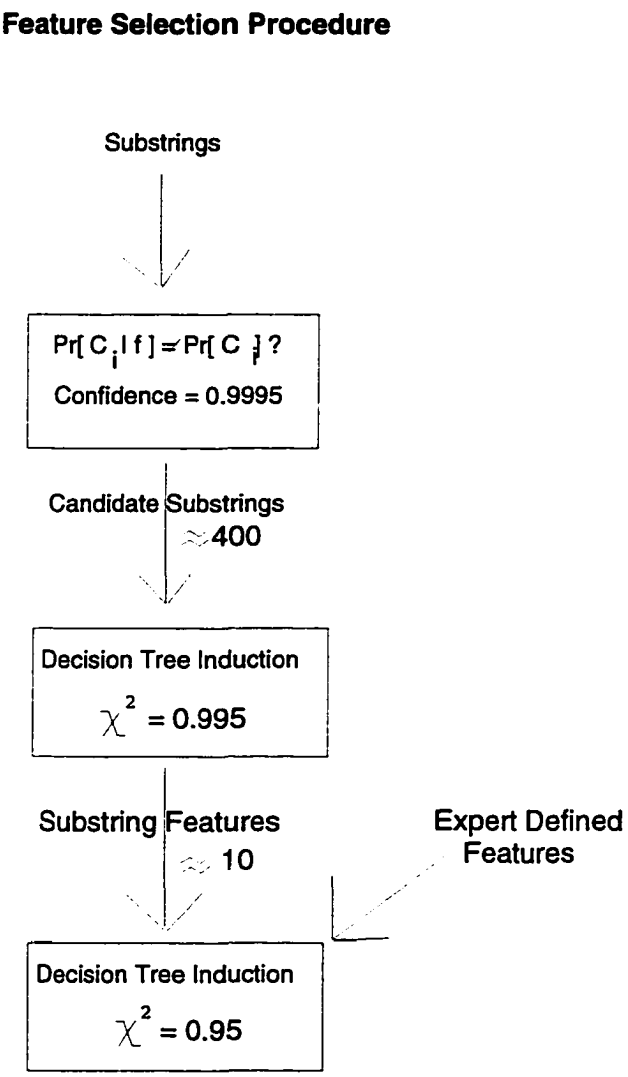


Figure 5.10: An overview of the generation, selection, and use of substring features is shown.

