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.

U·M·I

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

Order Number 9208024

A pattern matching system for biosequences

Mehldau, Gerhard, Ph.D.

The University of Arizona, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A PATTERN MATCHING SYSTEM FOR BIOSEQUENCES

by

Gerhard Mehldau

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 1

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Gerhard Mehldau

entitled A PATTERN MATCHING SYSTEM FOR BIOSEQUENCES

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Eugene W. Myers	9/23/91 Date
Ralph E. Griswold	9/23/91 Date
Scott E. Hudson	9/23/91 Date
Robert A. Schowengerdt	9/23/91 Date
Bobby R. Hunt	9/23/91 Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director	9/23/91 Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interest of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ABSTRACT

String pattern matching is an extensively studied area of computer science. Over the past few decades, many important theoretical results have been discovered, and a large number of practical algorithms has been developed for efficiently matching various classes of patterns. A variety of general pattern matching tools and specialized programming languages have been implemented for applications in areas such as lexical analysis, text editing, or database searching. Most recently, the field of molecular biology has been added to the growing list of applications that make use of pattern matching technology.

The requirements of biological pattern matching differ from traditional applications in several ways. First, the amount of data to be processed is very large, and hence highly efficient pattern matching tools are required. Second, the data to be searched is obtained from biological experiments, where error rates of up to 5% are not uncommon. In addition, patterns are often averaged from several, biologically similar sequences. Therefore, to be useful, pattern matching tools must be able to accomodate some notion of approximate matching. Third, formal language notations such as regular expressions, which are commonly used in traditional applications, are insufficient for describing many of the patterns that are of interest to biologists. Hence, any conventional notation must be significantly enhanced to accomodate such patterns. Taken together, these differences combine to render most existing pattern matching tools inadequate, and have created a need for specialized pattern matching systems.

This dissertation presents a pattern matching system that specifically addresses the three issues outlined above. A notation for defining patterns is developed by extending the regular expression syntax in a consistent way. Using this notation, virtually any

pattern of interest to biologists can be expressed in an intuitive and concise manner. The system further incorporates a very flexible notion of approximate pattern matching that unifies most of the previously developed concepts. Last, but not least, the system employs a novel, optimized backtracking algorithm, which enables it to efficiently search even very large databases.

CHAPTER 1: INTRODUCTION

String pattern matching has long been an important area of study in both theoretical and practical computer science. Given a string s of length N and a pattern p of size M, the problem is to locate any (or all) occurrences of p in s. On the theoretical side, a hierarchy of pattern classes has been identified [e.g., Hopcroft and Ullman, 1979], and numerous algorithms have been developed for efficiently matching entire classes or various subsets thereof [e.g., Aho, 1990]. On the practical side, specialized programming languages [Farber *et al.*, 1964; Griswold and Griswold, 1990] as well as general pattern matching tools [Thompson, 1968; McMahon, 1979; Aho *et al.*, 1979] have been implemented. Pattern matching algorithms have been incorporated in application programs in areas as diverse as text editing [Kernighan *et al.*, 1972], database searching [Aho and Corasick, 1975], lexical analysis [Lesk, 1975], and many more.

Most recently, the field of molecular biology has been added to this growing list [Sankoff and Kruskal, 1983]. Molecular biologists obtain nucleic acid (DNA) and amino acid (protein) sequence data from biological experiments. These sequences are represented in digital computers as strings over an alphabet of either four (DNA sequences) or twenty (protein sequences) characters. Biological analysis of this data requires locating all occurrences of certain patterns, or "motifs" (contiguous groups of nucleic acids or proteins that together form a biologically meaningful entity), in the sequence.

The requirements of biological pattern matching differ from most other pattern matching applications in several ways. First, the amount of data to be processed is very large, compared to applications such as text editing or lexical analysis. Until recently, sequences were short enough so that biologists could easily examine the data "by eye" to

determine the presence or absence of a pattern of interest. The situation has changed, however, with recent dramatic improvements to sequencing technology, which have allowed biologists to determine successively longer sequences of nucleotides at an ever increasing rate. In addition, public databases such as GenBank (sponsored by the National Institutes of Health [Burks *et al.*, 1990]), EMBL (maintained by the European Molecular Biology Laboratory, [Kahn and Cameron, 1990]), or PIR (Protein Identification Resource, supported by the National Biomedical Research Foundation [Barker *et al.*, 1990]), have given individual researchers access to the combined sequence data of the entire biological community. Finally, the Human Genome Initiative, a national research effort directed at sequencing the entire human genome, is expected to yield approximately 3 billion nucleotides (3 GigaBytes of biological sequence data) over the next two decades [National Research Council, 1988]. Consequently, any pattern matching tool used by molecular biologists must be highly efficient.

A second important aspect of biological pattern matching is the fact that the data to be searched is often inaccurate, and that the patterns to be found can not be defined exactly. The data is obtained experimentally, and error rates of up to 5% are not uncommon with some of the currently used techniques. The patterns, on the other hand, are frequently averaged from several, biologically similar sequences, or are the result of an "educated guess" on the part of the biologist. The basic pattern matching problem, as defined above, therefore needs to be modified to accomodate this uncertainty. A common formulation for the so-called approximate pattern matching problem is the following. Given a string s of length N and a pattern p of size M, find any (or all) substrings of s that "are not too different" from a string that is matched exactly by p. For example, Figure 1 shows an approximate match between the substring *dbcddaad* of the database s and the string

$w = dbddcabad$, which is an element of the set of strings L that are matched exactly by the regular expression $p = (d|b)^*c(a[bd])^*$.

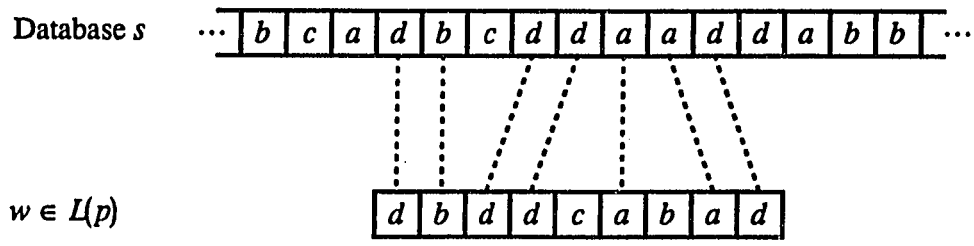

Figure 1: Approximate match

The criteria for determining a good match are still being debated by the biological community, and several schemes for approximate pattern matching have been proposed or implemented [e.g., Sankoff and Kruskal, 1983; Sellers, 1984; Staden, 1984; Myers and Mount, 1986; Gribskov *et al.*, 1987; Mehldau and Myers, 1991]. In order to be useful to molecular biologists, a pattern matching tool must therefore provide some means for accomodating different notions of approximation.

Finally, it is very difficult (or even impossible) to express many biological patterns by using the specification methods that are prevalent in other pattern matching applications. An example of such a pattern is the so-called hairpin pattern (or "stem-loop" structure) shown in Figure 2, which consists of three connected sequences of five to ten nucleotides each. The first and last of those sequences are "inverted repeats" of each other, meaning that the first nucleotide of the first sequence is the "Watson-Crick complement" of the last nucleotide of the last sequence, the second nucleotide of the first sequence is the complement of the second-to-last nucleotide of the last sequence, et cetera. Complementary nucleotides are connected by hydrogen bonds, thus forming the "stem" of the structure. In Figure 2, the chain of nucleic acids is shown as a solid line, individual

nucleotides as small solid circles, and the hydrogen bonds between complementary nucleotides as dashed lines.

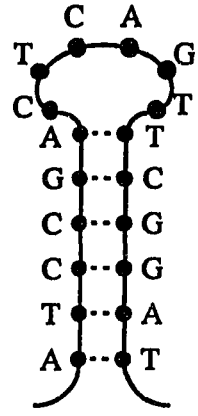

Figure 2: Hairpin pattern

Regular expressions or other formal languages neither provide a notation for defining repeated or cross-bonded elements of a pattern, nor do they support concepts such as inverted repeats that are specific to molecular biology. Other types of patterns that are useful for defining biological structures, but are not found in any of the exisiting general-purpose pattern matching tools, include overlapping motifs and hierarchically defined patterns, as well as specific biological extensions such as consensus sequences or density patterns. As a result of this lack of expressive power, and in order to address particular needs, molecular biologists have resorted to developing highly specific programs that search for particular kinds of biological patterns [e.g., Staden, 1980; Cohen *et al.*, 1983; Saurin and Marliere, 1987; Gautheret *et al.*, 1990]. Clearly, a system that permits the intuitive and concise description of complex biological patterns constitutes a welcome improvement of this situation.

Taken together, all these aspects combine to render most existing pattern matching systems inadequate for biological applications, and have created a genuine need for specialized pattern matching systems. This dissertation presents a practical pattern matching system for applications in molecular biology. The system addresses the three main issues outlined above — efficiency in the face of very large databases, the ability to describe and search for both exact and approximate matches, and the power to express complex biological patterns. The dissertation begins with a survey and discussion of related work — general programming languages, pattern matching languages, general pattern matching tools, and specialized systems for applications in molecular biology, as well as other, less conventional approaches to pattern matching. Next, the design goals and overall structure of the new system are presented, and the rationale behind them is explained. The following chapters discuss the system and its implementation in detail. First, a notation for concisely expressing a large class of patterns is developed, which is then translated into a practical system by defining the syntax and semantics of a new pattern matching language. Subsequent chapters discuss the implementation of the language, describe the overall pattern matching strategy (an optimized backtracking algorithm) in detail, and briefly review the pattern matching algorithms used by the system. Several examples, taken directly from the biological literature, illustrate the power and flexibility of the new approach. The performance of the system is evaluated in a chapter on results, and the dissertation concludes with a discussion of the limitations of the current system, possible improvements, and directions for future research.

CHAPTER 2: RELATED WORK

Much work has been done in the area of string pattern matching. Theoretical computer science (in the area of formal language theory) has identified various classes of patterns, such as regular, context-free, context-sensitive, and recursively enumerable languages. Researchers in the fields of automata theory and algorithms have developed a multitude of algorithms for solving the problem of finding matches to entire classes of patterns and to many subsets that are either important in practice or interesting in theory. Pattern matching algorithms, as they relate to this work, are reviewed and discussed in a subsequent chapter. Practical computer science has contributed general programming languages, pattern matching languages, and a variety of both general and application-specific pattern matching tools. This chapter examines those languages and tools, and assesses their value for pattern matching applications in molecular biology with respect to the criteria established above — the ability to handle large amounts of data efficiently, to search for approximate matches, and to support the concise expression of complex biological patterns.

General Programming Languages

From a computer scientist's point of view, a general, high-level programming language such as C [Kernighan and Ritchie, 1988], Pascal [Wirth, 1971], or even FORTRAN [American National Standard, 1978], is entirely sufficient for string pattern matching. Different languages provide different levels of support for string manipulation and pattern matching tasks, but in each of those languages, programs can be written that search a database for any given pattern from the class of recursively enumerable languages [Hopcroft and Ullman, 1979]. While this approach has the dual advantages of

simplicity and generality, it also has several drawbacks from the viewpoint of a molecular biologist. Biologists are not necessarily familiar with programming languages or efficient pattern matching algorithms, and may not want to invest the time and effort required to learn a new set of skills. These skills, on the other hand, are mandatory for producing software that is both reliable and efficient. Furthermore, writing a pattern matching program with a general programming language is not a one-time effort, since changes to existing programs, if not entirely new ones, are required to support new patterns or classes of patterns. Last, but not least, most biologists are used to expressing patterns in a declarative form, whereas writing a program in a high-level language usually requires a procedural pattern specification. Hence, despite their potential for efficiency and their ability to describe any pattern of interest, general programming languages are not very well suited for immediate use as biological pattern matching tools.

Pattern Matching Languages

Over the past decades, several languages for string and list processing have been developed, with SNOBOL and Icon being the best known among these. SNOBOL [Farber *et al.*, 1964] is an early example of such a language. Its successor, SNOBOL4 [Griswold *et al.*, 1971], provides many of the syntactic elements, control structures, and low-level support (in the form of library functions) required for many pattern matching tasks. SNOBOL4 implements the concepts of back referencing (which is called conditional/immediate value assignment), and backtracking as parts of the language itself. SNOBOL4 permits the declarative description of patterns, and hides most of the matching process from the user. SNOBOL4 is used as a pattern matching tool in areas as diverse as compilers, symbolic mathematics, and linguistics, among others. One of the descendants of SNOBOL, Icon [Griswold and Griswold, 1990], is a widely used language

for string and list processing. Besides modern control structures and built-in support for string scanning, it has several unique concepts such as goal-directed evaluation, generators, and co-expressions. With other general programming languages, Icon shares the drawback of requiring a procedural specification of patterns. In addition, its generality, as well as the fact that it is an interpreted, rather than a compiled, language, make it rather inefficient for performing searches of large databases.

General Pattern Matching Tools

General pattern matching tools have been used with much success for a wide variety of applications. Examples of such programs are the UNIX tools `egrep` (with its cousins `grep` and `fgrep`), `sed`, and `awk`. The `grep` (global regular expression printer) family of commands [Thompson, 1968] searches line-structured files for a user-specified pattern, printing every line containing a match. The most general tool of the `grep` family, `egrep`, permits the specification of regular expressions and supports many common shorthand notations; `grep` places some restrictions on the type of regular expression that it can search for (no Kleene closure, and no alternation) but allows back references, whereas `fgrep` can only handle fixed-string patterns ("keywords"). The `sed` (stream editor) command [McMahon, 1979] copies lines of text from one file to another, in the process editing selected lines according to a user-specified "script," where both the selection criteria and the editing script can involve pattern specifications. The latest and most powerful tool, `awk` [Aho *et al.*, 1979; Aho *et al.*, 1988], is a string scanning and processing language that uses an implicit input loop and a pattern/action paradigm reminiscent of SNOBOL. `Awk` operates on line-structured files, which are searched for occurrences of regular expressions. `Awk` uses a syntax similar to `egrep` and `sed` for pattern specification. For every match found, `awk` performs an associated action (called a

transformation), such as printing the match or reformatting the line containing the match. While all of these tools perform well in their respective domains, they are not very useful for applications in molecular biology, mostly due to their restriction to regular expressions and the absence of support they provide for approximate pattern matching.

The lack of a system for approximate pattern matching has recently been addressed by Wu and Manber [1991], who developed the tool `agrep` (approximate `grep`). Wu and Manber extended an existing algorithm for exact pattern matching [Baeza-Yates and Gonnet, 1989] to accomodate substitutions, insertions, or deletions of characters at a certain cost. A pattern matches a substring of the database under this so-called "edit distance," or Levenshtein measure [Levenshtein, 1966], if the overall cost does not exceed a user-defined threshold. While `agrep` can efficiently search large files for matches to approximate patterns, the algorithm permits only integer costs and a small total cost. In addition, `agrep`, like its cousins, is confined to regular expressions. These restrictions severely limit the tool's usefulness for applications in molecular biology.

Biological Pattern Matching Systems

The inadequacy of general programming languages and pattern matching tools for applications in molecular biology has led biologists to develop several application-specific tools of varying degree of generality and efficiency. These tools can be broadly categorized into programs that search for very specific biological patterns, pattern matching languages capable of searching for relatively large classes of patterns, and systems that address the problem of approximate pattern matching.

One of the earliest systems [Staden, 1980] searches a DNA sequence for a single class of biological patterns, called tRNAs. Characterized by a common, two-dimensional

cloverleaf structure, tRNA genes represent an example of a reasonably complex biological pattern that cannot be described with regular expressions. Staden's program supports approximate matches in the sense that certain, well-defined variations of the basic pattern are permitted, and allows the user to specify a scoring scheme, together with a cut-off threshold, for screening out low-scoring matches. Another early program that searches for a very specific class of patterns is described by Cohen *et al.* [1983]. The program predicts the secondary structure of so-called α/β proteins by locating increasingly fuzzy occurrences of short DNA fragments known as hydrophilic turn segments.

QUEST [Abarbanel *et al.*, 1984], later expanded and renamed PLANS [Cohen *et al.*, 1986], is the first system to use a pattern specification language developed specifically for biological applications. PLANS allows the hierarchical definition of patterns based on regular expression notation. The search algorithm interprets the patterns using finite state automata, with some optimizations to avoid certain worst-case conditions common to biological patterns. PLANS does not allow one to search for approximate matches. The system is implemented in LISP, and no results on its efficiency are reported. Devereux *et al.* [1984] assembled a collection of more than 30 individual programs for matching and handling patterns. Using the "software tools" approach of Kernighan and Plauger [1976], each of these programs performs a simple task, such as finding restriction sites or locating stem-loop structures. Programs communicate with each other by reading from and writing to files. While this approach has the advantage of simplicity and flexibility, it requires the user to run a sequence of programs even for very simple tasks, and to keep track of the results (i.e., files) produced by each of those programs. Saurin and Marliere [1987] describe patterns by constraining certain positions within a pattern through "logical clauses" such as "identical," "complementary," or "rich in." This scheme is very

different from conventional, regular expression-based notations, and despite the biological relevance of some of the clauses ("complement," "helix"), it is not clear how practical it is. The search strategy is implemented as a backtracking algorithm, which locates matches by attempting to satisfy all specified clauses simultaneously. Another program by Staden [1988] permits the definition of patterns consisting of motifs and separation ranges, also called spacers or offsets. The system supports several predefined classes of motifs, such as keywords, sets of keywords, repeats, stem-loop structures, approximate matches, et cetera. However, the way in which patterns are defined (a list of motifs and spacers, combined with the logical operators *and*, *or*, and *not*) severely restricts the way in which these motifs can be combined, and can lead to ambiguous patterns. The search is performed via a backtracking scheme, with different algorithms being used for the various classes of motifs. Gautheret *et al.* [1990] use a similar approach (a list of structural elements, or SEs) to describe the secondary structure of tRNA genes. Again, the search algorithm proceeds by matching individual SEs in a backtracking fashion.

Staden [1984] describes one of the first programs for approximate pattern matching. Matches to a consensus sequence (or "weight matrix") — a "template," derived as a weighted average of several, biologically related sequences — are located by computing a probability score from the weight matrix and the sequence to be examined, and listing all locations for which this score exceeds an automatically determined cut-off threshold. A system by Myers and Mount [1986] permits its users to search a DNA sequence for approximate matches to a fixed-string pattern. The system defines an approximate match as a maximum number of mismatches, insertions, and deletions, either individually or in groups, with respect to the given keyword. The search algorithm is based on the dynamic programming approach [Sellers, 1980], but modified to improve the worst-case running

time. Gribskov *et al.* [1987] use a technique called "profile analysis" for locating matches to consensus sequences. Their method is similar to Staden's, but accounts not only for the substitution of nucleic acids or proteins, but also for insertions and deletions. ANREP [Mehldau and Myers, 1991], a precursor to the current system, was implemented to prototype several of the ideas used in the current system. It is the first system capable of searching for approximate matches to networks (regular expressions without Kleene closure) rather than fixed strings, and permits the specification of arbitrary scoring schemes. The search strategy is a simpler version of the optimized backtracking approach used by the current system.

Some of the biological pattern matching systems presented in this section are capable of efficiently searching for highly specific patterns or classes of patterns. Others permit very general definitions of patterns, but often at the cost of conciseness or efficiency (or both). A third group of programs addresses the approximate pattern matching problem, but limits the class of patterns, or the notion of uncertainty. None of the systems presented above, however, implements the level of both generality and efficiency necessary for many of the pattern matching problems already facing molecular biologists.

Other Approaches to Pattern Matching

The "conventional" techniques described in previous sections are not the only approaches to string pattern matching problems. Researchers in the field of artificial intelligence (AI) have applied machine-learning techniques such as neural networks or rule-based systems to biological pattern matching problems [e.g., Stormo *et al.*, 1982; Lapedes *et al.*, 1989]. However, most of the current effort with respect to AI techniques is

directed towards higher-level types of analyses, such as secondary structure prediction or reconstruction of evolutionary relationships [Hunter, 1991].