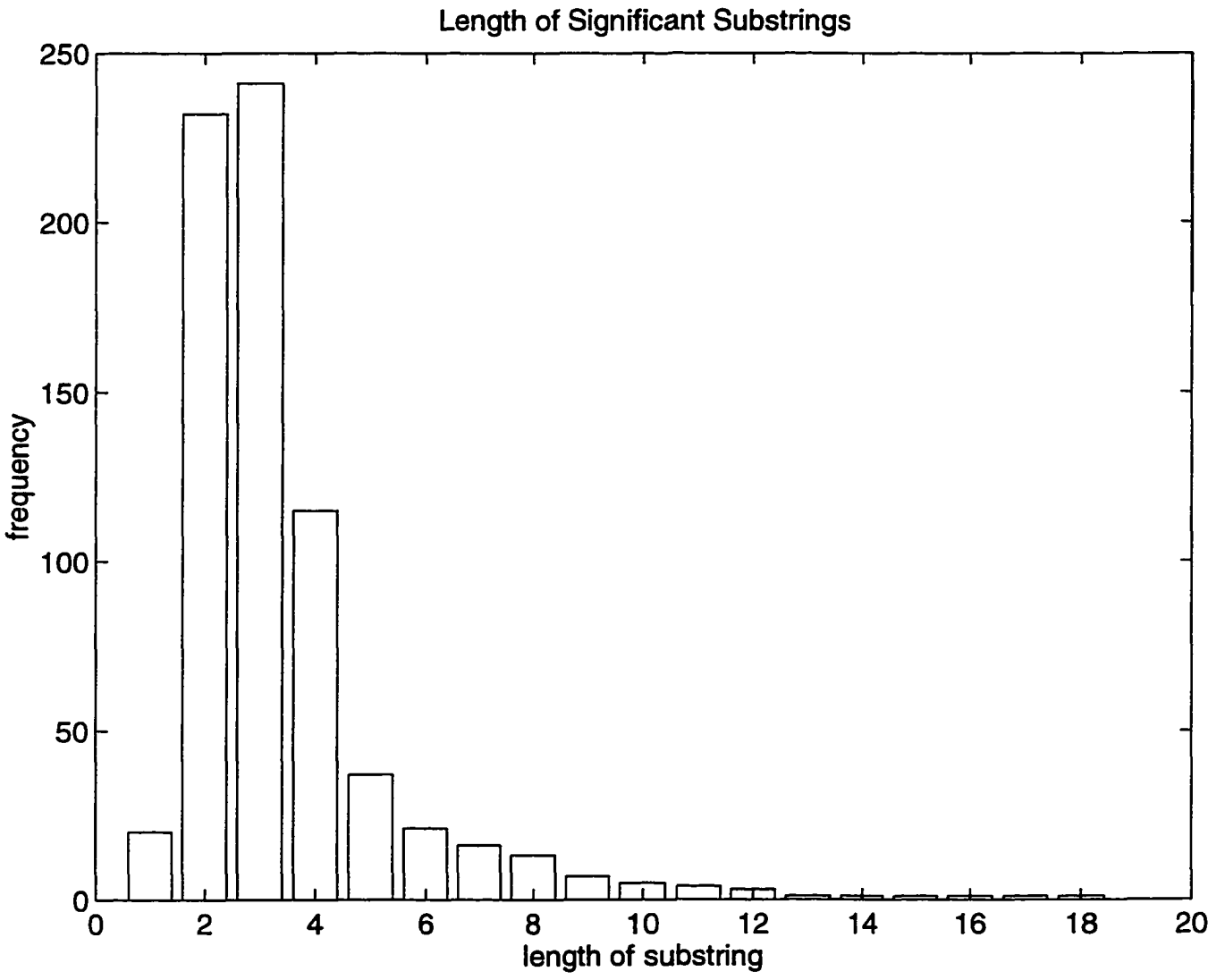


Figure 5.11: A histogram of the distribution of substrings which correlate significantly to at least one class is shown.