Digital neural networks [Rosenblatt, 1962; Minsky and Papert, 1969] attempt to simulate the analog, highly parallel function of biological nervous systems. A digital neural net is an interconnected set of nodes that is characterized by its topology, the type of nodes that are used, and a "training" procedure. The simplest type of digital node consists of a series of analog or binary inputs, which are connected to a single output. Given a set of input values, the node computes the output as a non-linear (sigmoidal) function of the sum of the weighted inputs. Input weights are determined in a training procedure, where a set of sample inputs is given to the network, and the weights, which are initialized to small random numbers, are adjusted until the node yields the desired output for each of the inputs. In a string pattern matching context, the pattern recognition capability of the network is encoded implicitly in the network architecture and in the weights on the connections. The input to the network is a binary encoding of the string to be classified, and the single output of the network is turned on if the string represented by the inputs matches the encoded pattern, and turned off otherwise.

A related machine-learning technique is based on the concept of rule-based ("expert") systems [e.g., Klahr and Waterman, 1986]. Rule-based systems classify their input into one of several categories based on a set of explicitly stated rules. Traditionally, these rules are obtained in a slow and tedious process from human experts. To avoid this so-called "knowledge acquisition bottleneck," Quinlan [1986] devised a method for using a set of training samples to automate this process. Given a representative set of sample inputs together with the desired output (i.e., classification category), the system infers a set of rules and builds a decision tree, which is then used to classify any unknown input.

In a string pattern matching context, the input samples consist of a set of strings, classified into one of two categories, depending on whether or not they contain the desired pattern. The process leads to a decision tree containing rules such as "symbol *a* required in position *i*," or "symbol *b* must not appear after symbol *c* in position *j*."

Machine-learning techniques, such as the ones described above, are most appropriate in situations where a large number of sample patterns is available for training and verification, and a relatively small number of unknown inputs needs to be classified. In the context of molecular biology, these techniques are promising in cases where researchers have identified a number of sequences believed to contain some common feature, but have been unable to extract a specific pattern. In such a situation, rule-based systems are preferable to neural networks, as they allow the biologist to build a pattern specification from the rules — something that is not possible with a neural network. For the type of biological string pattern matching problems considered in this work, however, neither technique is very well suited for several reasons. First, both methods require a substantial effort for training and verification, which needs to be repeated for every single pattern. Second, neither method is capable of efficiently handling large data sets. Finally, while neural nets and rule-based systems do perform some kind of approximate pattern matching, there is no semblance to the model of substitution, insertion, and deletion of single symbols that is commonly used in molecular biology.

CHAPTER 3: DESIGN GOALS

This dissertation presents the design and implementation of an expressive and powerful, yet practical and efficient pattern matching system for applications in molecular biology. This chapter describes the overall structure of the system, and explains and justifies the rationale behind the design decisions. In particular, the notation for describing the class of patterns is discussed, as are the requirements for the pattern specification language, the ideas behind the optimized backtracking algorithm, and the criteria for choosing the pattern matching algorithms used in the implementation.

Besides the general requirements discussed below, a major design goal was the enhancement, consolidation, or unification of useful concepts that are found in isolation in previous work. Specifically, the new system should

- enhance the traditional, regular expression-based pattern matching notation in a way that permits the description of even those biological patterns that until now have required specialized programs. In particular, the system should support the definition of and the search for extended regular expressions, back references, spacers, and approximate matches.
- provide a single, unified concept for approximate pattern matching, based on the edit cost model of symbol substitutions, insertions, and deletions. This concept should accomodate the previously used notions of consensus sequences and weight matrices.
- permit the user to specify positional weights and arbitrary scoring schemes for different portions of a single pattern.

Pattern Notation

Various notations for describing and defining patterns have been developed to date; among them are procedural ones as found in Icon, and declarative ones ranging from the traditional regular expressions used by many UNIX tools to the unique "constraints by logical clauses" found in the pattern matching system of Saurin and Marliere [1987]. While procedural specifications tend to be more flexible and powerful, they also are less intuitive than declarative descriptions. Declarative specifications do not need to be translated into a search process, but correspond directly to the way in which humans think about patterns. The single most important criterion for choosing a pattern notation was the desire to base the pattern notation on an already existing, declarative language, rather than to introduce yet another pattern specification notation. This stipulation required an existing concept that covered a wide range of patterns, and that could be extended in a consistent way.

The considerations above resulted in the decision to base the pattern notation on the concept of regular expressions, which already covers a wide range of frequently occuring patterns. The regular expression notation was extended in a consistent way so as to provide common and useful shorthand notations, as well as a number of enhancements that were considered important by biologists. These enhancements include extended regular expressions for overlapping motifs, back references for repeated and cross-bonded elements, approximate patterns with a definition corresponding to common usage in biological applications (including the notions of consensus sequences and weight matrices), specific biological extensions such as density patterns or the Watson-Crick complement ("inverted repeats"), and, finally, a powerful and general method for describing patterns that can not be expressed with any combination of the above expressions.

Pattern Matching Language

Directly related to the notation used for describing patterns is the design of the pattern matching language. While making the two identical has benefits such as ease of use or consistency, it is generally not possible in practice. Reasons include such real-world constraints as the finite character set of a digital computer, the desire to be able to build an efficient compiler for the language, and other considerations like the user interface, or the need to provide for the input and output of data. It is, however, reasonable to require that the pattern matching language match the formal notation as closely as possible. In addition, it was convenient to build the language on top of an efficient and widely available existing programming language, and to provide facilities for accessing this underlying language. Finally, any new computer language, be it a general-purpose programming language or an application-specific language such as the one presented in this dissertation, should adhere to the guidelines and criteria for good language design that have been developed over the past few decades. These criteria include, among others, uniformity and consistency ("law of least astonishment"), efficiency, and machine independence [Horowitz, 1984].

The pattern matching language chosen for the new system is an enhanced and improved version of the declarative language used in ANREP [Mehldau and Myers, 1991]. The language permits the hierarchical and parameterized specification of patterns, and facilitates libraries of patterns. Patterns are defined in a syntax closely resembling regular expressions, with corresponding enhancements to the pattern notation. The new language is based on (and implemented in) the ANSI standard of the C programming language, and provides access to C through several language features.

Optimized Backtracking

With efficiency being one of the main reasons for developing the new pattern matching system, it was necessary to develop a practical algorithm for efficiently locating matches to patterns in large databases. Given the large class of patterns that can be specified with the notation introduced earlier, it was apparent that no currently existing algorithm could possibly search for an arbitrary pattern. Even if such a generalist algorithm could be designed, it would almost certainly be highly complex and inefficient. Instead, a pattern matching strategy had to be developed that would decompose the overall pattern into smaller portions, search for each of the subpatterns with a specialized algorithm, and combine the results of the individual searches. To make this strategy as efficient as possible, an evaluation algorithm was required that would generate and evaluate a reasonable number of alternatives for decomposing the pattern and ordering the search, and select an optimum among those. The optimization criteria should be based on the idea of searching first for those portions of the pattern that were unlikely to match or inexpensive to search for.

Given the above requirements, an optimized backtracking scheme was the obvious choice. The evaluation algorithm considers a large number, but not necessarily all, possible combinations of pattern decomposition and backtracking order, and selects an optimum among those. Optimization criteria include theoretical considerations (such as the asymptotic time requirement of a given pattern matching algorithm), empirical data (e.g., the performance constants for a particular algorithm, running on a specific machine and operating system), as well as heuristics (such as the underlying strategy for determining the pattern decomposition and backtracking order). This combination of methods has resulted in an algorithm that is a pragmatic blend of theory and practice, and that performs very well in practice.

Pattern Matching Algorithms

The optimized backtracking scheme discussed in the previous section requires specialized algorithms for locating matches to different classes of patterns. A large number of such algorithms are available, and the task of choosing among them is not an easy one. While it may seem desirable to have available a collection of all pattern matching algorithms currently in existence, this goal is certainly not attainable in practice, nor is it necessary for obtaining a well-performing system. Instead, it suffices to assemble a small library of well-chosen algorithms, that, in combination with the backtracking scheme, permit the matching of any given pattern from the class of patterns discussed earlier. A reasonable level of redundancy among the algorithms, however, is useful, if only to provide the optimization scheme with a choice of methods.

Six algorithms were chosen for inclusion in the library. Selection criteria included performance with respect to running time (both in theory and in practice), space demands, and required implementation effort. The library contains a (trivial) algorithm for finding matches to spacers, the well-known algorithms by Boyer and Moore [1977] and Aho and Corasick [1975] for locating keywords and sets of keywords, respectively, and a regular expression algorithm that combines the advantages of the DFA and NFA methods [Aho, 1980]. For finding approximate matches, a regular expression algorithm by Myers and Miller [1989] is included, together with an improved version of this same algorithm that searches only for networks. The latter algorithm is described by Myers and Mehldau [1991].

CHAPTER 4: PATTERN NOTATION

Regular expressions [e.g., Hopcroft and Ullman, 1979] constitute an established way of describing patterns in a declarative manner, and are used in many popular pattern matching tools. They are not, however, sufficient to express the large variety of complex patterns occurring in molecular biology. Therefore, while the pattern matching notation presented below is based on regular expressions, it also extends that notation significantly to accomodate biological patterns. The following sections briefly review regular expressions and define the extensions.

Alphabets

A finite alphabet Σ is defined as a set of characters a_1, ..., a_k. Alphabets are commonly written in the form $[a_1 \ldots a_k]$. For example, the alphabet over DNA sequences and patterns consists of the four characters A, C, G, and T, and is written as [ACGT].

A string s is a finite sequence of characters, $a_1 \ldots a_K$, from an alphabet Σ. The empty string, denoted by ε, is the identity element with respect to string concatenation, and has the property that $s\varepsilon t = st$ for all strings s and t. The length of a string is the number of characters that compose the string; the length of ε is zero. A pattern p denotes a set of strings.

Regular Expressions

Regular expressions over an alphabet Σ, and the strings which they match, are defined recursively as follows.

- ε is a regular expression, and it matches the empty string.

- For all $a \in \Sigma$, a is a regular expression, and it matches the string a.
- The alternation of two regular expressions p and q, denoted by $p \mid q$, matches the union of the sets of strings matched by p and q.
- The concatenation of two regular expressions p and q, written as $p \cdot q$, or pq, matches all strings consisting of a string matched by p, immediately followed by a string matched by q.
- The Kleene closure of a regular expression p, denoted by p^*, matches all strings consisting of any number (including zero) of concatenations of strings matched by p.
- A regular expression in parentheses, (p), matches the same set of strings as p.

Parentheses are useful for changing the order of precedence, which is defined as Kleene closure (highest), followed by concatenation and alternation.

The pattern $a(b|c)d^*$ is an example of a regular expression; it is equivalent to the fully parenthesized pattern $(((a)((b)|(c)))((d)^*))$, and matches the strings *ab*, *ac*, *abd*, *acd*, *abdd*, *acdd*, ... et cetera, ad infinitum.

Shorthand Notations

Several common shorthand notations are convenient for expressing certain frequently occurring patterns, and are used in many well-known pattern matching systems. With the exception of negative-length spacers, it is always possible to write equivalent patterns, using only the basic regular expression notation defined above.

- The quantified Kleene closure of a regular expression, written as $p^{*m,n}$, or $p^{m,n}$, $0 \le m \le n$, matches all strings consisting of the concatenation of at least m and at most n strings matched by p.

- An optional regular expression, denoted by $p?$, matches the empty string as well as all strings matched by p.
- A character class c, written as $[a_1 \ldots a_k]$, matches any character $a_i \in c$.
- The wildcard symbol, denoted by a . (period), matches any character $a \in \Sigma$.
- A spacer of the form $<m,n>$, $m \leq n$, matches any string consisting of at least m and at most n characters $a \in \Sigma$. For positive m,n, the pattern $<m,n>$ is equivalent to the pattern $.^{m,n}$. Characters are matched "to the left" for $m, n < 0$, and "to the right" for $m, n > 0$.

An example of a shorthand notation is the pattern $a^{2,3}b?[cd]$, which is equivalent to the regular expression $aa(a|\varepsilon)(b|\varepsilon)(c|d)$, and matches the strings *aaabc*, *aaabd*, *aaac*, *aaad*, *aabc*, *aabd*, *aac*, and *aad*. On the other hand, the pattern $a<-2,0>b$, which matches the strings *ab* and *ba*, can not be written as regular expression.

Extended Regular Expressions

Two extensions to the class of regular expressions, the conjunction and difference operators, are useful for describing overlapping patterns.

- The conjunction of two regular expressions p and q, denoted by $p \,\&\, q$, matches any string that is matched by both p and q.
- The difference of two regular expressions p and q, written as $p - q$, matches any string that is matched by p, but not matched by q.

For example, assuming $\Sigma = [abcd]$, the pattern $ab.[cd] \,\&\, [ab]^{2,3}d$ matches the strings *abad* and *abbd*, whereas the pattern $ab.[cd] - [ab]^{2,3}d$ matches the strings *abac*, *abbc*, *abcc*, *abdc*, *abcd*, and *abdd*.

Back References

For describing relationships between elements of a pattern, it is useful to be able to refer to a previously matched portion of a pattern. This can be done by attaching a "back reference" attribute to the portion of the pattern that is to be referred to. The meaning of this construct is given by the following procedural definition.

- If a pattern contains two or more occurrences of some subpattern p that is marked as a back reference, then a single, but otherwise arbitrary, occurrence of p is considered "free" (written as p^F), while all other occurrences are "bound" (denoted by p^B). The free occurrence p^F can match any string matched by p; however, all bound occurrences p^B must match whatever string was matched by p^F.

If the pattern $p = a|b$ is designated as a back reference, then the pattern $q = p \cdot p$ matches the strings *aa* and *bb*, but not the strings *ab* or *ba*.

Back references are useful, and in some cases necessary, to describe biological patterns with repeated or cross-bonded elements. Back references extend the class of patterns that must be recognized from the regular languages to the context-sensitive languages.

Approximate Patterns

The notion of "approximate pattern" used by the system is based on the model of single-point mutations commonly used in molecular biology [Sellers, 1974]. This model assumes that changes in biological sequences occur as insertions, deletions, or substitutions ("edit operations") of single nucleotides. This biological concept is modeled by pattern matching algorithms with the notion of an "alignment," where the pattern is

altered ("edited") to match some substring of the database. Formally, an alignment is defined as follows.

An alignment L between a pattern p and a substring s of the database is defined as a sequence of pairs $\begin{bmatrix} a \\ b \end{bmatrix}$, $a, b \in \Sigma \cup \{\varepsilon\}$, where the concatenation of the upper elements of L spells a string matched by p, and the concatenation of the lower elements of L spells s. A pair $\begin{bmatrix} \varepsilon \\ b \end{bmatrix}$ is called an insertion pair, a pair $\begin{bmatrix} a \\ \varepsilon \end{bmatrix}$ a deletion pair, and all other pairs are called substitution pairs. Associated with each pair $\begin{bmatrix} a \\ b \end{bmatrix}$ is a score $\sigma(a,b)$. The score of the alignment L is defined as the sum, over all pairs that constitute L, of the score of each pair ("column-sum" cost model, [Sankoff and Kruskal, 1983]). An optimum alignment between p and s is one that has maximum score.

Alignment scores σ are defined through scoring schemes. A scoring scheme is a $(|\Sigma|+1)$-by-$(|\Sigma|+1)$ matrix σ, which defines the score of edit operations for all $a, b \in \Sigma$ and ε. Specifically, the score of substituting b for a is $\sigma(a,b)$, the score of deleting a is $\sigma(a,\varepsilon)$, and the score of inserting b is $\sigma(\varepsilon,b)$. Figure 3 shows a unitary scoring scheme over the alphabet [*abcd*], where insertions and deletions of characters are scored $-\infty$, substitutions of identical characters (i.e., matches) are scored one, and substitutions of different characters (i.e., mismatches) are scored zero. This scoring scheme effectively prohibits insertions and deletions, since no alignment containing either type of edit operation can ever be optimal.

	ε	*a*	*b*	*c*	*d*
ε		−∞	−∞	−∞	−∞
a	−∞	1	0	0	0
b	−∞	0	1	0	0
c	−∞	0	0	1	0
d	−∞	0	0	0	1

Figure 3: Unitary scoring scheme

In general, each symbol x of a pattern can be thought of as having associated with it two *score vectors*, which define the scores for inserting a symbol after x, deleting x, and substituting another symbol for x. For example, the score vectors for symbol c are the boxed rows of the matrix in Figure 3. Normally, these vectors are derived from the scoring scheme (as in the example), but in general, they can be defined arbitrarily.

Using the definitions above, approximate patterns are defined as follows.

- An approximate pattern, written as p % t, matches any string s for which the score of the highest-scoring alignment between p and s equals or exceeds the threshold t.

Since absolute scores would favor longer matches over shorter ones, all scores are normalized with respect to the length of the matched pattern [Sellers, 1984], i.e., comparison is made between the threshold t and the score of an alignment divided by the length of the matched pattern.

For example, assuming the scoring scheme from Figure 3, the pattern p = *abacd*%0.8 matches any five-character string, where at least four characters agree with the pattern in both position and value. For example, the optimum alignment between the pattern p and

the string *abaad*, written as $\left[\begin{matrix} a \\ a \end{matrix}\right]\left[\begin{matrix} b \\ b \end{matrix}\right]\left[\begin{matrix} a \\ a \end{matrix}\right]\left[\begin{matrix} c \\ a \end{matrix}\right]\left[\begin{matrix} d \\ d \end{matrix}\right]$, yields an absolute score of 4, which translates into a length-relative score of 0.8.

For simple approximate patterns, a single default scoring scheme is usually sufficient. For more complex patterns, however, it is often useful to be able to override such a default for either a portion of a pattern, or for an entire pattern. Similarly, it may be useful to emphasize some sections of a pattern, or to suppress others. This can be accomplished with the following two notations.

- A pattern with an associated scoring scheme, denoted by p ! s, uses the specified scoring scheme s when computing alignments.
- A pattern with an associated positional weight, written as p : w, causes the score of any alignment for p to be multiplied by w.

Consensus Symbols

Consensus sequences, also known as "weight matrices" or "matrix patterns," are an important pattern matching concept in molecular biology. The idea behind consensus sequences is to extract, from a set of known sequences, a consensus that emphasizes the similarities between the sequences and suppresses the differences. This is usually accomplished by specifying, for a given position of the sequence, different scores for matching different characters [Stormo, 1990]. Hence, consensus sequences can be considered a special case of approximate patterns and can benefit from the use of the concepts introduced earlier.

To capture the notion of consensus sequences, the concept of character classes has been generalized to consensus symbols. Consensus symbols are artificial symbols with

explicitly defined score vectors. Consensus symbols can be combined to consensus sequences or consensus patterns by using the various pattern construction operators introduced earlier in this chapter.

A consensus symbol consists of a reduction function f (such as *min*(), *avg*(), or *max*()), and any number of symbols $a_i \in \Sigma \cup \{+,-,\wedge\}$, with an associated weight w_i. The score vectors of the consensus symbol are computed as the f-reduction of the score vectors of all the symbols involved in the definition of the consensus symbol, with all elements of the vectors corresponding to symbol a_i multiplied by the weight w_i. Insertion and deletion scores are normally excluded from the reduction, unless the definition of the consensus symbol contains the special symbols "+" and "–", respectively. Similarly, all symbols $a_i \in \Sigma$ that are not listed in the definition of the consensus symbol, can be included in the computation with the special symbol "^". Any score vector entry that is not explicitly defined defaults to $-\infty$.

- A consensus symbol, written as $[f(a_i{:}w_i)]$, $a_i \in \Sigma \cup \{+,-,\wedge\}$, describes an artificial symbol that is part of an approximate pattern. The score vectors, computed as described above, define the scores for inserting a symbol after the consensus symbol, deleting the consensus symbol, and substituting another symbol for the consensus symbol

Figure 4 shows how the score vectors for the consensus symbol [*avg*(*a*:1*c*:2)] are computed as the weighted average of the score vectors of the two symbols *a* and *c*. Figure 4 assumes the unitary scoring scheme from Figure 3. Entries that are not included in the reduction are labeled with a question mark.

	ε	*a*	*b*	*c*	*d*	ε	*a*	*b*	*c*	*d*
a		?	?	?	?	?	1.0	0.0	0.0	0.0
2*c*		?	?	?	?	?	0.0	0.0	2.0	0.0
[*avg*(*a*:1*c*:2)]		−∞	−∞	−∞	−∞	−∞	0.5	0.0	1.0	0.0

Figure 4: Score vectors for symbol [*avg*(*a*:1*c*:2)]

A special consensus function, *def*(), is provided for directly defining individual entries of a score vector. In this case, the value w_i associated with a symbol a_i represents the insertion, deletion, or substitution score itself, rather than a weight to be used in a reduction.

Biological Extensions

In addition to the shorthand notations described above, there are several constructs that occur frequently enough in biological patterns to warrant their own abbreviations.

- A pattern of the form $p@q$, called a list, matches all strings consisting of the concatenation of strings from p, separated by strings from q. The quantized version of this pattern, $p@^{m,n}q$, $0 \le m \le n$, matches all lists consisting of at least m and at most n strings matched by p.
- The Watson-Crick complement of a pattern p, denoted by $\sim p$, matches the "reversed complement," or "inverted repeat," of p. The inverted repeat is defined only for patterns over nucleic acid (DNA) alphabets, and matches, for each string s matched by p, the reverse of s, with T substituted for A, G for C, C for G, and A for T. For example, the pattern ~TTCA matches the string TGAA.
- A density pattern, written as $\{n_1a_1 \ldots n_ka_k\}$, where $1 \le n_i$ and $a_i \in \Sigma$, matches all strings containing exactly n_1 characters a_1, ..., and n_k characters a_k, in any order.

Again, the above notations are only abbreviations for larger and more complex regular expressions. For example, the pattern $a\{2b|c\}d$ is equivalent to the regular expression $(abbcd \mid abcbd \mid acbbd)$, while the pattern $a@^{0,2}c$ could also be written as $(\varepsilon \mid a \mid aca)$.

Conditional Patterns

In some cases, it is not convenient or even possible to express a biological pattern with only the notations above. For those cases, a concept has been added that permits the description of virtually any pattern.

- A conditional pattern, denoted by $p \setminus e$, matches any string matched by p for which the boolean expression e is true.

For example, if the pattern p is defined as $p = a^*$, then the pattern $p \setminus |p| \geq 5$ matches the strings *aaaaa*, *aaaaaa*, *aaaaaaa*, ... et cetera, ad infinitum.

CHAPTER 5: LANGUAGE DEFINITION

The notation presented in the previous chapter has been implemented in the form of a declarative computer language, PAMALA (PAttern MAtching LAnguage), for specifying patterns and performing database searches. The language is free-format, strongly typed, permits the hierarchical and parameterized definition of patterns, and supports libraries of predefined patterns.

This chapter defines the syntax of the language using a modified BNF grammar, and describes the semantic meaning of each construct. The following notation is used. The syntactic element being defined is listed, followed by a "→" and its definition. Syntactic elements listed on a single line need to appear in the same order in the program; alternatives are separated by "|". **Bold** type indicates keywords that appear literally in the program. Terminals are printed in `Courier` typeface, and regular expressions of terminals are shown as quoted strings. Non-terminals are printed in *italic* type. A "*" after a syntactic element indicates zero or more occurrences of that element, and a "+" means one or more occurrences. Syntactic elements in square brackets are optional, i.e., they can appear zero or one times. A type associated with an identifier in the grammar is written as a subscript on the identifier. If the identifier appears on the left-hand side of an equal sign, the type is assigned to the identifier; otherwise, the type is required of the identifier. Annotations to the grammar are introduced by a "#", and extend until the end-of-line.

A PAMALA program consists of a sequence of declarations and executable statements, each of which is terminated by a semicolon.

Prog → ((*DeclStmt* | *ExecStmt*) ;)*

DeclStmt	→	*NumDecl*
	|	*AlpDecl*
	|	*ScoreDecl*
	|	*SymDecl*
	|	*PatDecl*
	|	*RefDecl*
	|	*FunDecl*
ExecStmt	→	*Include*
	|	*Evaluate*
	|	*Terminate*

White space (blanks, tabs, newlines, and comments) between tokens is insignificant; however, white space must not be part of any token. The syntax for comments follows the C notation; comments begin with the string "/*" and are terminated by the string "*/". Comments do not nest.

Variables

Variable names (identifiers) must begin with a letter, and may be followed by any number of letters, digits, and underscores. Variable names are case-sensitive and may be of arbitrary length, with all characters being significant. Associated with each variable is a type. All variables must be declared before they can be used.

Id	→	"[a-zA-Z][a-zA-Z_0-9]*"

Numbers

Numbers in PAMALA are either integer or floating point quantities. Numeric variables are declared with the **`int`** and **`float`** statements, resulting in variables of type "int" and "float", respectively. A numeric variable may be initialized with either a constant or with the value of a previously declared variable. If the initialization is omitted, the initial value of a numeric variable defaults to zero. PAMALA variables of type "int" or "float" are equivalent to global C variables of the same type, and may be used as such within C code that is part of the PAMALA program.

NumDecl → **`int`** Id_{int} [= *INum*]
| **`float`** Id_{float} [= (*INum* | *FNum*)]

INum → Id_{int}
| *ICon*

ICon → "[+-]"
| *Int*

Int → ["[+-]"] "[0-9]+"

FNum → Id_{float}
| *FCon*

FCon → *Int* "[eE]" *Int*
| *Flp* ["[eE]" *Int*]

Flp → ["[+-]"] "([0-9]+.[0-9]* | .[0-9]+)"

Integer constants consist of an optional sign, followed by a sequence of decimal digits. Integer constants must not contain a decimal point or exponent. As a special case, "+" and "–" are interpreted as shorthand for +1 and –1, respectively. Floating point constants contain a decimal point or an exponent, or both.

Alphabets

Alphabets are sets of characters. Variables of type "alpha" are declared with the **alpha** declaration statement; they are used in connection with the declaration of scoring schemes (see below).

AlpDecl	→	**alpha** $Id_{\text{alpha}} = Alp$
Alp	→	Id_{alpha}
	|	[*Atom*+]
Atom	→	"[a-zA-Z] | \."

Atoms are either upper- or lowercase letters, or any other ASCII character (written as a back slash, followed by the character itself).

Scoring Schemes

The **score** statement is used to declare both an alphabet (over which all patterns are defined), and a corresponding scoring scheme (which is required only for approximate patterns). PAMALA contains the concept of a default scoring scheme (and, by implication, a default alphabet), which applies to all patterns, unless a different scoring scheme is explicitly specified. The default scoring scheme is initially undefined and

hence must be established by the user before any patterns can be declared. Scoring schemes are defined or made the default with the **score** declaration statement.

With the first form of the **score** declaration statement, a variable of type "score" is declared, initialized according to the specification on the right-hand side of the equal sign, and made the default scoring scheme. If the PAMALA program does not contain approximate patterns, a scoring scheme is not needed, and the initialization inside the curly braces may be omitted; however, both the alphabet and the braces themselves are required. The second form of the **score** declaration statement simply sets the default scoring scheme to the one specified by the identifier.

ScoreDecl → **score** Id_{score} = *Alp* { (*Score* ;)* }
| **score** Id_{score}

The following two formats are provided for initializing scoring schemes; both formats may be mixed freely within the same **score** declaration statement.

Score → < *Set* ["[, .]" *Set*] > # (*INum* | *FNum*)
| *Expression* # *Expression*

Set → $
| *Atom*
| [($ | *Atom*)+]

Expression → *expression*
as defined in Kernighan and Ritchie [1988]

The initialization declaration is processed as follows. First, all entries of the scoring matrix σ are set to $-\infty$. Initialization statements are then executed sequentially, in the

order in which they are listed. If any matrix entry is set by more than one initialization statement, the value assigned by the last such statement prevails.

The first (or enumerative) format explicitly lists the entries to be set. A declaration of this form is interpreted as follows. Let set_1 be the first set of symbols listed inside the angle brackets, and set_2 be the second set of symbols (the dollar sign is a special symbol that stands for ε). If set_2 is not present in the declaration, it is assumed to be Σ. Let *op* be either "," or "." (assumed to be "." if not present), and *value* the value of the numeric constant to the right of the pound sign. Then, if *op* = ",", for all $a \in set_1$ and all $b \in set_2$, $\sigma(a,b)$ is set to *value*; if *op* = ".", both $\sigma(a,b)$ and $\sigma(b,a)$ are set to *value*. For example, the score declaration statement in Figure 5 defines the hybridization matrix shown in Figure 6.

```
score hybrid = [ACGT] { <{ACGT]>  #  -2;
                        <C.G>     #   3;
                        <A.T>     #   2;
                        <G.T>     #   1;
                      };
```

Figure 5: Hybridization scoring scheme declaration

	ε	A	C	G	T
ε		−∞	−∞	−∞	−∞
A	−∞	−2	−2	−2	2
C	−∞	−2	−2	3	−2
G	−∞	−2	3	−2	1
T	−∞	2	−2	1	−2

Figure 6: Hybridization scoring scheme

The second (or functional) format uses boolean conditions to specify the entries to be set. The C expression to the left of the pound sign is interpreted as a boolean expression in variables x and y, where x represents the row (first) index and y the column (second) index into matrix σ. The boolean expression is then applied to each element of the matrix σ. Each entry for which the expression evaluates to true (i.e., is not equal to zero), is set to the value of the C expression to the right of the pound sign, which also may be a function of x and y. For example, the score declaration statement in Figure 7 defines the unitary scoring scheme shown in Figure 8.

```
score uni = [ACGT] { x != '$'  &&  y != '$'   #   x == y; };
```

Figure 7: Unitary scoring scheme declaration

	ε	A	C	G	T
ε		$-\infty$	$-\infty$	$-\infty$	$-\infty$
A	$-\infty$	1	0	0	0
C	$-\infty$	0	1	0	0
G	$-\infty$	0	0	1	0
T	$-\infty$	0	0	0	1

Figure 8: Unitary scoring scheme

Symbols

Symbols are a generalization of characters to classes of characters and consensus symbols. Variables of type "sym" are declared with the **`symbol`** declaration statement.

SymDecl → **`symbol`** Id_{sym} = *Sym*

Sym → .
| Id_{sym}
| " *SCon* "

A symbol constant may be either the special wildcard symbol "." (which matches any character from the default alphabet, but has no effect on the score, if it is part of an approximate pattern), a single character (atom), a class of characters, or a consensus symbol.

SCon → .
| *Atom*
| *Class*
| *Cons*

Character classes are defined in a manner similar to alphabets; they match any of the characters in the class.

Class → [*Atom*+]

A consensus symbol represents an artificial symbol, whose score vectors are computed from the literals and weights involved in its definiton. Consensus symbols must be part of an approximate pattern, and are specified in a syntactic form similar to character classes.

Cons → ["[>~<=]" (*Lit* + | (*Lit*+ : (*INum* | *FNum*))+ [*Lit*+])]

Lit → *Atom*
| +

| –
| ^

The function symbol immediately following the left square bracket determines how the score vectors are computed from the literals and their weights. A ">" stands for a maximum reduction, a "~" for an average reduction, a "<" for a minimum reduction, and a "=" (a "tailored" consensus symbol) means that the specification is to be taken literally — no reduction function is applied. Following the function symbol, the literals involved in the definition of the consensus symbols are listed in groups. A literal may be either a single atom, or it can be one of the three special sysmbols "+" (which stands for insertion), "–" (deletion), and "^" (all characters from the default alphabet not explicitly listed in the declaration of the consensus symbol). Each group is followed by a colon and a weight, which applies to each of the literals in the group. The colon and the weight may be omitted for the last group, in which case a default weight of 1.0 is used.

A consensus symbol of the form [> $a_1 \ldots a_n$] matches the same symbols as a character class of the form [$a_1 \ldots a_n$]; however, the latter may be used with any pattern, whereas the former must be part of an approximate pattern.

The computation of the score vectors for a consensus symbol α can be described by defining sets $A = \{l_i{:}w_{l_i}\}$ (where l_i is a literal from the declaration, and w_{l_i} is its weight) and $C = A - \{+,-,\hat{}\}$ (i.e., the set of all regular characters in the specification). Substitution, deletion, and insertion scores are then defined for the case of a reduction function f according to Figure 9; the definition of the score vectors for the case of a tailored consensus symbol is given in Figure 10.

$$\sigma(\alpha,c) = \begin{cases} f\left(\underset{x \in C}{f}\left(\sigma(x,c)\cdot w_x\right), \underset{x \in \Sigma-C}{f}\left(\sigma(x,c)\cdot w_\wedge\right)\right) & \text{if } C \neq \varnothing \wedge \wedge \in A \\ \underset{x \in C}{f}\left(\sigma(x,c)\cdot w_x\right) & \text{if } C \neq \varnothing \wedge \wedge \notin A \\ \underset{x \in \Sigma-C}{f}\left(\sigma(x,c)\cdot w_\wedge\right) & \text{if } C = \varnothing \wedge \wedge \in A \\ -\infty & \text{otherwise} \end{cases}$$

$$\sigma(\alpha,\varepsilon) = \begin{cases} f\left(\underset{x \in C}{f}\left(\sigma(x,\varepsilon)\cdot w_x\right), \underset{x \in \Sigma-C}{f}\left(\sigma(x,\varepsilon)\cdot w_\wedge\right)\right)\cdot w_- & \text{if } - \in A \wedge \wedge \in A \\ \underset{x \in C}{f}\left(\sigma(x,\varepsilon)\cdot w_x\right)\cdot w_- & \text{if } - \in A \wedge \wedge \notin A \\ -\infty & \text{otherwise} \end{cases}$$

$$\sigma(\varepsilon,c) = \begin{cases} \sigma(\varepsilon,c)\cdot w_+ & \text{if } + \in A \\ -\infty & \text{otherwise} \end{cases}$$

Figure 9: Score vectors for consensus symbol with reduction function f

$$\sigma(\alpha,c) = \begin{cases} w_c & \text{if } c \in C \\ w_\wedge & \text{if } c \notin C \wedge \wedge \in A \\ -\infty & \text{otherwise} \end{cases}$$

$$\sigma(\alpha,\varepsilon) = \begin{cases} w_- & \text{if } - \in A \\ -\infty & \text{otherwise} \end{cases}$$

$$\sigma(\varepsilon,c) = \begin{cases} w_+ & \text{if } + \in A \\ -\infty & \text{otherwise} \end{cases}$$

Figure 10: Score vectors for tailored consensus symbol

Patterns

Patterns are specified by combining pattern literals (the wildcard symbol, spacers, and strings of symbols) with variables of types "sym", "pat", and "ref", using the various pattern construction operators listed below. Variables of type "pat" are declared with the **pattern** declaration statement, and may be parameterized.

PatDecl → **pattern** Id_{pat} [{ *Id* (, *Id*)* }] = *Pat*

With the exception of the complement operator (which groups from right to left), all pattern construction operators associate from left to right. Operators are listed in order of increasing precedence.

Pat	→	*Pat* \ (*Expression*)	# Conditional
	|	*Pat* | *Pat*	# Disjunction
	|	*Pat* & *Pat*	# Conjunction
	|	*Pat* - *Pat*	# Difference
	|	*Pat Pat*	# Concatenation
	|	*Pat* @ [^ *INum* [, *INum*]] *Pat*	# List
	|	*Pat* * [^ *INum* [, *INum*]]	# Kleene closure
	|	*Pat* ?	# Optionality
	|	*Pat* ! Id_{score}	# Scoring scheme
	|	*Pat* : (*INum* | *FNum*)	# Positional weight
	|	*Pat* % (*INum* | *FNum*)	# Approximation
	|	~ *Pat*	# Complement
	|	(*Pat*)	# Grouping
	|	Id_{pat} [{ *PatArg* (, *PatArg*)* }]	# Pattern variable
	|	Id_{ref}	# Reference variable

|	Id_{sym}	# Symbol variable
|	{ [*INum*] *Sym* (, [*INum*] *Sym*)* }	# Density
|	< *INum* [, *INum*] >	# Spacer
|	" *SCon+* "	# Symbol literal(s)
|	.	# Wildcard symbol

The various pattern construction operators have the meaning described in the previous chapter. In addition, the following restrictions and limitations apply.

The boolean expression on the right-hand side of a conditional pattern is specified in the form of an integral C expression, with a value of zero corresponding to *false*, and all other values being equivalent to *true*.

Expression → *expression*
as defined in Kernighan and Ritchie [1988]

The C expression may reference and assign to PAMALA variables of types "int" and "float", and may contain calls to PAMALA variables of type "fun". The expression may also reference PAMALA variables of type "ref", *but only if at least one occurence of each referenced variable is contained within the left-hand side of the conditional pattern.*[1] Furthermore, the expression must not assign values to such variables.

The right-hand side of a difference pattern must not contain any back reference patterns.

---

1 See the chapter on pattern evaluation for an explanation of this restriction.

In patterns containing quantified lists or Kleene closures, the minimum number of repeats must be equal to or greater than zero, and the maximum must be equal to or greater than one. The numbers may be given in any order.

Patterns with an associated scoring scheme must be part of an approximate pattern. If multiple scoring schemes are specified for the same pattern, the first (innermost) scoring scheme is used.

Patterns with an associated positional weight must be part of an approximate pattern. Nested positional weights accumulate multiplicatively.

Approximate patterns must not contain conditionals, conjunctions, differences, or spacers of negative length, and approximate patterns must not be nested. Approximate patterns may not contain any sub-patterns that are designated as back references.

The Watson-Crick complement is defined only for patterns over the nucleic acid (DNA) alphabet.

If a referenced variable of type "pat" was parameterized, values must be provided for all arguments. Parameters may be numbers, variables of type "score", and other patterns.

PatArg	→	*INum*
	|	*FNum*
	|	Id_{score}
	|	*Sym*
	|	*Pat*

Back References

Patterns with an associated "back reference" attribute are declared with the **reference** declaration statement. The declaration of the pattern itself is identical to the declaration of general patterns above, except that variables of type "ref" can not be parameterized.

RefDecl → **reference** Id_{ref} = *Pat*

PAMALA variables of type "ref" are equivalent to global, *read-only* C variables of type "char *", and may be used as such within C code that is part of the PAMALA program.

Functions

Functions control the execution of a PAMALA program in conjunction with the **evaluate** statement, which is described below.

PAMALA provides two built-in functions, `matches()` and `contains()`, which search either a string or a database for a user-specified pattern. The first three parameters are identical for both functions. The first parameter is the pattern to be searched for; it must be a PAMALA *variable* of type "sym", "pat", or "ref" — it can not be a pattern literal.[2] The second parameter is either a string to be searched (in which case the third parameter must be the keyword `STRING`), or it is the name of a file containing the database (in which case the third parameter is a keyword indicating the format of the database). The function `contains()` has an additional parameter, which indicates whether the function is to search for just one, or all, occurrences of the given pattern. The

---

2 See the chapter on system implementation for an explanation of this restriction.

function `matches()` returns a value of one if the pattern matches the database, and zero otherwise; the function `contains()` returns the number of matches found. When called directly from an **`evaluate`** statement (as opposed to from within a user-defined function), both functions output any matched strings, together with their positions in the database. The interface is summarized in Figure 11.

```
typedef struct { ..... } pattern;
typedef enum {STRING, PLAIN, PIR} database;
typedef enum {ANY, ALL} number;
int matches  (pattern, char *, database);
int contains (pattern, char *, database, number);
```

Figure 11: Search function interfaces

Both functions may be used either directly with the **`evaluate`** statement, or may be called from within user-defined functions. However, the data types defined in Figure 11 are restricted to the functions `matches()` and `contains()`; they are not available within user-defined functions.

In addition to the two system-provided functions defined above, PAMALA users can define their own functions with the **`function`** declaration statement. The syntax of this statement is virtually identical to that of an ANSI C function declaration. The only differences are the initial **`function`** keyword, and the equal sign between the argument list and the opening curly brace — both of which represent "syntactic sugar" that was added to make the PAMALA syntax more consistent. In addition, the function declaration — like all PAMALA statements — is terminated by a semicolon.

FunDecl → **`function`** [*Specifiers*] *Declarator* = *Statements*

Specifiers → *declaration-specifiers*
as defined in Kernighan and Ritchie [1988], except
that function names must not begin with an underscore

Declarator → *declarator*
as defined in Kernighan and Ritchie [1988]

Statements → *compound-statement*
as defined in Kernighan and Ritchie [1988]

PAMALA variables of type "fun" are equivalent to global C functions, and may be used as such within C code that is part of the PAMALA program.