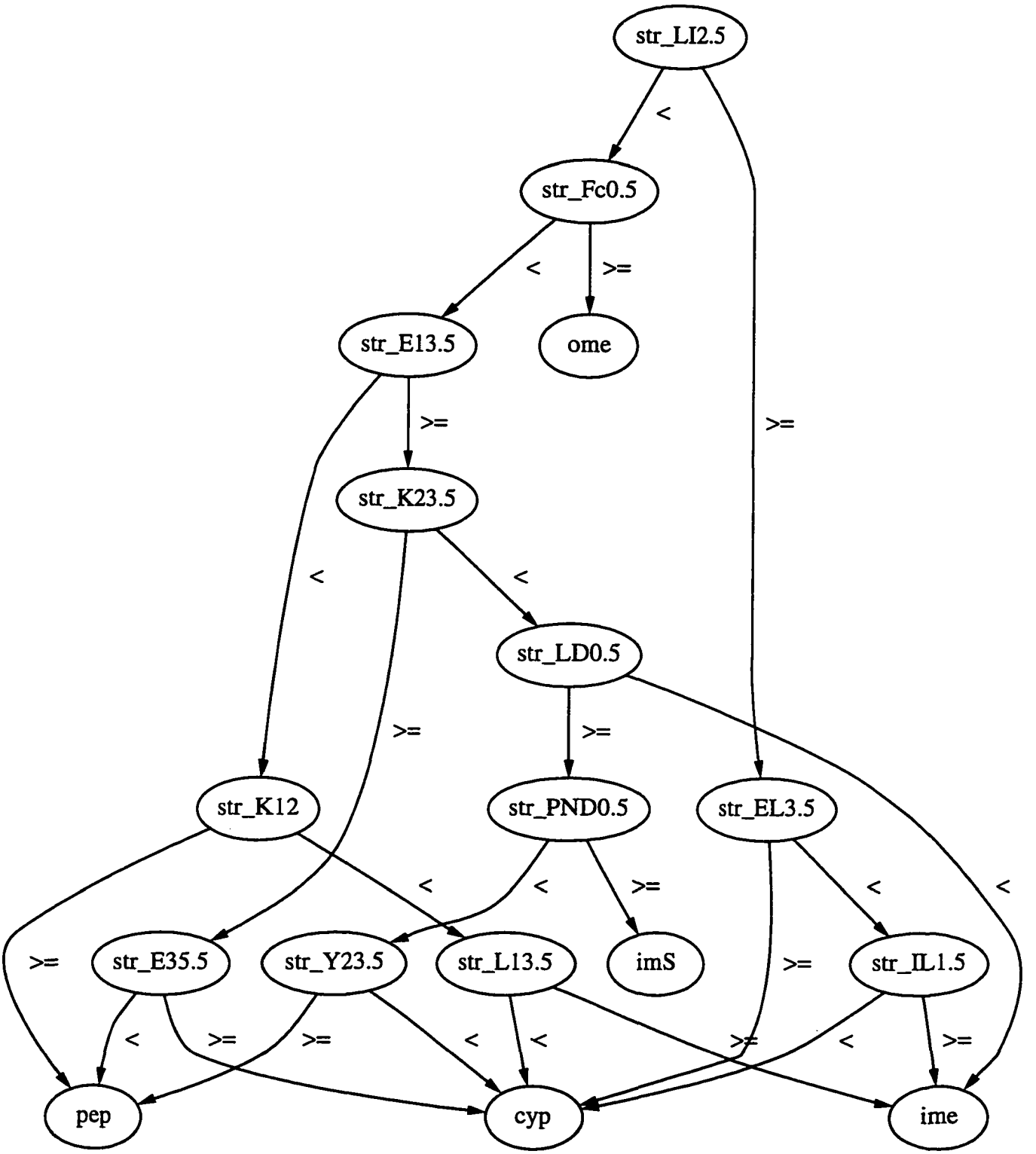


Figure 5.12: The decision tree induced from the first partition of cross-validation with the substring features that passed the first statistical test is shown. Internal nodes are labeled with their features and thresholds. Substring feature have names starting in "str_". Thus the node labeled "str_EL3.5" tests for 4 or more occurences of the substring "EL".

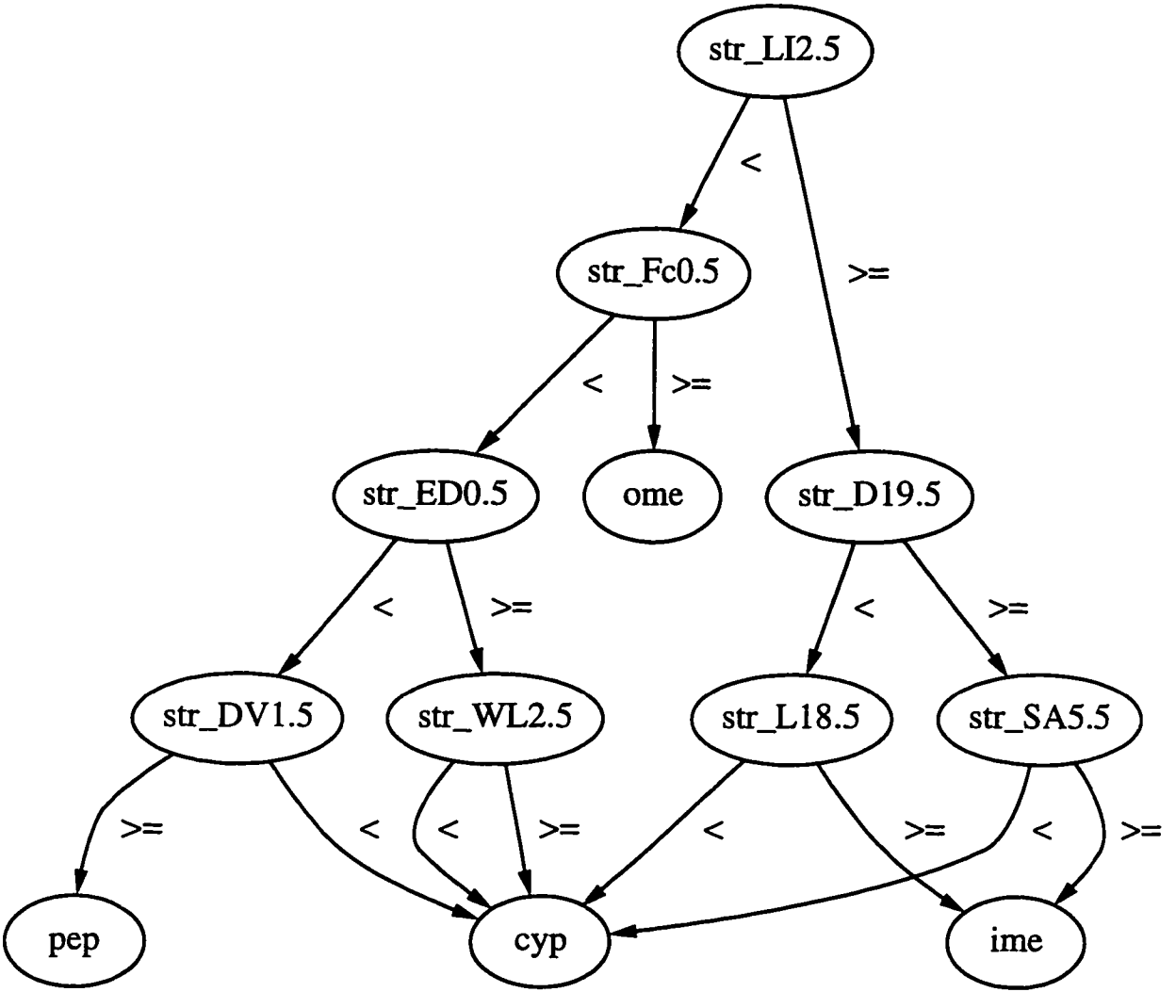


Figure 5.13: The decision tree induced from the second partition of cross-validation with the substring features that passed the first statistical test is shown. Internal nodes are labeled with their features and thresholds. Substring feature have names starting in "str_". Thus the node labeled "str_WL2.5" tests for 3 or more occurences of the substring "WL".