Functions can be used in conjunction with conditional patterns, either to facilitate the hierarchical definition of patterns, or simply to encapsulate lengthy or complex conditions. Functions are also useful for the interactive input of parameters, for special purposes such as accumulating statistics, or for creating customized output. Last, but not least, by providing an "escape" to the power of the underlying C programming language, functions permit the implementation of any desired functionality that is not directly supported by PAMALA.

File Inclusion

The **`include`** statement is provided to support libraries of predefined patterns. An **`include`** statement is processed by opening the file specified via the file identifier, reading and processing the contents of the file, and closing the file. Include statements may be nested.

Include → **`#include`** *FileId*

FileId → "[^;]+"

Function Evaluation

This statement evaluates the function whose name is specified immediately after the keyword **`evaluate`**. The function result, if any, is discarded. If the function was parameterized, values must be provided for all arguments. In keeping with C syntax, the parentheses after the function name are required, even for functions without arguments. Parameters may be numbers of type "int" or "float", and variables of type "ref" (which must correspond to arguments of the C type "char *"), and "fun".

Evaluate → **`evaluate`** Id_{fun} ([*FunArg* (, *FunArg*)*])

FunArg → *INum*
| *FNum*
| Id_{ref}
| Id_{fun} ([*FunArg* (, *FunArg*)*])
| *character-constant*
as defined in Kernighan and Ritchie [1988], except
that the wide-character prefix 'L' is not recognized
| *string-literal*
as defined in Kernighan and Ritchie [1988], except
that the wide-character prefix 'L' is not recognized

Program Termination

The **quit** and **exit** statements terminate a PAMALA program. Both forms are equivalent to an end-of-file.

Terminate → **exit**
| **quit**

CHAPTER 6: SYSTEM IMPLEMENTATION

A general PAMALA program consists of a sequence of declarations and executable statements. Some of the declarations may contain embedded C code in the form of scoring scheme initializations, boolean expressions that are part of conditional patterns, and user-defined functions. Furthermore, PAMALA variables of types "int", "float", and "ref" may be used within the C code as numbers and read-only strings, respectively, and the C code may contain calls to the PAMALA-provided pattern matching functions `matches()` and `contains()`. This tight integration of PAMALA and C code suggests one of the following two implementation strategies.

One possibility is to build a large, monolithic system consisting of a combined interpreter or compiler for both PAMALA and C, the code for the backtracking algorithm, and the library of pattern matching functions. Such a system would parse the entire program in a single pass, and either execute it immediately (in the case of an interpreter), or generate a machine-language program (in the case of a compiler) that would be compiled and executed separately. This approach has the advantage that the system could check the entire program for syntax errors, and that the C code could be verified with respect to the proper use of PAMALA variables. The disadvantages of the approach include the sheer size and complexity of such a system, and the amount of effort required to implement an interpreter or compiler for the entire C language.

The alternative to the monolithic approach is a system that operates in two phases. The first phase parses the PAMALA code proper, and generates equivalent C code, which is then combined with the unmodified C code extracted from the PAMALA source. In an intermediate step, the generated program is compiled and linked with the code for the backtracking algorithm and the library of pattern matching functions. The second phase

then simply executes the resulting program. The two-phase approach has the advantage that only a source-to-source translator (from PAMALA to C) is required, since compilation of the resulting C code could be left to the native C compiler of the underlying operating system. However, with this scheme the task of error-checking the C code is left entirely to the standard C compiler, which can only detect syntax errors, but can not verify the correct use of PAMALA variables.

Since the emphasis of this dissertation rests on the design and implementation of a novel pattern matching system, rather than on issues of language design and implementation, the second approach was chosen for reasons of simplicity, despite its disadvantages. This chapter describes the implementation of the two-phase scheme by way of an example. The first section illustrates the source-to-source translation done in the first phase, and the second section discusses the issues involved in the execution of the resulting program.

Source-to-Source Translation

The first phase consists of a single-pass, source-to-source translator, which parses the PAMALA statements, reports any syntax or semantic errors, and translates the PAMALA source code into an equivalent C program. Any C source code contained in the PAMALA program is passed on to the second phase unmodified.

The source-to-source translation is illustrated by the sample PAMALA program in Figure 12. Albeit trivial in function, this program contains an example of almost every type of PAMALA statement (the implementation of the remaining two statement types, **`include`** and **`exit/quit`**, is straightforward). The program first declares and initializes the unitary scoring scheme `uni` over the nucleic acid alphabet `DNA`. The values assigned

to the scoring matrix are identical to those shown in Figure 3. The scoring scheme definition is followed by three statements which declare and initialize the numeric variables `thr`, `min`, and `max`, and by the declaration of the function `input()`, which interactively obtains values for those variables. Following the function definition, the variables `s`, `r`, and `p` are declared. The symbol `s` is defined as a character class that matches either of the two nucleic acids adenine (A) and cytosine (C). The pattern `r`, which is declared as a back reference, matches any string containing at least `thr` percent of these two nucleotides. Finally, the pattern `p` restricts the length of any match to `r` (via the conditional expression that references the string matched by `r`) to at least `min` and at most `max` symbols. The variable declarations are followed by two executable statements, the first of which obtains values for the parameters `thr`, `min`, and `max` from the user by calling the function `input()`, and the second uses the system-provided function `contains()` to search the string "TGCAACGT" for all occurrences of the pattern `p`.

```
alpha DNA = [ACGT];
score uni = DNA { x != '$'  &&  y != '$'   #   x == y; };

float thr = 0;
int   min = 0;
int   max = min;

function void input (void) = {
    printf("thr: "); scanf("%f",&thr);
    printf("min: "); scanf("%d",&min);
    printf("max: "); scanf("%d",&max);
};

symbol    s = "[AC]";
reference r = (s*)%thr;
pattern   p = r \ (min <= strlen(r)  &&  strlen(r) <= max);

evaluate input();
evaluate contains(p,"TGCAACGT",STRING,ALL);
```

Figure 12: Sample PAMALA program

From the PAMALA program in Figure 12, the translator generates the variable declarations, function declarations, and main program shown in Figures 13, 14, and 15, respectively.[3]

```
static score *uni;
static float thr;
static int min;
static int max;
static void input (void);
static pattern _s;
static pattern _r; static char *r;
static pattern _p;
```

Figure 13: Variable declarations

Several aspects of the variable declarations in Figure 13 are worth emphasizing. First, Figure 13 does not contain a variable declaration that corresponds to the alphabet DNA in Figure 12. Since an alphabet can be used only in combination with the definition of scoring schemes, it is integrated directly into the code which initializes the scoring scheme (see Figure 14). The declarations of C variables in Figure 13 for the scoring scheme `uni` and the numbers `thr`, `min`, and `max` correspond to the PAMALA declarations in Figure 12 in a straightforward way; the initialization code for the scoring scheme `uni` is contained in Figure 14. For the function `input()`, only a protoype is generated in the declaration section of the program; the function definition itself is contained in Figure 14. PAMALA variables `s`, `r`, and `p` are all translated into C variables of type `pattern`, which is defined as a pointer to the parse tree of the pattern. For the back reference pattern `r`, the translator additionally generates a pointer to a C string, which, during program execution, provides access to the string matched by the pattern.

---

[3] Figures 13, 14, and 15 show the original output of the source-to-source translator, slightly reformatted to improve the readability of the code.

While it would be possible to map the user-defined variable names in PAMALA to arbitrary names in the code generated by the translator, there are several reasons that argue against such a mapping. Since PAMALA variables of types "int", "float", and "ref" can be used directly in the C code, establishing an arbitrary mapping would mean that these names had to be altered in the C code as well — something that would be hard to do without parsing the C code. In addition, any errors in the C code are not detected until the code is compiled. Modifying the C code would therefore make it much more difficult for the user to interpret messages from the compiler or linker, and to pinpoint errors in the original code. Consequently, the C variables generated from PAMALA variables of types "int", "float", and "ref" (in addition to scoring schemes) retain their original names. The names of parse trees generated from PAMALA variables of types "sym", "ref", and "pat" consist of an underscore, followed by the original name. This scheme permits the system to distinguish, for a variable of type "ref", between the parse tree and the string. Since user-defined names can not begin with an underscore, it suffices to begin the names of PAMALA-generated variables and PAMALA-owned globals with two underscores in order to avoid any conflicts between system- and user-defined names.

```
static score *__Score0 (void) {
    register score *__s;
    register float *__w;
    register int *__m,X,x,Y,y,__i,__j,__a[5];
    __s=__AllocateScore(4);
    __w=__s->weight;
    __m=__s->map;
    __m[0]=0;
    __m[65]=1;
    __m[67]=2;
    __m[71]=3;
    __m[84]=4;
    __a[4]=84;
    __a[3]=71;
    __a[2]=67;
    __a[1]=65;
    __a[0]=0;
    for (X=x=__a[__i=0]; __i<5; X=x=__a[++__i])
        for (Y=y=__a[__j=0]; __j<5; Y=y=__a[++__j])
            if (x != '\0'  &&  y != '\0')
                __w[__m[x]*5+__m[y]] = (x == y);
    return(__s);
}

static void input (void) {
    printf("thr: "); scanf("%f",&thr);
    printf("min: "); scanf("%d",&min);
    printf("max: "); scanf("%d",&max);
}

static int __Conditional0 (void) {
    return((int)(min <= strlen(r)  &&  strlen(r) <= max));
}

int (*__Conditionals[])(void) = {__Conditional0};
```

Figure 14: Function declarations

Figure 14 contains three different types of function declarations — functions generated by PAMALA, user-defined functions, and hybrid functions. The function `__Score0()` is an example of the first category — it contains the code for computing the scoring scheme `uni`. The function allocates a scoring scheme for an alphabet of size four with the system-provided function `__AllocateScore()`, and initializes the resulting data structure (the numbers 65, 67, 71, and 84 are the ASCII codes for the letters

A, C, G, and T, respectively). The PAMALA initialization statements themselves are translated into the two nested `for`-loops (which iterate over the scoring matrix), and the enclosed `if`-statement. The `if`-statement's conditional expression is taken directly from the PAMALA program, as is the right-hand side of the assignment statement on the following line. The function returns a pointer to the scoring scheme.

User-defined functions, such as `input()` in the example above, are passed through the translator unchanged.

Finally, the translator generates a hybrid function for the boolean expression on the right-hand side of the conditional pattern `p`. The expression is "wrapped in a function envelope," which simply evaluates the expression, casts the result to an integer, and then returns that integer. The address of the envelope function is placed in a global array, so that it can be called from the pattern matching code without the function name being hard-wired into the code.

```
void main (int argc, char *argv[]) {
    __Initialize(argv[1]);
    __Score=uni=__Score0();
    __Variable("uni",TSCR,__Pointer(PSCR,(void *)uni));
    thr=0;
    __Variable("thr",TFLP,__Pointer(PFLP,(void *)&thr));
    min=0;
    __Variable("min",TINT,__Pointer(PINT,(void *)&min));
    max=min;
    __Variable("max",TINT,__Pointer(PINT,(void *)&max));
    _s=__Symbol("\"[AC]\"");
    __Variable("s",TSYM,_s);
    _r=__Reference("(s*)%thr",(void *)&r);
    __Variable("r",TREF,_r);
    _p=__Pattern("r \\ ()");
    __Variable("p",TPAT,_p);
    input();
    contains(p,"TGCAACGT",STRING,ALL);
}
```

Figure 15: Main program

The main program shown in Figure 15 determines the flow of control in the second phase. After initializing a number of internal parameters with the system-provided function `__Initialize()`, the scoring scheme `uni` is computed and made the default scoring scheme by assignment to the global variable `__Score`. The name `uni`, together with its type and value, is placed in a symbol table with the call to the function `__Variable()`. Similarly, the numeric variables `thr`, `min`, and `max` are initialized, and their names, types, and addresses are added to the symbol table.

As is shown in the following chapter, the second phase requires the parse tree of each pattern in order to determine the optimum decomposition and backtracking order. There are two alternatives for obtaining the parse tree. One possibility is to save the parse trees created by the first phase in a file, and to read this file from the second phase. Alternatively, the patterns themselves can be passed to the second phase and parsed again. The implementation employs the second approach, which explains the need for the

symbol table mentioned above. Not only is this approach more elegant, but it also offers the opportunity to tailor the parse trees to the needs of each phase.

The functions `__Symbol()`, `__Reference()`, and `__Pattern()` parse the patterns `s`, `r`, and `p`, respectively. All functions take the pattern in the form of a C string as their first argument. They construct and return a pointer to the parse tree, which is assigned to the corresponding variable. The conditional pattern `p` has been simplified by removing the boolean expression on the left-hand side; the remaining parentheses are sufficient for parsing, and the correspondence between pattern and function is retained through the order of the declarations. The variable names `s`, `r`, and `p`, together with their types and values, are placed in the symbol table.

The translation of PAMALA **`evaluate`** statements yields the expected C code — a call to the function to be evaluated. However, in order for functions `contains()` and `matches()` to receive the correct types of arguments, the first argument must have an underscore prepended to its name. This can be accomplished by a suitable directive to the C preprocessor, as long as only variables of type "sym", "pat", or "ref" (as opposed to literals) are permitted as arguments to these functions.

At this point, it bears repeating that in the two-phase implementation scheme none of the C code contained in the PAMALA program is parsed by the translator. This fact has two important consequences. First, syntax errors in the C code can not be detected until the entire program is compiled and linked. Hence, any errors that are reported by the C compiler or the linker are with respect to the intermediate program generated by the translator, which may make the search for the error in the PAMALA source more difficult. The second, and more serious, implication is the fact that the resulting C code can not be checked for errors with respect to PAMALA variables. In particular, the

system can not enforce the read-only restriction on the C strings that correspond to back reference patterns, and it can not verify that the boolean expression on the right-hand side of a conditional pattern references only those patterns that occur on the left-hand side. Consequently, the result of violating these restrictions is undefined.

Pattern Evaluation and Matching

Concatenating the code shown in Figures 13, 14, and 15 yields a complete C program. In an intermediate step, this program is compiled with the native C compiler of the underlying operating system. The object code is then linked with the system-provided functions for parsing the patterns and building the parse trees, the pattern evaluation algorithm, the code implementing the backtracking search process, and the library of pattern matching functions.

The second phase executes the resulting program by calling the function `main()`, which consists of variable initializations and calls to the functions `input()` and `contains()`. The scoring scheme `uni` is initialized with a call to the function `__Score0()`, which was generated from the initialization declarations by the translator. Initialization of the numeric variables `thr`, `min`, and `max` is trivial. For patterns `s`, `r`, and `p`, initialization involves parsing the string argument and building a parse tree. While this step is essentially the same as in the first phase, the second phase need not check the patterns for syntax errors. Instead, however, it must check for semantic errors caused by dependencies on global variables that could not be detected in the first phase. Furthermore, the parse tree generated by the second phase differs from the one built in the first phase, since it must accomodate the various attributes and values computed during the pattern evaluation process described in the following chapter. Finally, since the

pattern evaluation algorithm requires a fully expanded parse tree, shorthand notations (for, e.g., list patterns and density symbols) are expanded to the corresponding regular expressions, and references to other pattern variables are replaced by the appropriate parse trees.

Calling a user-defined function, such as `input()` in the example in Figure 12, is straightforward. Any logical errors in the user-defined function, however, can manifest themselves at this point (e.g., in the form of an infinite loop). The system-provided pattern matching functions (function `contains()` in the example above) first evaluate the pattern to determine the optimum decomposition and backtracking order, and then perform the pattern matching process itself.

As noted above, a pattern can depend on one or more global variables, which may not have their final values until a pattern matching function is called. For example, the pattern `p` in Figure 12 depends on the global variables `thr`, `min`, and `max`, which are obtained from the user before the call to `contains()`, but after the pattern has been defined. This dependency seems to require that the pattern be evaluated whenever the matching function is called, rather than at the time of its declaration. However, for some hierarchically defined patterns such as the α/β protein example given in a later chapter, this strategy would result in expensive and unnecessary reevaluations of the pattern. To avoid this inefficiency, the pattern is evaluated once when it is defined. At this time, all dependencies on global variables are recorded, together with the values of those globals. When a pattern matching function is called, the stored values are compared to the current ones, and the pattern is reevaluated only if one or more of these values have changed.

The evaluation algorithm results in a parse tree similar to the one shown in Figure 16. Differently shaded subtrees at the bottom correspond to subpatterns that are matched with

different algorithms, and the arrows in the nodes at the top of the tree represent the backtracking order during the search process.

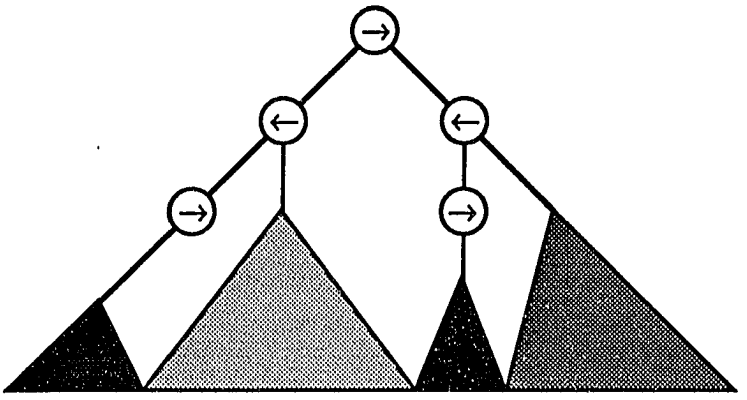

Figure 16: Sample parse tree after pattern evaluation

Once the pattern decomposition and backtracking order have been determined, the algorithm searches the database by interpreting the information encoded in the parse tree. This is, in general, a straightforward process, where the algorithm follows the "arrows" in the tree to the first node representing a pattern matching algorithm. Then, any preprocessing required by the algorithm is performed (e.g., computing the shift tables for the algorithm by Boyer and Moore), and the database is searched for the pattern. Once a match has been found, the backtracking algorithm proceeds to the next such node, performs the necessary preprocessing, and attempts to grow the match. The algorithm continues in this fashion until either the match is complete, or the algorithm "gets stuck," in which case it retreats to the last successful node and attempts to locate an alternate match that allows it to continue in the forward direction.

Whenever the backtracking algorithm encounters a node that has previously been processed, the necessary information has already been computed and the preprocessing

step need not be repeated — unless the pattern contains back references. A pattern containing two or more occurrences of the same back reference is searched for by designating one occurrence as "free" (which is searched for first), and all other occurrences as "bound." This assignment of free and bound occurrences is determined by the pattern evaluation algorithm. Whenever the free occurrence matches a particular string, this string needs to be propagated through the parse tree, as it represents the pattern that must be matched by the bound occurrences. Consequently, whenever the free occurrence matches a different string, the preprocessed information at all bound nodes of the tree must be marked as "out-of-date," so that it will be rebuilt when the backtracking algorithm encounters those nodes. The information necessary for propagating the string is computed when the pattern is parsed during the second phase, and updated when the evaluation algorithm decides on the assignment of free and bound occurrences and on the pattern matching strategy.

This chapter concludes with an illustration of the output generated by the system. In the first phase, PAMALA simply echoes its input, together with any error messages related to the input. In the second phase, the system lists each match and the database position at which that match was found. Customized output for the second phase can be created easily by using the conditional pattern construct in conjunction with user-defined functions. Figure 17 shows the output from a sample PAMALA session, using the program from Figure 12.

```
PAMALA Version 1.0 (Sep 23 1991) Phase 1

alpha DNA = [ACGT];
score uni = DNA { x != '$'  &&  y != '$'   #   x == y; };

float thr = 0;
int   min = 0;
int   max = min;

function void input (void) = {
    printf("thr: "); scanf("%f",&thr);
    printf("min: "); scanf("%d",&min);
    printf("max: "); scanf("%d",&max);
};

symbol    s = "[AC]";
reference r = (s*)%thr;
pattern   p = r \ (min <= strlen(r)  &&  strlen(r) <= max);

evaluate input();
evaluate contains(p,"TGCAACGT",STRING);

Compiling and Linking...

PAMALA Version 1.0 (Sep 23 1991) Phase 2

thr: 0.8
min: 4
max: 5

Match to pattern "p" found at position        2: CAAC
Match to pattern "p" found at position        1: GCAAC
Match to pattern "p" found at position        2: CAACG
```

Figure 17: Sample PAMALA output

CHAPTER 7: PATTERN EVALUATION

A variety of algorithms exists that permits one to directly search for many simple types of patterns, such as keywords, sets of keywords, or regular expressions. However, none of the currently available pattern matching algorithms is capable of searching for an arbitrary pattern from the class of patterns that can be defined with the notation introduced earlier. Developing such a generalist algorithm would neither be feasible nor advisable, as the resulting algorithm would almost certainly be highly complex and inefficient. In the absence of a generalist algorithm, it is necessary to decompose the pattern into smaller subpatterns. These subpatterns can then be searched for individually, using a library of specialized algorithms in conjunction with a backtracking approach for combining the results. The problem then becomes one of determining a pattern decomposition, selecting the algorithms to use for individual subpattern searches, and the order in which to perform these searches. Therefore, an evaluation algorithm needs to be developed that considers many, but not necessarily all, different possibilities of decomposing the pattern and ordering the search, and selects an optimum among them. The criteria for choosing a particular strategy should reflect the goal of producing the most efficient search strategy possible. This approach results in an optimized backtracking algorithm that delays searching for patterns that occur frequently or that are expensive to search for.