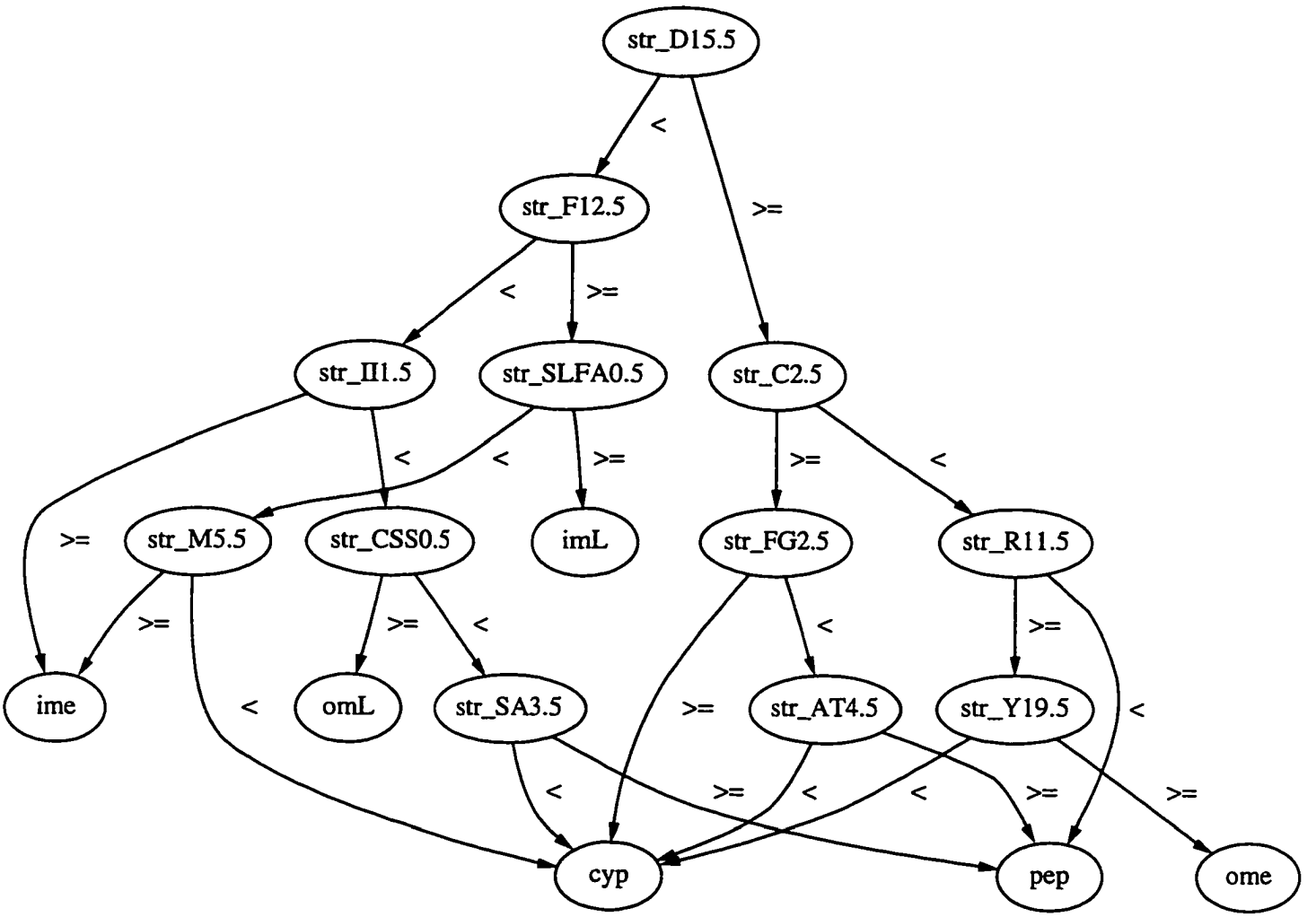


Figure 5.14: The decision tree induced from the third partition of cross-validation with the substring features that passed the first statistical test is shown. Internal nodes are labeled with their features and thresholds. Substring feature have names starting in "str_". Thus the node labeled "str_SLFA0.5" tests for 1 or more occurences of the substring "SLFA".

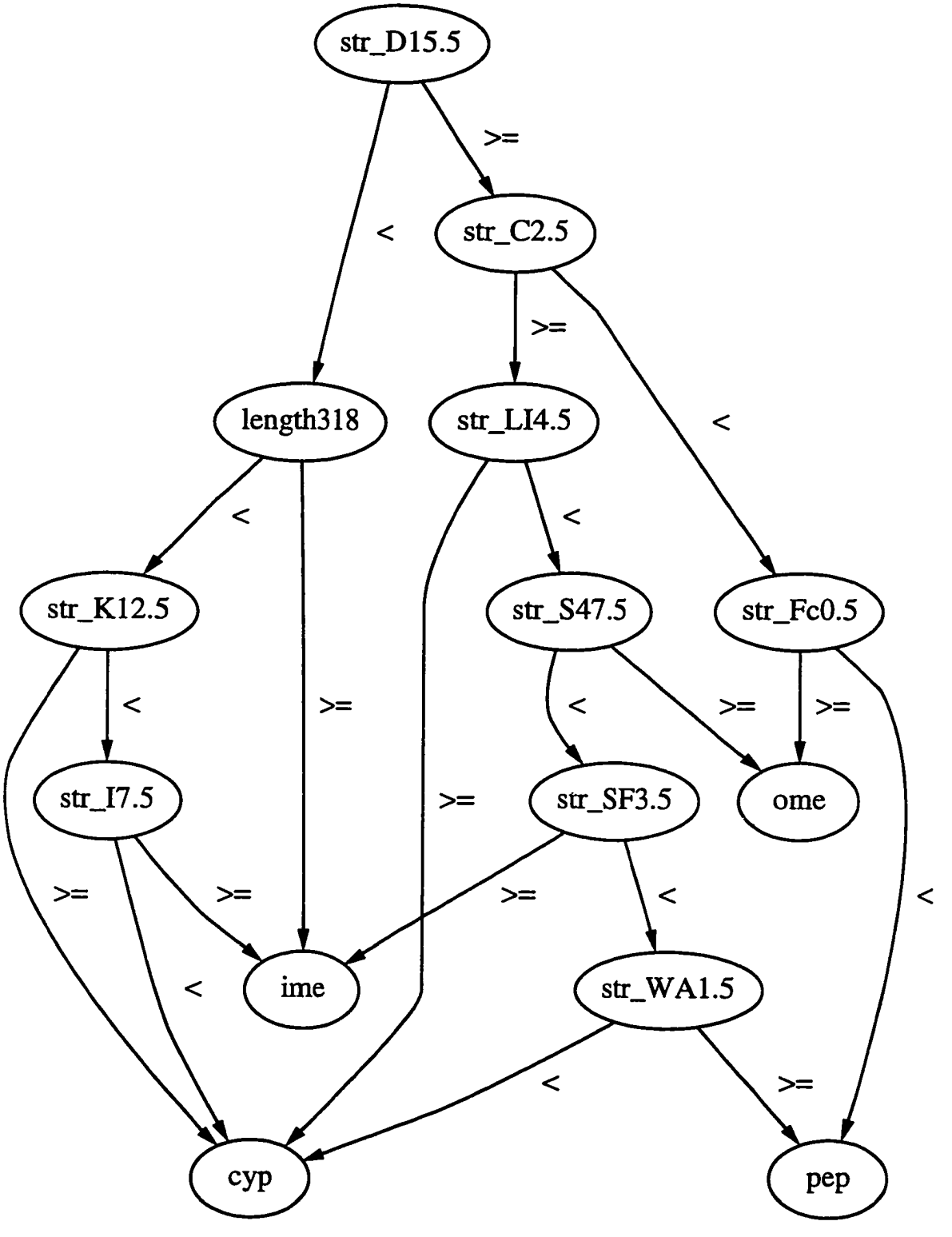


Figure 5.15: The decision tree induced from the fourth partition of cross-validation with the substring features that passed the first statistical test is shown. Internal nodes are labeled with their features and thresholds. Substring feature have names starting in "str_". The node labeled "length318" tests for sequences of length greater than or equal to 318.

Bibliography

[1] A. Bairoch and R. Apweiler. The swiss-prot protein sequence data bank and its new supplement trembl. *Nucleic Acids Research*, 24:21–25, 1996.

[2] Thomas G. Dietterich. Statistical tests for comparing supervised classification learning algorithms. http://www.CS.ORST.EDU/ tgd/cv/jr.html, 1996.

[3] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.

[4] Harvey Lodish *et al. Molecular Cell Biology.* Scientific American Books, 1995.

[5] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[6] J. Garcia-Bustos, J. Heitman, and M. N. Hall. Nuclear protein localization. *Biochimica et Biophysica Acta*, 1071:83–101, 1991.