The following example illustrates this strategy. Given the regular expression *acad(bb|dd)*, it is possible to search for the entire pattern in a single pass over the database by using a regular expression pattern matching algorithm. Alternatively, the keyword *acad* can be distributed over the expression in parentheses, yielding two keywords, *acadbb* and *acaddd*. These can be searched for either in a single pass over the

database with a set-of-keywords algorithm, or in two separate passes with a single-keyword algorithm. Yet another alternative is to first search for the keyword *acad* with a single-keyword algorithm. If that search succeeds, the algorithm needs to search for the remainder of the pattern, using either two passes of a single-keyword algorithm, or a single pass with a set-of-keywords algorithm. Finally, the order of the search can be reversed for the last alternative. The pattern evaluation algorithm to be developed should consider all of these possibilities, and select the one expected to be the most efficient.

The optimized backtracking strategy was inspired by the goal-directed evaluation mechanism first introduced in Icon [Griswold *et al.*, 1981]. Goal-directed evaluation is based on the concept of generators — expressions that are capable of producing more than one value. Briefly, any computation in Icon is assumed to be driven by some goal, and continues execution until either that goal has been reached, or until all possible alternatives have been tried (and have failed). Every Icon expression returns a signal, which indicates whether the expression succeeded or failed, and which controls the flow of execution. If an expression succeeds, it also returns a value, which is used in the standard fashion. In either case, the expression is suspended after the result has been returned. The computation proceeds in the standard order, as long as all the expressions involved succeed. Once an expression fails, however, the computation backtracks by resuming the last suspended expression. If this expression can produce an alternate value, execution continues in the forward direction from that point; otherwise, backtracking continues. The programmer can control the backtracking process with both conventional and specialized control structures. By treating pattern matching functions as generators, PAMALA implements a similar backtracking strategy. The first call of such a function causes it to begin a search of the database for the specified subpattern. The function returns with the first match found. When control returns to this function during

backtracking, the function continues the search where it left off, and returns with the next match. The overall goal of the backtracking process is to produce all matches to the entire pattern. In contrast to Icon, however, the backtracking process itself is controlled mostly by the system, so as to relieve the user of as much of the programming burden as possible.

Some of Staden's ideas [Staden, 1988] also influenced the design of the current system. Noting that the backtracking strategy in his system could easily result in highly inefficient searches, Staden suggested that the user order motifs and spacers by increasing match frequency so as to improve the efficiency of the search. In a later paper, Staden presents several algorithms for calculating the probabilities of finding matches to certain kinds of patterns in biological sequences [Staden, 1989]. However, in the context of his system, this proposal merely serves as an *ad hoc* solution to an important problem, and relies on the user to avoid inefficiencies inherent in the design of the search algorithm. In PAMALA, on the other hand, decomposing the pattern and determining an optimum backtracking order is a central aspect of the pattern matching strategy and is an integrated, fully automated part of the system design. In addition, PAMALA differs from Staden's program in that it supports a much larger class of patterns, does not require an artificial distinction between motifs and spacers, and provides a general concept for approximate pattern matching.

Optimized Backtracking

A detailed description of backtracking in general is given by, among others, Golomb and Baumert [1965], and the performance of backtracking algorithms has been analyzed and discussed by Knuth [1975]. Briefly, a backtracking algorithm can be applied to a

problem Q, if the overall problem can be broken into several smaller, mutually independent problems Q_1, ..., Q_K. Each of the subproblems Q_i, $1 \leq i \leq K$, can be viewed as a multi-valued function which can assume any of m_i values q_i. The overall problem Q can then be solved by finding one (or, depending on the problem, all) vector(s) of the form $(q_1, \ldots, q_K)$, which are extremal with respect to some criterion function f (e.g., which maximize f). The straightforward approach at solving the problem Q would enumerate the problem space $Q_1 \times \ldots \times Q_K$ by evaluating the function f for each of the $M = \prod_{i=1}^{K} m_i$ sample vectors. Backtracking, on the other hand, takes advantage of the fact that the subproblems Q_i are independent of each other, and hence the vector $(q_1, \ldots, q_K)$ can be assembled one component at a time. By using this fact, it may be possible to eliminate large sections of the problem space from consideration, if it is possible to determine whether or not a partial vector $(q_1, \ldots, q_k)$, $1 \leq k < K$, can still lead to the desired extremal value. If the cost of computing the solution to any given subproblem and the chances of success are known (or can be estimated), the algorithm can be further optimized by ordering the Q_i, such that the subproblems with the least cost of evaluation or the least chance of success are computed first.

The general formulation of the backtracking algorithm above can be applied to the pattern matching domain as follows. The overall problem Q corresponds to locating all matches to a pattern p in a database s of length N. Given a decomposition of the pattern p into subpatterns p_1, ..., p_K, each subpattern p_i corresponds to a function Q_i, which, for every database position j, assumes either the value *true* or *false*. The value *true* corresponds to a match to p_i beginning at position s_j, whereas the value *false* indicates that no such match exists. A solution to the overall problem Q therefore consists of all

positions j for which a particular predicate (the form of which depends on p) of all elements of the vector $(q_1, \ldots, q_K)$ yields *true*.[4]

The straightforward method for solving the pattern matching problem evaluates every function Q_i, $1 \leq i \leq K$, for every database position j, $1 \leq j \leq N$. The standard backtracking method begins by evaluating the function Q_1 for every database position j, $1 \leq j \leq N$. If $Q_1(j)$ fails, i.e., if there is no match to p_1 at position j, the remaining functions $Q_2, \ldots, Q_K$ need not be evaluated at this particular position. If, on the other hand, $Q_1(j)$ succeeds, $Q_2(j)$ is evaluated. If it, too, succeeds, then $Q_3(j)$, $Q_4(j)$, ... are evaluated, until either some $Q_i(j)$, $i \leq K$, fails, or until $Q_K(j)$ succeeds, in which case a match has been found. The optimized backtracking approach additionally orders the subproblems Q_i such that the expected time spent searching the database is minimized. The following section describes how the optimum decomposition and backtracking order can be determined.

Pattern Decomposition and Backtracking Order

To determine an optimum decomposition of a pattern p into subpatterns, together with the corresponding backtracking order, it is necessary to develop a criterion for assessing the "goodness" of any particular combination. An intuitively appealing criterion is the expected cost (i.e., time) of searching a database s of length N for the overall pattern p. This cost, written as $C_u(p,N)$, depends on the particular choice of subpatterns (through the relative costs of the algorithms that are used to search for those subpatterns), and on the order in which the search proceeds (through the match frequencies of the subpatterns). An optimum combination of subpatterns and backtracking order is one that minimizes this

---

[4] To simplify the treatment, this formulation of the problem ignores the fact that different subpatterns match at different offsets from the beginning of the overall pattern. However, the generalization is straightforward.

cost. This section describes how the minimum cost of searching for a given pattern can be determined, together with the corresponding decomposition of the pattern and backtracking order. To simplify the treatment, it is assumed that the overall pattern does not contain any back references. The following section explains how the algorithm needs to be modified to accomodate back references.

Considering the large number of decompositions that are possible for any non-trivial pattern, it is reasonable to limit the number that must be considered by the evaluation algorithm. This is achieved by two separate measures. First, certain types of patterns (such as spacers and keywords) are designated as *atomic*, and no further decompositions of those patterns are considered. Second, the decomposition of a pattern is based on a depth-first traversal of the pattern's parse tree in the following way. The evaluation algorithm decides, at each node, whether to match the entire pattern represented by this node with one of the basic pattern matching algorithms (and if so, which one), or whether to match the subpatterns represented by the left and right subtrees separately (and if so, in what order). This strategy results in an evaluation algorithm that considers a large number of alternatives and assembles the pattern matching strategy for the overall pattern from bottom to top. It is important to recognize, however, that this strategy does not, in general, consider all possibilities, because all decompositions are based on a depth-first traversal of the parse tree, where at every binary node the entire left subtree must have been evaluated before evaluation of the right subtree can begin (or vice versa).

The evaluation algorithm can, in general, choose between algorithms for matching spacers, keywords, sets of keywords, regular expressions, approximate regular expressions, and approximate networks, as well as two orders of evaluation, namely left-to-right and right-to-left. Collectively, these choices are referred to as *methods of*

evaluation. Depending on the particular type of node, however, the number of choices may be more limited. For example, atomic patterns or approximate patterns must not be broken down, but must be searched for with one of the basic pattern matching algorithms. On the other hand, conditional patterns, or the conjunction or difference of two patterns can not be evaluated atomically, but must be broken into smaller subpatterns.

The evaluation algorithm traverses the parse tree, determines at each node the set of available strategies, calculates the costs associated with each alternative, and makes a choice. The set of methods that are available for choosing at any given node can be determined by computing, for each node of the parse tree, the predicates *IsSpacer*, *IsKeyword*, *IsKeySet*, *IsNetwork*, and *IsRegular*, together with the numeric attribute *NumberOfKeywords*. Since the pattern evaluation algorithm operates on the fully expanded parse tree generated by the second phase, the tree contains nodes which correspond to patterns of the form $p \setminus e, p \mid q, p \,\&\, q, p - q, p{\cdot}q, p^*, p?, p \,\%\, t$, ~p, <i,j>, "$a_1a_2\ldots a_n$", and . (the wildcard symbol). Figures 18 through 23 list, for these types of nodes, the recurrences for computing the aforementioned predicates and attributes. The numeric attribute *NumberOfKeywords*, in conjunction with a threshold T, limits the size of any set of keywords that is matched with a set-of-keywords algorithm. The predicate *IsApproximate* "modulates" the predicates *IsNetwork* and *IsRegular* — its value (*true* for all nodes that have an ancestor of the form $p \,\%\, t$, and *false* for all other nodes) determines whether an algorithm for exact or approximate patterns is used for searching the database. The recurrence for the predicate *IsSpacer* makes use of the predicates *IsCharacterClass* and *IsWildcard*. The value of these predicates is *true* for any node that represent character classes and wildcard symbols, respectively, and *false* for all other nodes. The values of all predicates and attributes can be computed in a single, depth-first pass over the parse tree.

$$
\begin{aligned}
IsSpacer(p\backslash e) &= false \\
IsSpacer(p \mid q) &= false \\
IsSpacer(p \,\&\, q) &= false \\
IsSpacer(p - q) &= false \\
IsSpacer(p \cdot q) &= IsSpacer(p) \wedge IsSpacer(q) \\
IsSpacer(p^*) &= false \\
IsSpacer(p?) &= false \\
IsSpacer(p \,\%\, t) &= false \\
IsSpacer(\sim p) &= IsSpacer(p) \\
IsSpacer(<i,j>) &= true \\
IsSpacer(\text{"}a_1 a_2 \cdots a_n\text{"}) &= false \\
IsSpacer(.) &= \begin{cases} false & \text{if } IsApproximate(.) \\ true & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 18: *IsSpacer* recurrences[5]

[5] The recurrences reflect the limitations of the algorithm. For example, while the alternation, conjunction, or difference of two spacers may still be considered to be a spacer, it can not be matched with the implemented algorithm, which requires a single, contiguous interval.

$$
\begin{array}{lcl}
\mathit{IsKeyword}(p\backslash e) & = & \mathit{false} \\
\mathit{IsKeyword}(p \mid q) & = & \mathit{false} \\
\mathit{IsKeyword}(p \,\&\, q) & = & \mathit{false} \\
\mathit{IsKeyword}(p - q) & = & \mathit{false} \\
\mathit{IsKeyword}(p \cdot q) & = & \mathit{IsKeyword}(p) \wedge \mathit{IsKeyword}(q) \\
\mathit{IsKeyword}(p^*) & = & \mathit{false} \\
\mathit{IsKeyword}(p?) & = & \mathit{false} \\
\mathit{IsKeyword}(p \,\%\, t) & = & \mathit{IsKeyword}(p) \\
\mathit{IsKeyword}(\sim p) & = & \mathit{IsKeyword}(p) \\
\mathit{IsKeyword}(<i,j>) & = & \mathit{false} \\
\mathit{IsKeyword}(\text{"}a_1 a_2 \cdots a_n\text{"}) & = & \begin{cases} \mathit{false} & \text{if } \mathit{IsApproximate}(\text{"}a_1 a_2 \cdots a_n\text{"}) \ \vee \\ & \exists\, i\text{: } \mathit{IsCharacterClass}(a_i) \vee \mathit{IsWildcard}(a_i) \\ \mathit{true} & \text{otherwise} \end{cases} \\
\mathit{IsKeyword}(.) & = & \mathit{false}
\end{array}
$$

Figure 19: *IsKeyword* recurrences

$$
\begin{aligned}
IsKeySet(p \backslash e) &= false \\
IsKeySet(p \mid q) &= \begin{cases} IsKeySet(p) \wedge IsKeySet(q) \wedge \\ NumberOfKeyWords(p \mid q) < T \end{cases} \\
IsKeySet(p \,\&\, q) &= false \\
IsKeySet(p - q) &= false \\
IsKeySet(p \cdot q) &= \begin{cases} IsKeySet(p) \wedge IsKeySet(q) \wedge \\ NumberOfKeyWords(p \cdot q) < T \end{cases} \\
IsKeySet(p^*) &= false \\
IsKeySet(p?) &= IsKeySet(p) \\
IsKeySet(p \,\%\, t) &= IsKeySet(p) \\
IsKeySet(\sim p) &= IsKeySet(p) \\
IsKeySet(<i,j>) &= false \\
IsKeySet(\text{"}a_1 a_2 \cdots a_n\text{"}) &= \begin{cases} false & \text{if } IsApproximate(\text{"}a_1 a_2 \cdots a_n\text{"}) \\ true & \text{otherwise} \end{cases} \\
IsKeySet(.) &= \begin{cases} false & \text{if } IsApproximate(.) \\ true & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 20: *IsKeySet* recurrences[6]

---

6 The recurrences reflect the fact that the concatenation or alternation of two sets of keywords can be transformed into a single set of keywords. At the same time, the recurrence limits the size of the sets created in this way by applying a threshold T to the number of words in the set.

$$
\begin{aligned}
NumberOfKeywords(p \backslash e) &= \infty \\
NumberOfKeywords(p \mid q) &= \begin{cases} NumberOfKeywords(p) + \\ NumberOfKeywords(q) \end{cases} \\
NumberOfKeywords(p \,\&\, q) &= \infty \\
NumberOfKeywords(p - q) &= \infty \\
NumberOfKeywords(p \cdot q) &= \begin{cases} NumberOfKeywords(p) \cdot \\ NumberOfKeywords(q) \end{cases} \\
NumberOfKeywords(p^*) &= \infty \\
NumberOfKeywords(p?) &= NumberOfKeywords(p) + 1 \\
NumberOfKeywords(p \,\%\, t) &= \infty \\
NumberOfKeywords(\sim p) &= NumberOfKeywords(p) \\
NumberOfKeywords(<i,j>) &= \infty \\
NumberOfKeywords("a_1 a_2 \cdots a_n") &= \begin{cases} \infty & \text{if } IsApproximate("a_1 a_2 \cdots a_n") \\ \prod_{i=1}^{n} |a_i| & \text{otherwise} \end{cases} \\
NumberOfKeywords(.) &= \begin{cases} \infty & \text{if } IsApproximate(.) \\ |\Sigma| & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 21: *NumberOfKeywords* recurrences[7]

[7] The recurrences reflect the fact that the concatenation or alternation of two sets of keywords can be transformed into a single set of keywords.

$$
\begin{aligned}
\mathit{IsNetwork}(p \backslash e) &= \mathit{false} \\
\mathit{IsNetwork}(p \mid q) &= \mathit{IsNetwork}(p) \wedge \mathit{IsNetwork}(q) \\
\mathit{IsNetwork}(p \,\&\, q) &= \mathit{false} \\
\mathit{IsNetwork}(p - q) &= \mathit{false} \\
\mathit{IsNetwork}(p \cdot q) &= \mathit{IsNetwork}(p) \wedge \mathit{IsNetwork}(q) \\
\mathit{IsNetwork}(p^*) &= \mathit{false} \\
\mathit{IsNetwork}(p?) &= \mathit{IsNetwork}(p) \\
\mathit{IsNetwork}(p \,\%\, t) &= \mathit{IsNetwork}(p) \\
\mathit{IsNetwork}(\sim p) &= \mathit{IsNetwork}(p) \\
\mathit{IsNetwork}(<i,j>) &= \mathit{false} \\
\mathit{IsNetwork}(\text{"}a_1 a_2 \cdots a_n\text{"}) &= \mathit{true} \\
\mathit{IsNetwork}(.) &= \mathit{true}
\end{aligned}
$$

Figure 22: *IsNetwork* recurrences

$$
\begin{array}{lcl}
\mathit{IsRegular}(p\backslash e) & = & \mathit{false} \\
\mathit{IsRegular}(p \mid q) & = & \mathit{IsRegular}(p) \wedge \mathit{IsRegular}(q) \\
\mathit{IsRegular}(p \,\&\, q) & = & \mathit{false} \\
\mathit{IsRegular}(p - q) & = & \mathit{false} \\
\mathit{IsRegular}(p \cdot q) & = & \mathit{IsRegular}(p) \wedge \mathit{IsRegular}(q) \\
\mathit{IsRegular}(p^*) & = & \mathit{IsRegular}(p) \\
\mathit{IsRegular}(p?) & = & \mathit{IsRegular}(p) \\
\mathit{IsRegular}(p \,\%\, t) & = & \mathit{IsRegular}(p) \\
\mathit{IsRegular}(\sim p) & = & \mathit{IsRegular}(p) \\
\mathit{IsRegular}(<i,j>) & = & \mathit{false} \\
\mathit{IsRegular}(\text{"}a_1 a_2 \cdots a_n\text{"}) & = & \mathit{true} \\
\mathit{IsRegular}(.) & = & \mathit{true}
\end{array}
$$

Figure 23: *IsRegular* recurrences

Given the set of methods available at a particular node, the evaluation algorithm then needs to determine the cost of searching for the pattern represented by that node for each alternative, and select the method that achieves the minimum cost.

In the case of a pattern that can be matched directly with one of the basic algorithms, the cost of searching depends on the particular algorithm used. However, since the final backtracking order is not known until the entire pattern has been evaluated, the evaluation algorithm needs to consider the case where the subpattern p_i is the first one to be matched, as well as the case where other subpatterns have already been matched. If the subpattern p_i is the first one to be matched, the entire database must be searched for all occurrences of p_i. The cost of performing such an *unanchored* search is denoted by $C_u(p_i,N)$. If, on the other hand, the subpatterns on one or both sides of the subpattern p_i have already been matched, one or both ends of p_i are already known, and the pattern matching algorithm only needs to determine whether or not there is in fact a match at this particular location. The costs of performing such a *left-*, *right-*, and *both-anchored* search are denoted by $C_l(p_i)$, $C_r(p_i)$, and $C_b(p_i)$, respectively. Hence, the evaluation algorithm needs to compute four different cost values (and the corresponding methods of evaluation) at each node. For the basic pattern matching algorithms, the costs of performing left-anchored, right-anchored, and both-anchored searches are considered to be identical.