[7] I. J. Good. *The Estimation of Probabilities: An Essay on Modern Bayesian Methods.* MIT Press, 1965.

[8] Dan Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge Press, 1997.

[9] Paul Horton. Using substrings to classify proteins by their cellular localization sites. Class Project for CS281 at UC Berkeley, 1996.

[10] Paul Horton and Kenta Nakai. A probabilistic classification system for predicting the cellular localization sites of proteins. In *Proceeding of the Fourth International Conference on Intelligent Systems for Molecular Biology*, pages 109–115, Menlo Park, 1996. AAAI Press.

[11] Paul Horton and Kenta Nakai. Better prediction of protein cellular localization sites with the k nearest neighbors classifier. In *Proceeding of the Fifth International Conference on Intelligent Systems for Molecular Biology*, pages 147–152, Menlo Park, 1997. AAAI Press.

[12] P. Klein, Minoru Kanehisa, and C. DeLisi. The detection and classification of membrane-spanning proteins. *Biochim. Biophys. Acta*, 815:949–951, 1985.

[13] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, 1995.

[14] P. Langley, W. Iba, and K. Thompson. An analysis of bayesian classifiers. In *Proceedings of the tenth National Conference on Artificial Intelligence*, pages 223–228. AAAI Press and MIT Press, 1992.

[15] Richard J. Larsen and Morris L. Marx. *An Introduction to Mathematical Statistics and its Applications*. Prentice-Hall, 1986.

[16] D. J. MeGeoch. On the predictive recognition of signal peptide sequences. *Virus Research*, 3:271–286, 1985.

[17] P. M. Murphy and D. W. Aha. Uci repository of machine learning databases. http://www.ics.uci.edu/ mlearn, 1996.

[18] Kenta Nakai and Minoru Kanehisa. Expert system for predicting protein localization sites in gram-negative bacteria. *PROTEINS: Structure, Function, and Genetics*, 11:95–110, 1991.

[19] Kenta Nakai and Minoru Kanehisa. A knowledge base for predicting protein localization sites in eukaryotic cells. *Genomics*, 14:897–911, 1992.

[20] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[21] S. Salzberg. On comparing classifiers: A critique of current research and methods. http://www.cs.jhu.edu/ salzberg, 1995.

[22] et al. Struyve. Carboxyl-terminal phenylalanine is essential for the correct assembly of a bacterial outer membrane protein. *Journal of Molecular Biology*, 218:141–148, 1991.

[23] Edward C. Uberbacher and Richard J. Mural. Locating protein-coding regions in human dna sequences by a multiple sensor-neural network approach. *Proc. Natl. Acad. Sci., USA*, 88:11261,11265, 1991.

[24] Esko Ukkonen. Constructing suffix trees on-line in linear time. *"Algorithms, Software, Architecture"*, 1:484–492, 1992.

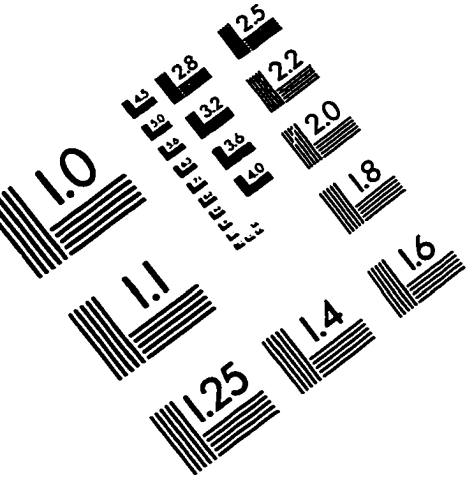
[25] G. von Heijne. A new method for predicting signal sequence cleavage sites. *Nucleic Acids Research*, 14:4683–4690, 1986.