In the case of a pattern that can be decomposed into several subpatterns, the cost of searching for the combined pattern depends on the respective costs for matching the subpatterns, the frequencies with which the subpatterns match (since these determine the likelihood that the algorithm must search for the remainder of the pattern), and the possible variations in the length of the match (since these determine the interval over

which an anchored search must be performed). Hence, in addition to the predicates and attributes determined earlier, the evaluation algorithm also needs to compute the numeric attributes f (a rough approximation of the match frequency of the pattern), d (the length of the shortest match), and δ (the difference between the shortest and longest match) for each node of the parse tree. The recurrences for computing these attributes are given in Figures 24 through 26. Like the predicates and attributes discussed earlier, they can be computed in a single, depth-first pass over the parse tree of the pattern.

Given the values of the predicates and attributes, the four cost values for any node in the parse tree can be computed from the recurrences in Figures 27 through 30. The basis of those recurrences is given by the *cost constants* in Figures 31 and 32. The cost $c_u(p,D)$ represents the minimum expected cost, over all applicable basic pattern matching algorithms, of performing an unanchored search for the pattern p over a database of length D. Similarly, the cost $c_a(p)$ represents the minimum expected cost of performing an anchored search for the pattern p. Numeric values for the individual c_a and c_u are determined for any given machine and operating system with a Monte-Carlo simulation over a random database, upon installation of the PAMALA software. The accuracy of the constants (and, by implication, the accuracy of the underlying model) is discussed in the chapter on results.

$$
\begin{aligned}
f(p \backslash e) &= f(p) \cdot f(e) \\
f(p \mid q) &= \min \left\{ \begin{array}{c} f(p)+f(q) \\ 1 \end{array} \right\} \\
f(p \,\&\, q) &= f(p) \cdot f(q) \\
f(p - q) &= f(p) \cdot (1 - f(q)) \\
f(p \cdot q) &= f(p) \cdot f(q) \\
f(p^*) &= 1 \\
f(p?) &= 1 \\
f(p \,\%\, t) &= \mathit{MonteCarlo}(p \,\%\, t) \\
f(\sim p) &= f(p) \\
f(<i,j>) &= 1 \\
f(\text{"}a_1 a_2 \cdots a_n\text{"}) &= \left\{ \begin{array}{ll} 1 & \text{if } \mathit{IsApproximate}(\text{"}a_1 a_2 \cdots a_n\text{"}) \\ f_{a_1} \cdot f_{a_2} \cdot \cdots \cdot f_{a_n} & \text{otherwise} \end{array} \right. \\
f(.) &= 1
\end{aligned}
$$

Figure 24: Frequency recurrences[8]

---

8 The recurrences are based on the match frequency of individual characters, f_a. For an alphabet of size $|\Sigma|$, f_a is assumed to be $1/|\Sigma|$.

For patterns of the form $p \mid q$, $p \,\&\, q$, and $p - q$, the recurrences assume that p and q are uncorrelated. This is a necessary approximation which simplifies determination of the respective frequencies.

For approximate patterns of the form $p \,\%\, t$, the match frequency can not be determined analytically, not even under simplifying assumptions. While theoretically the strings matched by such a pattern could be enumerated, this methods requires exponential time for some patterns. The match frequencies are therefore determined empirically with a Monte-Carlo simulation over a random database. A later section describes how this estimate is subsequently refined.

For conditional patterns of the form $p \backslash e$, the match frequency of the boolean expression e, $f(e)$, can not be determined analytically or empirically. The recurrence therefore assumes this match frequency to be 1. A later section describes how this estimate is subsequently refined.

$$
\begin{array}{lcl}
d(p\backslash e) & = & d(p) \\
d(p \mid q) & = & \min\left\{ \begin{array}{l} d(p) \\ d(q) \end{array} \right\} \\
d(p \,\&\, q) & = & \max\left\{ \begin{array}{l} d(p) \\ d(q) \end{array} \right\} \\
d(p - q) & = & d(p) \\
d(p \cdot q) & = & d(p) + d(q) \\
d(p^*) & = & 0 \\
d(p?) & = & 0 \\
d(p \,\%\, t) & = & 0 \\
d(\sim p) & = & d(p) \\
d(<i,j>) & = & i \\
d(\text{"}a_1 a_2 \cdots a_n\text{"}) & = & \left\{ \begin{array}{ll} 0 & \text{if } \mathit{IsApproximate}(\text{"}a_1 a_2 \cdots a_n\text{"}) \\ n & \text{otherwise} \end{array} \right. \\
d(.) & = & \left\{ \begin{array}{ll} 0 & \text{if } \mathit{IsApproximate}(.) \\ 1 & \text{otherwise} \end{array} \right.
\end{array}
$$

Figure 25: Minimum size recurrences[9]

---

[9] For approximate patterns of the form $p \,\%\, t$, the minimum size can not be determined analytically. The recurrence therefore assumes the minimum size to be 0.

For spacers of the form $<i,j>$, the recurrence assumes $i \leq j$.

$$
\begin{aligned}
\delta(p\backslash e) &= \delta(p) \\
\delta(p \mid q) &= \max\left\{\begin{matrix} d(p)+\delta(p) \\ d(q)+\delta(q) \end{matrix}\right\} - \min\left\{\begin{matrix} d(p) \\ d(q) \end{matrix}\right\} \\
\delta(p \,\&\, q) &= \min\left\{\begin{matrix} d(p)+\delta(p) \\ d(q)+\delta(q) \end{matrix}\right\} - \max\left\{\begin{matrix} d(p) \\ d(q) \end{matrix}\right\} \\
\delta(p-q) &= \delta(p) \\
\delta(p \cdot q) &= \delta(p) + \delta(q) \\
\delta(p^*) &= \infty \\
\delta(p?) &= d(p) + \delta(p) \\
\delta(p \,\%\, t) &= \infty \\
\delta(\sim p) &= \delta(p) \\
\delta(<i,j>) &= j - i \\
\delta(\text{"}a_1 a_2 \cdots a_n\text{"}) &= \begin{cases} \infty & \text{if } \mathit{IsApproximate}(\text{"}a_1 a_2 \cdots a_n\text{"}) \\ 0 & \text{otherwise} \end{cases} \\
\delta(.) &= \begin{cases} \infty & \text{if } \mathit{IsApproximate}(.) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 26: Variance recurrences[10]

[10] For approximate patterns of the form $p \,\%\, t$, the variance can not be determined analytically. The recurrence therefore assumes the variance to be ∞.

For spacers of the form $<i,j>$, the recurrence assumes $i \leq j$.

$$
\begin{aligned}
C_u(p\backslash e, D) &= C_u(p, D) + f_p \cdot D \cdot C(e) \\
C_u(p \mid q, D) &= \min \left\{ \begin{array}{c} c_u(p \mid q, D) \\ C_u(p, D) + C_u(q, D) \end{array} \right\} \\
C_u(p \,\&\, q, D) &= \min \left\{ \begin{array}{c} c_u(p \,\&\, q, D) \\ C_u(p, D) + f_p \cdot D \cdot C_b(q) \\ C_u(q, D) + f_q \cdot D \cdot C_b(p) \end{array} \right\} \\
C_u(p - q, D) &= \min \left\{ \begin{array}{c} c_u(p - q, D) \\ C_u(p, D) + f_p \cdot D \cdot C_b(q) \end{array} \right\} \\
C_u(p \cdot q, D) &= \min \left\{ \begin{array}{c} c_u(p \cdot q, D) \\ C_u(p, D) + f_p \cdot D \cdot C_l(q) \\ C_u(q, D) + f_q \cdot D \cdot C_r(p) \end{array} \right\} \\
C_u(p*, D) &= \min \left\{ \begin{array}{c} c_u(p*, D) \\ C_u(p, D) + \dfrac{f_p}{1 - f_p} \cdot D \cdot C_l(p) \\ C_u(p, D) + \dfrac{f_p}{1 - f_p} \cdot D \cdot C_r(p) \end{array} \right\} \\
C_u(p?, D) &= \min \left\{ \begin{array}{c} c_u(p?, D) \\ C_u(p, D) + C_u(\varepsilon, D) \end{array} \right\} \\
C_u(p \,\%\, t, D) &= c_u(p, D) \\
C_u(\sim p, D) &= C_u(p, D) \\
C_u(<i,j>, D) &= c_u(<i,j>, D) \\
C_u(\text{"}a_1 a_2 \cdots a_n\text{"}, D) &= c_u(\text{"}a_1 a_2 \cdots a_n\text{"}, D) \\
C_u(., D) &= c_u(., D)
\end{aligned}
$$

Figure 27: Cost recurrences for unanchored searches[11]

---

11 For conditional patterns of the form $p \backslash e$, the cost of evaluating the boolean expression e, $C(e)$, can not be determined analytically or empirically. The recurrence therefore assumes this cost to be 0. A later section describes how this estimate is subsequently refined.

$$
\begin{aligned}
C_l(p \backslash e) &= C_l(p) + f_p \cdot C(e) \\
C_l(p \mid q) &= \min \left\{ \begin{array}{c} c_a(p \mid q) \\ C_l(p) + C_l(q) \end{array} \right\} \\
C_l(p \,\&\, q) &= \min \left\{ \begin{array}{c} c_a(p \,\&\, q) \\ C_l(p) + f_p \cdot C_b(q) \\ C_l(q) + f_q \cdot C_b(p) \end{array} \right\} \\
C_l(p - q) &= \min \left\{ \begin{array}{c} c_a(p - q) \\ C_l(p) + f_p \cdot C_b(q) \end{array} \right\} \\
C_l(p \cdot q) &= \min \left\{ \begin{array}{c} c_a(p \cdot q) \\ C_l(p) + f_p \cdot C_l(q) \\ C_u(q, \delta(p)) + f_q \cdot \delta(q) \cdot C_b(p) \end{array} \right\} \\
C_l(p^*) &= \min \left\{ \begin{array}{c} c_a(p^*) \\ C_l(p) + \frac{f_p}{1 - f_p} \cdot C_l(p) \end{array} \right\} \\
C_l(p?) &= \min \left\{ \begin{array}{c} c_a(p?) \\ C_l(p) + C_l(\varepsilon) \end{array} \right\} \\
C_l(p \,\%\, t) &= c_a(p) \\
C_l(\sim p) &= C_a(p) \\
C_l(<i,j>) &= c_a(<i,j>) \\
C_l(\text{"}a_1 a_2 \cdots a_n\text{"}) &= c_a(\text{"}a_1 a_2 \cdots a_n\text{"}) \\
C_l(.) &= c_a(.)
\end{aligned}
$$

Figure 28: Cost recurrences for left-anchored searches[12]

12 For conditional patterns of the form $p \backslash e$, the cost of evaluating the boolean expression e, $C(e)$, can not be determined analytically or empirically. The recurrence therefore assumes this cost to be 0. A later section describes how this estimate is subsequently refined.

$$
\begin{aligned}
C_r(p \backslash e) &= C_r(p) + f_p \cdot C(e) \\
C_r(p \mid q) &= \min \left\{ \begin{array}{c} c_a(p \mid q) \\ C_r(p) + C_r(q) \end{array} \right\} \\
C_r(p \,\&\, q) &= \min \left\{ \begin{array}{c} c_a(p \,\&\, q) \\ C_r(p) + f_p \cdot C_b(q) \\ C_r(q) + f_q \cdot C_b(p) \end{array} \right\} \\
C_r(p - q) &= \min \left\{ \begin{array}{c} c_a(p - q) \\ C_r(p) + f_p \cdot C_b(q) \end{array} \right\} \\
C_r(p \cdot q) &= \min \left\{ \begin{array}{c} c_a(p \cdot q) \\ C_u(p, \delta(q)) + f_p \cdot \delta(p) \cdot C_b(q) \\ C_r(q) + f_q \cdot C_r(p) \end{array} \right\} \\
C_r(p^*) &= \min \left\{ c_a(p^*), C_r(p) + \frac{f_p}{1 - f_p} \cdot C_r(p) \right\} \\
C_r(p?) &= \min \left\{ \begin{array}{c} c_a(p?) \\ C_r(p) + C_r(\varepsilon) \end{array} \right\} \\
C_r(p \,\%\, t) &= c_a(p) \\
C_r(\sim p) &= C_a(p) \\
C_r(<i,j>) &= c_a(<i,j>) \\
C_r("a_1 a_2 \cdots a_n") &= c_a("a_1 a_2 \cdots a_n") \\
C_r(.) &= c_a(.)
\end{aligned}
$$

Figure 29: Cost recurrences for right-anchored searches[13]

---

[13] For conditional patterns of the form $p \backslash e$, the cost of evaluating the boolean expression e, $C(e)$, can not be determined analytically or empirically. The recurrence therefore assumes this cost to be 0. A later section describes how this estimate is subsequently refined.

$$
\begin{aligned}
C_b(p \backslash e) &= C_b(p) + f_p \cdot C(e) \\
C_b(p \mid q) &= \min \left\{ \begin{array}{c} c_a(p \mid q) \\ C_b(p) + C_b(q) \end{array} \right\} \\
C_b(p \,\&\, q) &= \min \left\{ \begin{array}{c} c_a(p \,\&\, q) \\ C_b(p) + f_p \cdot C_b(q) \\ C_b(q) + f_q \cdot C_b(p) \end{array} \right\} \\
C_b(p - q) &= \min \left\{ \begin{array}{c} c_a(p - q) \\ C_b(p) + f_p \cdot C_b(q) \end{array} \right\} \\
C_b(p \cdot q) &= \min \left\{ \begin{array}{c} c_a(p \cdot q) \\ C_l(p) + f_p \cdot C_l(q) \\ C_r(q) + f_q \cdot C_r(p) \end{array} \right\} \\
C_b(p^*) &= \min \left\{ \begin{array}{c} c_a(p^*) \\ C_l(p) + \frac{f_p}{1 - f_p} \cdot C_l(p) \\ C_r(p) + \frac{f_p}{1 - f_p} \cdot C_r(p) \end{array} \right\} \\
C_b(p?) &= \min \left\{ \begin{array}{c} c_a(p?) \\ C_b(p) + C_b(\varepsilon) \end{array} \right\} \\
C_b(p \,\%\, t) &= c_a(p) \\
C_b(\sim p) &= C_a(p) \\
C_b(<i,j>) &= c_a(<i,j>) \\
C_b(\text{"}a_1 a_2 \cdots a_n\text{"}) &= c_a(\text{"}a_1 a_2 \cdots a_n\text{"}) \\
C_b(.) &= c_a(.)
\end{aligned}
$$

Figure 30: Cost recurrences for left-and-right-anchored searches[14]

---

[14] For conditional patterns of the form $p \backslash e$, the cost of evaluating the boolean expression e, $C(e)$, can not be determined analytically or empirically. The recurrence therefore assumes this cost to be 0. A later section describes how this estimate is subsequently refined.

$$c_u(p,D) = \min \begin{cases} c_u^{SP} \cdot |p| \cdot D & \text{if } \neg IsApproximate(p) \wedge IsSpacer(p) \\ c_u^{BM} \cdot \frac{1}{|p|} \cdot D & \text{if } \neg IsApproximate(p) \wedge IsKeyWord(p) \\ c_u^{AC} \cdot D & \text{if } \neg IsApproximate(p) \wedge IsKeySet(p) \\ c_u^{RE} \cdot D & \text{if } \neg IsApproximate(p) \wedge IsRegExpr(p) \\ c_u^{AN} \cdot \frac{1}{t} \cdot D & \text{if } IsApproximate(p) \wedge IsNetWork(p) \\ c_u^{AR} \cdot |p| \cdot D & \text{if } IsApproximate(p) \wedge IsRegExpr(p) \\ \infty \end{cases}$$

Figure 31: Cost of performing an unanchored search[15]

[15] The length-relative threshold associated with an approximate pattern is denoted by t.

$$c_a(p) = \min \begin{cases} c_a^{SP} \cdot |p| & \text{if } \neg IsApproximate(p) \wedge IsSpacer(p) \\ c_a^{BM} \cdot |p| & \text{if } \neg IsApproximate(p) \wedge IsKeyWord(p) \\ c_a^{AC} & \text{if } \neg IsApproximate(p) \wedge IsKeySet(p) \\ c_a^{RE} & \text{if } \neg IsApproximate(p) \wedge IsRegExpr(p) \\ c_a^{AN} & \text{if } IsApproximate(p) \wedge IsNetWork(p) \\ c_a^{AR} \cdot |p| & \text{if } IsApproximate(p) \wedge IsRegExpr(p) \\ \infty \end{cases}$$

Figure 32: Cost of performing an anchored search

The values of the various attributes and predicates can be computed together with the costs for alternative methods of evaluation and the backtracking order during a single, depth-first traversal of the parse tree. The resulting evaluation algorithm is summarized in Figure 33. At each node, the inherited predicate *IsApproximate* is computed first, followed by the evaluation of the left and right subtrees (Λ stands for a nonexisting subtree), and the computation of the synthesized predicates and attributes (the predicates *IsSpacer*, *IsKeyword*, *IsKeySet*, *IsNetwork*, *IsRegular*, *IsCharacterClass*, and *IsWildcard*, and the attributes *NumberOfKeywords*, f, d, and δ). Finally, the cost values are computed for unanchored, left-, right-, and both-anchored searches, and the optimum method for each case is retained.

```
EvaluatePattern (p)

    compute inherited predicates and attributes
    if p.lftSubtree ≠ Λ then
        EvaluatePattern(p.lftSubtree)
    end
    if p.rgtSubtree ≠ Λ then
        EvaluatePattern(p.rgtSubtree)
    end
    compute synthesized predicates and attributes
    compute Cu(p,N), Cl(p), Cr(p), Cb(p) and retain optimum methods
end
```

Figure 33: Pattern evaluation algorithm

The following example illustrates the operation of the pattern evaluation algorithm. Assuming a pattern of the form $p = q \cdot r$, the pattern evaluation algorithm has a choice of three methods — it can either search for the entire pattern p with a single algorithm, or it can match q and r separately, either from left to right, or from right to left. These choices

are pictured in Figure 34, where solid lines represent successful searches, and dashed lines indicate unsucessful searches.