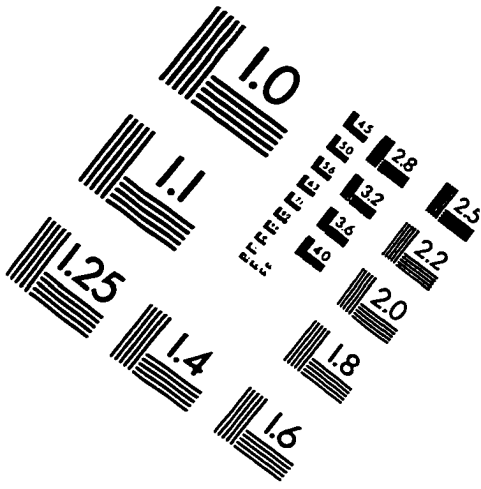
[26] G. von Heijne. The structure of signal peptides from bacterial lipoproteins. *Protein Engineering*, 2:531–534, 1989.

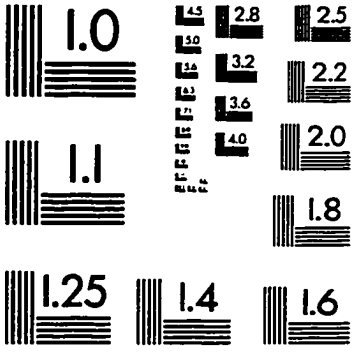
[27] YPD. Yeast protein database. http://www.proteome.com/YPDhome.html, 1997.

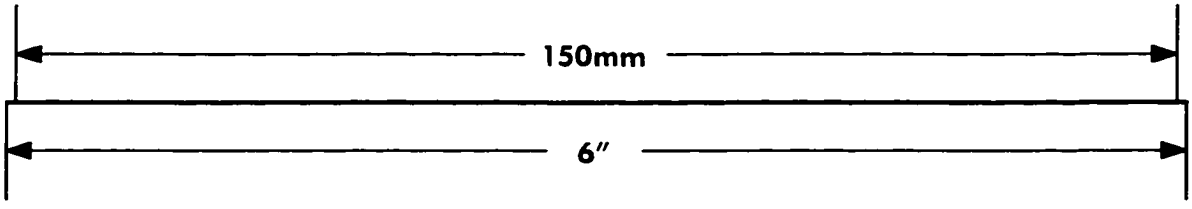
[28] L. Zhao and R. Padmanabhan. Nuclear transport of adenovirus dna polymerase is facilitated by interaction with preterminal protein. *Cell*, 55:1005–1015, 1988.

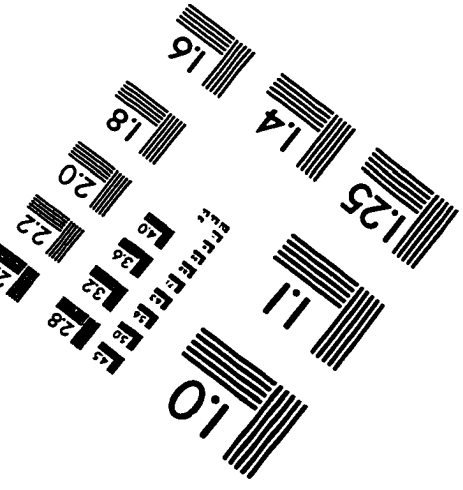
IMAGE EVALUATION
TEST TARGET (QA-3)

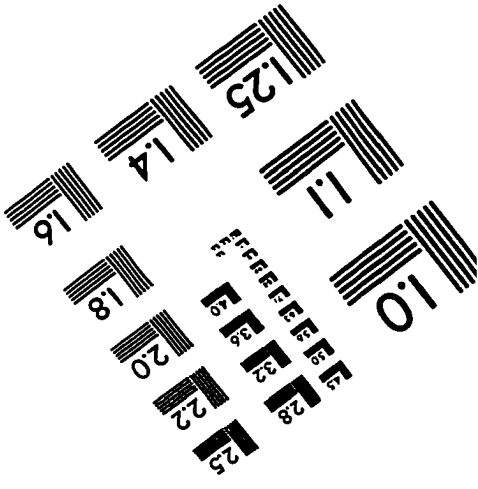












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989