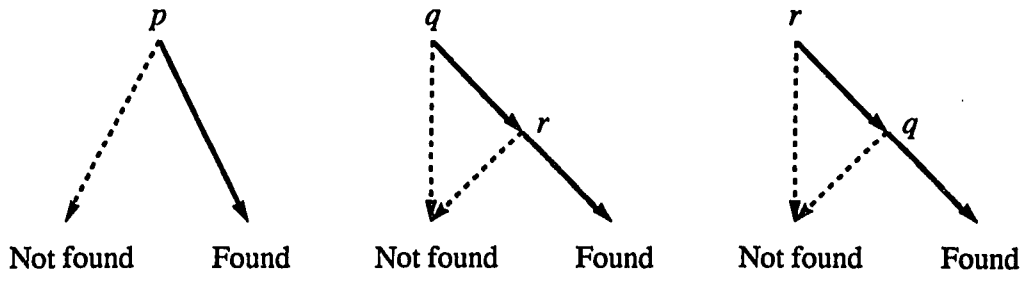

Figure 34: Methods of evaluation for $p = q \cdot r$

To determine the optimum pattern matching strategy, the evaluation algorithm first computes the value of the inherited predicate *IsApproximate*, and recursively evaluates the left and right subtrees (i.e., the subpatterns q and r). The algorithm then determines the values of the predicates *IsSpacer*, *IsKeyword*, *IsKeySet*, *IsNetwork*, *IsRegular* and the attributes *NumberOfKeywords*, f, d, and δ, according to the recurrences in Figure 35. Finally, the cost values $C_u(p,N)$, $C_l(p)$, $C_r(p)$, and $C_b(p)$ are computed according to the recurrences shown in Figure 36. In Figure 36, the three alternatives for each of the cost values correspond to the three methods of searching for the pattern p. Values for the cost constants c_u and c_a are taken as the appropriate minima from Figures 31 and 32, respectively, based on the values of the previously computed predicates. The expected cost of searching a database s of length N for the pattern p is then given by $C_u(p,N)$, and the optimum method of evaluation is the one that led to the minimum value for this cost.

$$
\begin{aligned}
IsSpacer(q \cdot r) &= IsSpacer(q) \wedge IsSpacer(r) \\
IsKeyword(q \cdot r) &= IsKeyword(q) \wedge IsKeyword(r) \\
NumberOfKeywords(q \cdot r) &= \begin{cases} NumberOfKeywords(q) \cdot \\ NumberOfKeywords(r) \end{cases} \\
IsKeySet(q \cdot r) &= \begin{cases} IsKeySet(q) \wedge IsKeySet(r) \wedge \\ NumberOfKeywords(q \cdot r) < T \end{cases} \\
IsNetwork(q \cdot r) &= IsNetwork(q) \wedge IsNetwork(r) \\
IsRegular(q \cdot r) &= IsRegular(q) \wedge IsRegular(r) \\
f(q \cdot r) &= f(q) \cdot f(r) \\
d(q \cdot r) &= d(q) + d(r) \\
\delta(q \cdot r) &= \delta(q) + \delta(r)
\end{aligned}
$$

Figure 35: Predicate and attribute recurrences for $p = q \cdot r$

$$
\begin{aligned}
C_u(q \cdot r, N) &= \min \begin{Bmatrix} c_u(q \cdot r, N) \\ C_u(q, N) + f_q \cdot N \cdot C_l(r) \\ C_u(r, N) + f_r \cdot N \cdot C_r(q) \end{Bmatrix} \\
C_l(q \cdot r) &= \min \begin{Bmatrix} c_a(q \cdot r) \\ C_l(q) + f_q \cdot C_l(r) \\ C_u(r, \delta(q)) + f_r \cdot \delta(r) \cdot C_b(q) \end{Bmatrix} \\
C_r(q \cdot r) &= \min \begin{Bmatrix} c_a(q \cdot r) \\ C_u(q, \delta(r)) + f_q \cdot \delta(q) \cdot C_b(r) \\ C_r(r) + f_r \cdot C_r(q) \end{Bmatrix} \\
C_b(q \cdot r) &= \min \begin{Bmatrix} c_a(q \cdot r) \\ C_l(q) + f_q \cdot C_l(r) \\ C_r(r) + f_r \cdot C_r(q) \end{Bmatrix}
\end{aligned}
$$

Figure 36: Predicate and attribute recurrences for $p = q \cdot r$

Back References

The evaluation algorithm described in the previous section computes $C_u(p,N)$, the minimum cost of searching a database s of length N for a pattern p, assuming that p does not contain any back references. This section explains how the evaluation algorithm needs to be modified to accomodate back references.

The PAMALA code fragment in Figure 37 shows a simple pattern, consisting of the concatenation of two occurrences of the same back reference. In the example, r is an arbitrary pattern.

```
reference r = .....;
pattern   p = r r;

evaluate contains(p,"database",PLAIN,ALL);
```

Figure 37: PAMALA declaration for the pattern $p = r \cdot r$

The definition of back reference patterns given in the chapter on pattern notation permits two orders of evaluation, or *permutations*, for the pattern in the example above, namely $p_1 = r^F r^B$ and $p_2 = r^B r^F$. The first permutation corresponds to a left-to-right backtracking order, i.e., the leftmost occurrence of r is searched for first (using whatever technique is appropriate for r), and then a left-anchored keyword search is performed for the string matched by r. The second permutation reverses that order of evaluation, i.e., it first searches for the rightmost occurrence of r, and then performs a right-anchored keyword search for whatever string was matched by r. The two permutations of the pattern can also be thought of as parse trees, with directional restrictions of the form $\rightarrow$ (left-to-right) and $\leftarrow$ (right-to-left) on the backtracking order added to the topmost node, as illustrated in Figure 38.

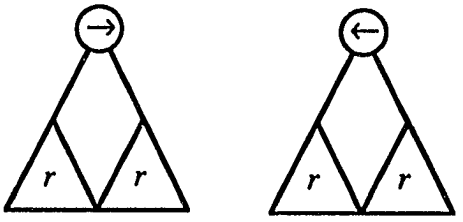

Figure 38: Restricted parse trees for the patterns $p_1 = r^F r^B$ and $p_2 = r^B r^F$

The parse trees shown in Figure 38 can be evaluated with the algorithm described in the previous section, provided that the directional restriction on the topmost node is honored. In other words, instead of considering the various pattern matching algorithms and backtracking orders at the restricted node, the evaluation algorithm simply needs to accept the directional restriction as the optimum method for this particular node. Hence, for the example above, the minimum cost of searching for the overall pattern p, together with the optimum method of evaluation, can be determined by evaluating the two restricted parse trees separately, and selecting the one that yields the lesser cost.

In general, if a pattern p contains k different back references, each of which occurs n_i times, $1 \le i \le k$, there are $K = \prod_{i=1}^{k} n_i$ theoretically possible orders of evaluating p. This function leads to exponential growth in the number of permutations. Aho [1980] has shown that the problem of whether a regular expression p with back references matches a string s is *NP*-complete, by giving a reduction from the vertex cover problem. This result suggests that the worst-case time complexity for matching a pattern with back references is inherently exponential. In practice, however, both k and n_i are quite small, and acceptable performance can be achieved by using a backtracking approach. For the example above, $k = 1$, $n_1 = 2$, and $K = 2$. Figure 59 shows that one of the more complex biological patterns, the tRNA gene, can be described with a PAMALA pattern for which $k = 4$ and $n_i = 2$, $1 \le i \le 4$, for a total of $K = 16$ theoretically possible permutations.

The number of theoretically possible permutations is, however, reduced by the way in which the evaluation algorithm considers the decomposition of patterns. Since the algorithm performs a depth-first traversal of the parse tree, the evaluation of the entire left subtree of a binary node must have been completed before the evaluation of the right subtree can begin (or vice versa). Hence, some theoretically possible permutations are not considered by the algorithm. A *legal permutation* is defined as one that can be generated by traversing the parse tree in a depth-first, but otherwise arbitrary order, and labeling as free the first occurrence of each back reference pattern encountered during the traversal, and labeling as bound all other occurrences of the same pattern.[16]

The effect of the depth-first traversal of the parse tree performed by the evaluation algorithm on the number of (legal) permutations, K', is illustrated by the pattern $p = (r \cdot s) \cdot (s \cdot r)$ defined in Figure 39, and the corresponding parse tree shown in Figure 40. Again, r and s are arbitrary patterns. There exist $K = 4$ theoretically possible permutations, namely $r^F s^F s^B r^B$, $r^F s^B s^F r^B$, $r^B s^F s^B r^F$, and $r^B s^B s^F r^F$. However, the above order of evaluation only permits the first and the last of those, hence, $K' = 2$ for this particular example. Figure 41 shows the restricted parse trees for the patterns $r^F s^F s^B r^B$ and $r^B s^B s^F r^F$.

---

16 This definition of a legal permutation explains why the boolean expression on the right-hand side of a conditional pattern may refer to only those back references that also occur on the left-hand side. Since the use of a back reference pattern within such an expression is equivalent to a bound occurrence of the back reference at the same position, the resulting parse tree must correspond to a legal permutation. This, however, can only be guaranteed with the above restriction. Assuming that p and q are two arbitrary back reference patterns, the pattern $(p \setminus f(p,q)) \cdot (q \setminus g(p,q))$ is an example of a pattern that violates the above condition, and for which no legal permutation exists.

```
reference r = .....;
reference s = .....;
pattern   p = (r s) (s r);

evaluate contains(p,"database",PLAIN,ALL);
```

Figure 39: PAMALA declaration for the pattern $p = (r{\cdot}s){\cdot}(s{\cdot}r)$

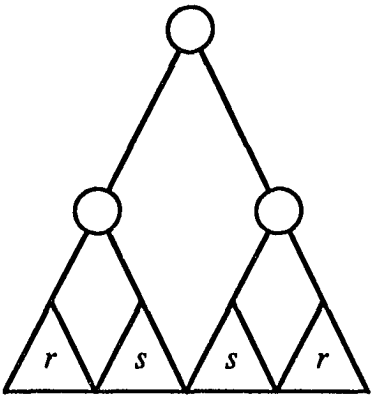

Figure 40: Parse tree for the pattern $p = (r{\cdot}s){\cdot}(s{\cdot}r)$

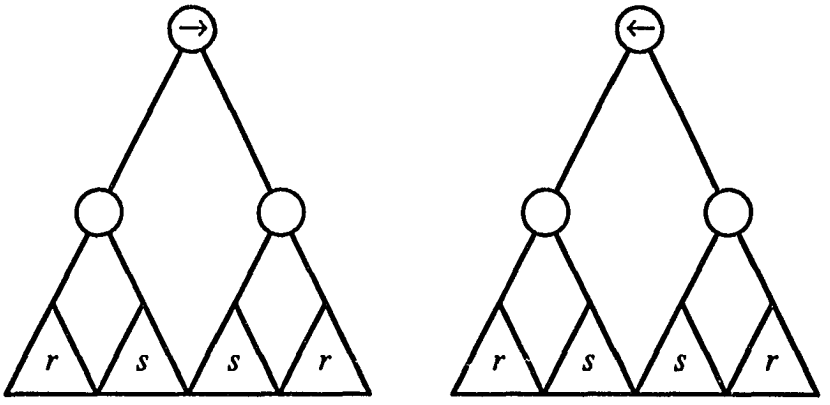

Figure 41: Restricted parse trees for patterns $p_1 = r^F s^F s^B r^B$ and $p_2 = r^B s^B s^F r^F$

In general, therefore, a pattern containing back references can be evaluated by generating all K' legal permutations and evaluating each of the corresponding restricted

parse trees separately. The method of evaluation (and its cost) for the overall pattern is then given by the permutation with the least cost.

Generating the K' legal permutations can be done by marking a subset of the parse tree's nodes, and enumerating all possible combinations of directional restrictions for those nodes. The nodes to be marked are determined as follows. For each node of the parse tree, the set of back reference patterns that occur anywhere in the tree rooted at that node is determined. Every node that has one or more occurrences of the same back reference pattern in both its left and its right subtree is marked. Figure 42 shows the marked parse tree for the pattern $p = (r \cdot s) \cdot (s \cdot r)$, together with the sets of back reference patterns at each node.

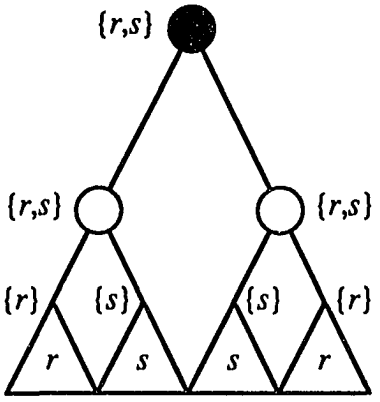

Figure 42: Marked parse tree for the pattern $p = (r \cdot s) \cdot (s \cdot r)$

Figure 43 translates this informal description into a recursive algorithm, which marks the tree in a single, depth-first traversal of the tree. The procedure takes a node p from the parse tree as its argument. It initializes the attribute *labels* (a set containing the labels of all back references that occur anywhere in the subtree rooted at p), calls itself recursively on the children of p, and adds the children's *labels* to its own. The algorithm then sets the

value of the node's predicate *marked* to *true* if the intersection of the children's *labels* is non-empty, and to *false* otherwise.

```
MarkTree (p)

    if ¬IsBackReference(p) then
        p.labels ← ∅
    else
        p.labels ← {p}
    end
    if p.lftSubtree ≠ Λ then
        MarkTree(p.lftSubtree)
        p.labels ← p.labels ∪ p.lftSubtree.labels
    end
    if p.rgtSubtree ≠ Λ then
        MarkTree(p.rgtSubtree)
        p.labels ← p.labels ∪ p.rgtSubtree.labels
    end

    if p.lftSubtree ≠ Λ and p.rgtSubtree ≠ Λ and
      p.lftSubtree.labels ∩ p.rgtSubtree.labels ≠ ∅ then
        p.marked ← true
    else
        p.marked ← false
    end
end
```

Figure 43: Algorithm for marking parse tree

In Figure 43, Λ stands for a nonexisting subtree, and ∅ represents the empty set. The predicate *IsBackReference* assumes the value *true* if its argument is a back reference pattern, and the value *false* otherwise. It remains to be shown that all legal permutations of a pattern can indeed be generated by enumerating all possible directional restrictions

on the nodes marked by the algorithm in Figure 43.[17] The following is a proof by induction.

Assumption. Given the parse tree of a pattern that contains back reference subpatterns, all legal permutations of the overall pattern can be generated by marking those binary nodes in the parse tree which have one or more occurrences of the same back reference pattern in both their left and right subtrees, enumerating all possible combinations of left-to-right and right-to-left restrictions for the marked nodes, traversing each such parse tree in a depth-first order that is consistent with the assigned restrictions, and labeling as free the first encountered occurrence of each back reference, and labeling as bound all other occurrences of the same back reference.

Basis. It needs to be shown that the assumption above is true for an atomic back reference pattern, i.e., a tree consisting of a single leaf. Since there is only one occurrence of the back reference pattern, the only legal permutation is the one that labels this occurrence (i.e., the leaf) as free. Since there is only one possible way of traversing the "tree," and this traversal necessarily labels the leaf as free, no restrictions are required, and hence the leaf must not be marked. □

Induction. Assuming that the assumption is true for every subtree of an interior node, it needs to be shown that it also holds for the node itself. The case of a unary node is trivial. Since adding a unary node to the top of an existing tree does not result in any additional legal permutations, no further restrictions are required, and hence the node must not be marked. In the case of a binary node, there are two cases — (1) there are no back reference patterns that occur in both the left and right subtree, and (2) there exist one

---

[17] However, the same (legal) permutation may be generated by more than one combination of directional restrictions on the set of marked nodes.

or more back reference patterns that occur in both the left and right subtree. In the first case, any permutation of the entire tree that is generated according to the rules above is the same, regardless of the order in which the subtrees are evaluated. Hence, the node must not be marked. In the second case, all legal permutations over the entire tree can be generated by concatenating a legal permutation of the left subtree with a legal permutation of the right subtree, and relabeling as bound either the rightmost or the leftmost free occurrence of those back references that occur in both the left and the right subtree. Since these are exactly the permutations that are generated by evaluating the tree twice, once with a left-to-right restriction and once with a right-to-left restriction, the node must be marked. □

In general, the set of marked nodes can be partitioned into one or more mutually disjoint subsets based on the *labels* attributes of the individual nodes. Each subset consists of all nodes that have at least one back reference in common, and corresponds to a subtree of the overall pattern. The root of such a subtree is the (only) node in the set that does not have a marked ancestor. These *marked subtrees* are mutually disjoint, and hence the legal permutations for one marked subtree can be generated independently of the others.

Figures 44 and 45 show an algorithm that generates, with the minimum number of restrictions, all legal permutations of a single marked subtree, without generating any permutation more than once. The algorithm backtracks over the marked subtree and puts directional restrictions on some of the marked nodes by assigning either $\rightarrow$ or $\leftarrow$ as the value of the node's *method* attribute. When traversing the marked subtree, the algorithm keeps track of all the back references that have been encountered, and the restrictions that have been assigned. This is done with two global stacks that are maintained in parallel.

The first stack, called *Nodes*, contains the nodes of the tree that have been restricted. Nodes are put on the stack in the order in which they are encountered during the traversal. The second stack, called *Labels*, contains, for each node on the first stack, the set of labels of those back references for which the free occurrence has been assigned. At any given time, the combination of these two stacks therefore represents a partial assignment of directional restrictions, together with the set of those back references for which those restrictions have resulted in the assignment of the free occurrence.

```
int Top ← 0
labelSet Labels[...]; Labels[0] ← ∅
node Nodes[...]

InitPermutations (p)

    if IsBackReference(p) then
        if p ∉ Labels[Top] then
            p ← p^F
            Labels[Top] ← Labels[Top] ∪ p
        else
            p ← p^B
        end
    end

    if p.marked and p.labels — Labels[Top] ≠ ∅ then
        p.method ← →
        Top ← Top + 1
        Nodes[Top] ← p
        Labels[Top] ← Labels[Top–1]
    end

    if p.lftSubtree ≠ Λ then
        InitPermutations(p.lftSubtree)
    end
    if p.rgtSubtree ≠ Λ then
        InitPermutations(p.rgtSubtree)
    end
end
```

Figure 44: Algorithm for generating legal permutations (initialization)

The initialization step of the algorithm is shown in Figure 44. The argument to the function is a node of the marked subtree. When processing this node, the algorithm first determines whether or not it represents a back reference pattern. If it does, and that particular back reference has not been seen before, the node is labeled as free, and the label of the back reference pattern is added to the set of labels on top of the stack. If the node represents a back reference pattern that has already been seen, it is labeled as bound.

The algorithm then determines whether there exist any back references in the tree below the node being processed that have not been seen before. This is done by comparing the node's *labels* attribute to the set of labels on the stack. Only if such a back reference exists is the node assigned a restriction of the form $\rightarrow$ and put on the stack. This technique ensures that only the absolutely necessary number of restrictions is assigned. The algorithm then calls itself recursively on the left and right subtrees of the node. Since all directional restrictions assigned by the algorithm are of the form $\rightarrow$, the generated restricted parse tree corresponds to the permutation that has the leftmost occurrence of each back reference labeled as free, and all other occurrences labeled as bound.

```
boolean MorePermutations (p)

    while Top > 0 and Nodes[Top].method = ← do
        Top ← Top – 1
    end

    if Top > 0 then
        p ← Nodes[Top]
        p.method ← ←
        Labels[Top] ← Labels[Top–1]
        if p.rgtSubtree ≠ Λ then
            InitPermutations(p.rgtSubtree)
        end
        if p.lftSubtree ≠ Λ then
            InitPermutations(p.lftSubtree)
        end
        return true
    else
        return false
    end
end
```

Figure 45: Algorithm for generating legal permutations (increment)

The remaining restricted parse trees are generated with the algorithm shown in Figure 45. Given a restricted parse tree, the algorithm generates the following one by processing nodes on the stack from top to bottom. Nodes are popped from the stack as long as their restrictions are of the form $\leftarrow$. Once the algorithm encounters a restriction of the form $\rightarrow$, its value is changed to $\leftarrow$, and the nodes below it are reinitialized (set to $\rightarrow$ and pushed on the stack) from right to left. This technique has the effect of slowly moving the free occurrences of the back references from left to right. The function returns a value of *true* for each new restricted parse tree, and a value of *false* after the last restricted parse tree has been generated. The last tree contains only restrictions of the form $\leftarrow$, and corresponds to the permutation that has the rightmost occurrence of each back reference labeled as free, and all other occurrences labeled as bound.

The complete algorithm for evaluating a general pattern (with or without back references) is shown in Figure 46. The algorithm traverses the parse tree in depth-first order, and evaluates the pattern as described in the previous section. Whenever the algorithm reaches the root of one of the marked subtrees, it records this fact by setting the global boolean variable *Outside* to *false*. The algorithm then generates and evaluates all legal permutations of the subpattern represented by the subtree, and retains the subtree that achieved the lowest cost. After the last permutation has been generated and evaluated, the algorithm continues its depth-first evaluation of the overall pattern.

```
boolean Outside ← true

EvaluatePattern (p)

    if ¬p.marked or ¬Outside then
        compute inherited predicates and attributes
        if p.lftSubtree ≠ Λ then
            EvaluatePattern(p.lftSubtree)
        end
        if p.rgtSubtree ≠ Λ then
            EvaluatePattern(p.rgtSubtree)
        end
        compute synthesized predicates and attributes
        compute Cu(p,N), Cl(p), Cr(p), Cb(p) and retain optimum methods
    else
        Outside ← false
        InitPermutations(p)
        EvaluatePattern(p)
        q ← CopyTree(p)
        while MorePermutations(p) do
            EvaluatePattern(p)
            if C(q) > C(p) then
                q ← CopyTree(p)
            end
        end
        p ← q
        Outside ← true
    end
end
```

Figure 46: Complete pattern evaluation algorithm[18]

Refinements

There are several refinements to the pattern evaluation algorithm described in the previous sections of this chapter that can improve the performance of the algorithm.

---

[18] To simplify the treatment, this formulation of the algorithm ignores the fact that the evaluation algorithm needs to keep different copies for the unanchored, left-, right-, and both-anchored cost values when evaluating all legal permutations of a subtree. Once the entire backtracking order has been determined, however, the unneeded copies can be discarded.

These refinements either improve the accuracy of the cost estimate, and or they reduce the time spent evaluating and searching for the pattern.

To determine the match frequency of approximate patterns, the evaluation algorithm relies on a Monte-Carlo simulation. This is a necessary compromise in light of the fact that the match frequency can not be determined analytically in an efficient way. However, Monte-Carlo simulation is computationally expensive, especially if a high degree of accuracy is desired. While this time may be well spent for a large database, it may not be worth the effort if the database is relatively short. The solution to this dilemma is to use an adaptive approach. Initially, only a rough estimate of the match frequency is computed. This estimate is then refined over time by accumulating statistics as the search of the database progresses. Periodically, the pattern is reevaluated, using the refined value in place of the initial estimate. This approach has the advantage that only little time is spent during the initial evaluation. For short databases, the resulting estimate is entirely sufficient. For longer databases, the estimate is refined and the backtracking order is fine-tuned during the search. Disadvantages of this strategy include the overhead for acquiring the statistics during the course of the search, and the time spent for the occasional reevaluations of the pattern. However, both of these factors turn out to be negligible, compared to the amount of time required for the search process itself.

A similar problem occurs for the boolean expression on the right-hand side of a conditional pattern, where neither the match frequency nor the cost of evaluation can be determined analytically or through simulation. The problem can be solved by using the same adaptive approach as described above, i.e., by initializing the match frequency and cost of evaluation to one and zero, respectively, and refining these "estimates" by gathering statistics during the search process.

A different problem occurs when evaluating patterns that contain back references. Whenever the algorithm in Figure 46 generates a new permutation for some marked subtree of the pattern, the entire subtree is reevaluated. A more efficient method is to only reevaluate the portion of the subtree for which the assignment of restrictions has actually changed, and to then propagate those changes from the root of the modified portion up to the root of the marked subtree.

Finally, the evaluation algorithm as described in the previous section may result in the assignment of a single pattern matching algorithm for the free occurrence of a back reference subpattern and a neighboring subpattern. In such a case, the string matched by the back reference pattern must be extracted from the string matched by the combined pattern, so that it is available for use with bound occurrences. This is, however, not an easy process in general. Instead, the algorithm prevents the assignment of a single pattern matching algorithm by computing another predicate, called *CanDirect*, for each node of the tree. The value of this predicate is *true* at a particular node, if the subpatterns represented by the children of the node do not contain any free occurrences of back references, and *false* otherwise. By adding this predicate to the condition for each of the cost constants in Figures 31 and 32, the algorithm is prohibited from combining the free occurrence of a back reference subpattern with neighboring subpatterns.

CHAPTER 8: PATTERN MATCHING

An large number and variety of pattern matching algorithms has been developed over the past few decades. These algorithms can be characterized based on the class of patterns which they match, by their performance with respect to both space and time requirements, and whether they perform exact or approximate pattern matching (and, in the latter case, by the underlying model). A detailed review of many of these algorithms is given by, e.g., Aho [1990]. Several authors have attempted to compare the performance of different algorithms with the apparent goal of selecting a single best algorithm, but the results have generally remained inconclusive (e.g., [Horspool, 1980; Smit, 1982; Davies and Bowsher, 1986; Hume, 1988]). Hence, a small set of practical and efficient algorithms was chosen for the pattern matching system described in this dissertation. This library of pattern matching functions consists of six algorithms for matching spacers, keywords, sets of keywords, regular expressions, approximate regular expressions, and approximate networks. All other patterns supported by the system are matched by suitably decomposing the overall pattern and searching separately for individual subpatterns, using the backtracking approach described in the preceeding chapter.

The following sections briefly review the implemented algorithms and discuss their respective advantages and disadvantages. References containing a detailed treatment are provided for all of the chosen algorithms, and for a large number of alternatives as well. The algorithms are analyzed with respect to both space and time requirements, using the familiar notations for upper bounds (O), and both lower and upper bounds (Θ). The analysis distinguishes between preprocessing steps (such as computing shift tables for the Boyer-Moore algorithm, or transition tables for the Aho-Corasick algorithm) and execution of the algorithm proper. Since the time required for preprocessing is generally

negligible, compared to the time spent for searching a large database, preprocessing is not included in any performance comparisons between different algorithms. The analysis further assumes that the alphabet size is a small constant in the spirit of the O notation. This is a reasonable assumption, and certainly holds for biological applications, where the size of the nucleic acid alphabet is four, and the size of the protein alphabet is twenty characters.

In addition to the worst-case analysis, the following sections also discuss the expected-time performance of each algorithm. This analysis, which is reflected in the formulae in Figures 31 and 32, forms the basis for computing the estimated cost of searching a database for a composite pattern via the recurrences in Figures 27 through 30. The accuracy of the cost models for the various algorithms is discussed in the chapter on results.

The algorithms as described below perform an unanchored search over an entire database; adapting the algorithms for an anchored search (where a match must begin at a specific location of the database) is a straightforward task.

Spacers

Given a spacer $p = \langle m,n \rangle$, $m \leq n$, of length $M = m-n$, and a string s ("database") of length N, the algorithm needs to locate all occurrences of p in s. Since spacers match any character in the database, the algorithm itself is trivial and is given here only to justify the analysis below. The algorithm shown in Figure 47 assumes a single, contiguous interval; multiple, disjoint intervals are handled by the backtracking mechanism described previously.

```
for i ← 0 to N do
    for j ← m to n do
        if 0 ≤ i+j and i+j ≤ N then
            write "match found between positions", i, "and", j
        end
    end
end
```

Figure 47: Spacer algorithm

The algorithm requires a constant amount of space, i.e., $O(1)$, to store the interval limits m and n, and its time complexity is $\Theta(MN)$.

Each of the algorithms described in the remaining sections of this chapter outputs the locations of the right endpoint of every match to the pattern. The corresponding left endpoints can be found readily by performing an anchored search in the opposite direction, beginning at the known right endpoints.

Keywords

Given a fixed-string pattern p ("keyword") of length M, and a string s of length N, the algorithm needs to locate all occurrences of p in s. A straightforward approach would align the left endpoints of pattern and database and then repeatedly compare pattern and database from left to right, shifting the pattern one character to the right whenever a mismatch occurs. This method requires time $O(MN)$ in the worst case. Several algorithms have been proposed that improve on the straightforward approach [Boyer and Moore, 1977; Knuth *et al.*, 1977; Karp and Rabin, 1981]. While they all share the same worst-case time complexity, namely $O(N)$, the method of Boyer and Moore is generally preferred because of its superior performance in practice. Hence, PAMALA uses an

implementation of the Boyer-Moore algorithm, which is described below. The treatment follows Aho [1990].

The Boyer-Moore algorithm looks for a match by moving the pattern along the database from left to right, and comparing characters from right to left. Initially, the left endpoints of pattern and database are aligned, and comparison begins at the right end of the pattern, one character at a time. If the comparison successfully reaches the left end of the pattern, a match is reported. Then (or if a mismatch occurs before the left end of the pattern has been reached) the pattern is shifted to the right and comparison starts anew. The amount of the shift depends on the pattern and the character causing the mismatch. Figure 48 summarizes the Boyer-Moore algorithm.

```
j ← M
while j ≤ N do
    i ← M
    while i > 0 and p_i = s_j do
        i ← i − 1
        j ← j − 1
    end
    if i = 0 then
        write "match found at position", j
    end
    j ← j + max{d1[sj], d2[i]}
end
```

Figure 48: Boyer-Moore algorithm

In Figure 48, tables d_1 and d_2 determine how far the pattern can be shifted to the right. The first table, d_1, is indexed by characters. For every character c, $d_1[c] = M–i$, where i is the position of the rightmost occurence of c in p, or $d_1[c] = M$ if c does not occur in p. The second table, d_2, is indexed by pattern position, and gives the minimum shift g such that when p_m is aligned above s_{k+g}, the substring $p_{i+1-g} \ldots p_{m-g}$ of the pattern agrees with

the substring $s_{k-m+i+1} \ldots s_k$ of the database, assuming p_i did not match s_{k-m+i}. Both tables can be precomputed in a single pass over the pattern, i.e., in time $O(M)$.

The algorithm requires $O(M)$ space to store the pattern p and shift tables d_1 and d_2. The worst-case time complexity of the Boyer-Moore algorithm is $O(N)$, and thus equivalent to the methods of Knuth *et al.*, and Karp and Rabin. On average, however, the Boyer-Moore algorithm only requires $O(N/M)$ comparisons, i.e., its performance is sub-linear in expectation. This behavior is explained by the fact that, for large alphabets and relatively small patterns, the mismatch often occurs at or near the right end of p, and is caused by a character that does not occur in p; and hence p can be shifted M positions to the right.

Sets of Keywords

Given a pattern p consisting of a set of fixed strings $p_1, \ldots, p_k$ ("set of keywords") with combined length M, and a database s of length N, the algorithm needs to locate all occurrences of any p_i in s. The straightforward method for recognizing sets of keywords would scan the database with a single-keyword algorithm separately for each of the keywords, resulting in a worst-case time complexity of $O(M+kN)$. The method by Aho and Corasick [1975] improves on this result by searching for all of the keywords simultaneously.

The Aho-Corasick algorithm utilizes a data structure called a trie [Aho *et al.*, 1983]. A trie of a set of keywords is a tree whose edges are labeled with single characters such that the concatenation of labels along every path from the root to a leaf spells a keyword from the set. Figure 49 shows the trie for the set of keywords (*aaba* | *abb* | *abcb* | *abcd*).

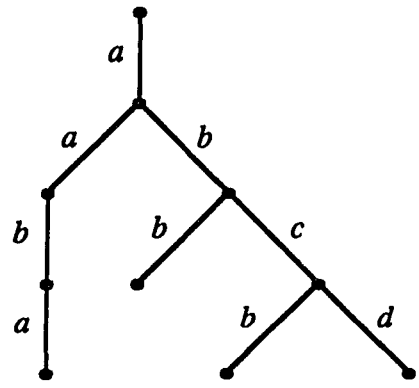

Figure 49: Trie for the set of keywords (*aaba* | *abb* | *abcb* | *abcd*)

The trie is used, in conjunction with two transition tables, as a pattern matching automaton. The root of the trie is designated as the start state q_0, and a set F of trie nodes are labeled as final states. Matching begins with the automaton in the start state. Characters are read in and processed one at a time. Depending on the pattern and the character read, the automaton makes a series of failure transitions, and a single forward transition. If, after the forward transition, the automaton is in one of the final states, a match is reported. The algorithm is summarized in Figure 50.

```
q ← q0
for j ← 1 to N do
    while g[q,sj] = fail do
        q ← h[q]
    end
    q ← g[q,sj]
    if q ∈ F then
        write "match found at position", j
    end
end
```

Figure 50: Aho-Corasick algorithm

In Figure 50, q represents the current state, and g and h denote the forward and failure transition tables, respectively. Both tables can be computed in a single preprocessing pass over the pattern, i.e., in time $O(M)$.

The algorithm requires $O(M)$ space to build the trie and to hold the transition tables. It achieves a time complexity of $O(N)$ in expectation, and $O(M+N)$ in the worst case. Commentz-Walter [1979] has proposed combining the ideas of the Boyer-Moore and the Aho-Corasick algorithms, i.e., building a trie for the keywords reversed, and matching characters from right to left. While this version of the algorithm runs faster than the Aho-Corasick method for small numbers of keywords, its worst-case running time is $\Theta(MN)$, and hence inferior to the Aho-Corasick algorithm.

Regular Expressions

Given a regular expression p of size M, and a database s of length N, the algorithm needs to locate every occurence in s of any string matched by p. Traditionally, two approaches have been used — simulation of all possible execution paths of the nondeterministic finite automaton (NFA) corresponding to the regular expression [Thompson, 1968], or simulation of the single execution path of the deterministic finite automaton (DFA) derived from the pattern [McNaughton and Yamada, 1960]. For practical pattern matching, however, each of these algorithms has a serious drawback — the NFA approach results in a worst-case running time of $O(MN)$, while the DFA algorithm can require $O(2^M)$ space and $O(2^{M+N})$ time for some regular expressions. PAMALA uses a hybrid approach described by Aho [1980], which avoids these disadvantages.

Aho's algorithm constructs from p an NFA with ε-transitions, a start state q_0, and a single final state f, as described by, e.g., Aho *et al.* [1974]. In addition, a cache C is created for holding the most frequently used sets of states. The algorithm then simulates the NFA by computing the set of states Q the NFA could be in after having read a prefix of s. Initially, the state set Q contains the start state and all states reachable from the start state via ε-transitions. After reading a character, the algorithm first determines whether the set of states reachable from Q on that character has already been computed. If that is the case, it can be obtained from the cache in constant time. Otherwise, the state set is computed and added to the cache. A match is reported whenever Q contains the final state f. Figure 51 summarizes the algorithm.

```
Q ← epsilon({q0})
if f ∈ Q then
    write "match found at position", 0
end
for j ← 1 to N do
    if C[Q,sj] is undefined then
        C[Q,sj] ← epsilon(goto(Q,sj))
    end
    Q ← C[Q,sj]
    if Q ∉ C then
        if C is full then
            empty C
        end
        add Q to C
    end
    if f ∈ Q then
        write "match found at position", j
    end
end
```

Figure 51: Regular expression algorithm

In Figure 51, the function *goto* computes all states that are reachable from a given set of states on a specific input character, and the function *epsilon* computes all states that are reachable from a given set of states via ε-transitions. The ε-NFA can be computed in a single preprocessing pass over the pattern, i.e., in time $O(M)$.

The theoretical worst-case time complexity of the hybrid algorithm remains $O(MN)$. In practice, however, the algorithm closely approximates the $O(N)$ time efficiency of the DFA method, while retaining the $O(M)$ space efficiency of the NFA approach [Aho, 1980].

Approximate Regular Expressions

Given an approximate regular expression p of size M, together with a threshold t and a database s of length N, the algorithm needs to locate every substring of s for which the score of the highest-scoring alignment with any string matched by p equals or exceeds the threshold t. Myers and Miller [1989] developed an algorithm based on a combination of the ideas of NFA simulation [Thompson, 1968] and the dynamic programming algorithm for aligning two sequences [Sankoff, 1972]. This algorithm is preferable over a similar one by Wagner and Seiferas [1978], which limits scores to integer values and is not easily adapted to database searches.

From the regular expression p, Myers and Miller build a state-labeled NFA with ε-transitions as described by, e.g., Aho *et al.* [1974]. The initial state of the NFA is labeled as q_0, and the final state as f. The NFA, which is treated as a directed graph, is then used to construct a so-called *regular expression edit graph* G, where edges correspond to the insertion, deletion, and substitution operations that were defined in the chapter on pattern notation. To construct G, each row of the sequence comparison graph

of Sankoff is replaced with a copy of the NFA. Thus, the transition edges of the NFA become the deletion edges of the edit graph. Substitution and insertion edges are inserted between rows of G as appropriate. A vertex v in row i (written as v_i) is labeled $\lambda(v) \in \Sigma \cup \{\varepsilon\}$, according to the state of the NFA to which it corresponds. A deletion edge from a vertex v_i to some other vertex w_i in the same row is labeled as $\begin{bmatrix} \lambda(w) \\ \varepsilon \end{bmatrix}$, and has associated with it a score $\sigma(\lambda(w),\varepsilon)$. Similarly, an insertion edge from a vertex v_i to the same vertex v_{i+1} in the following row is labeled as $\begin{bmatrix} \varepsilon \\ s_{i+1} \end{bmatrix}$, and has an associated score of $\sigma(\varepsilon,s_{i+1})$. A substitution edge from a vertex v_i to some other vertex w_{i+1} in the following row is labeled as $\begin{bmatrix} \lambda(w) \\ s_{i+1} \end{bmatrix}$, and has an associated score of $\sigma(\lambda(w),s_{i+1})$. Figure 52 shows the regular expression edit graph for the pattern $p = a(a|b)c^*$ and the database $s = abcc$. In Figure 52, ε-labeled vertices are shown as small, solid circles. All edges $v \rightarrow w \in G$ that are not annotated in Figure 52 are labeled $\begin{bmatrix} \varepsilon \\ \varepsilon \end{bmatrix}$. An edge is called a forward edge if it is directed from left to right (or top to bottom), and called a backward edge if it is directed from right to left.

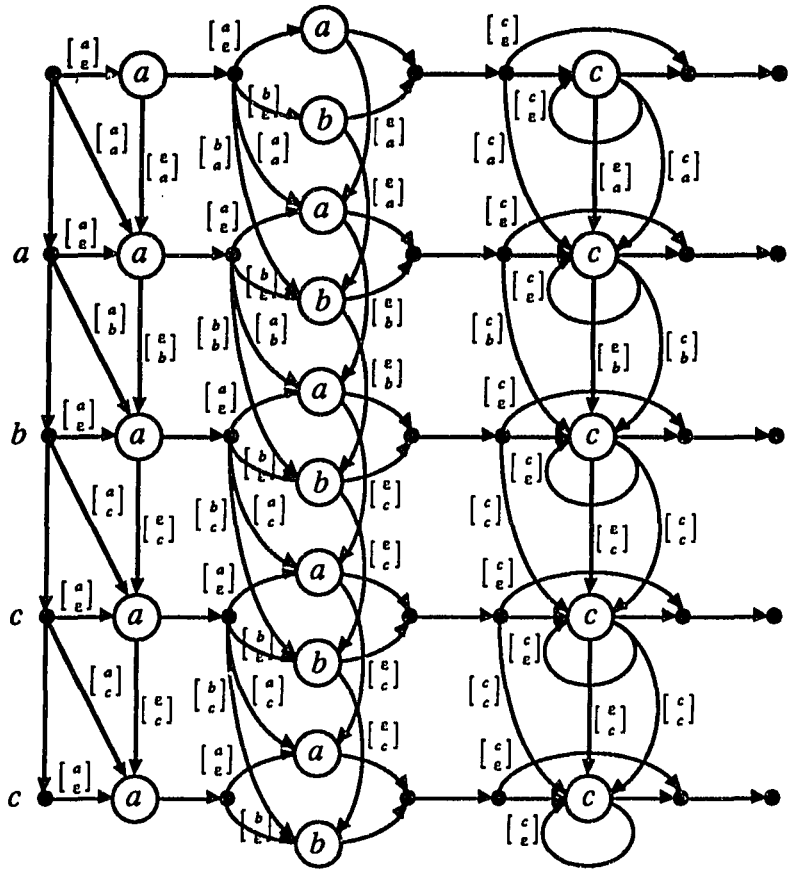

Figure 52: Regular expression edit graph for $p = a(a|b)c^*$ and $s = abcc$

Myers and Miller then proceed to show that the concatenation of the edge labels along every path in G that begins at vertex q_0 in some row i and ends at vertex f in row j, $0 \leq i \leq j$, constitutes an alignment between some string matched by p and the substring $s_i \ldots s_j$ of the database, and that such a path exists for every possible alignment. Furthermore, the score of this alignment is given by the sum of the scores associated with the edges along the path. Hence, s_j is the right endpoint of an approximate match to p if and only if the maximum score among all paths starting at some vertex q_{0_i} and ending at vertex f_j, $0 \leq i \leq j$, equals or exceeds the threshold t. To compute this score, the algorithm associates with every vertex v_j of G a score value $C(j,v)$, which represents the maximum

score among all paths starting at some vertex q_{0_i} and ending at vertex v_j, $0 \le i \le j$. The algorithm then computes all score values $C(j,v)$ by treating the graph G as a dynamic programming matrix and traversing each row of G twice in topological order of the forward edges. A match is reported for database position j if the score $C(j,f)$ equals or exceeds the threshold t.[19] The algorithm is summarized in Figure 53.

$C(0,q_0) \leftarrow 0$
for $w \in V - \{q_0\}$ in topological order of the forward edges **do**
  $C(0,w) \leftarrow \max_{v \to w \in F} \{C(0,v)\} + \sigma(\lambda(w),\varepsilon)$
end
if $C(0,f) \ge t$ **then**
  write "match found at position", 0
end
for $j \leftarrow 1$ **to** N **do**
  $C(j,q_0) \leftarrow \max\{0, C(j-1,q_0) + \sigma(\varepsilon,s_j)\}$
  for $w \in V - \{q_0\}$ in topological order of the forward edges **do**
    $C(j,w) \leftarrow \max_{v \to w \in F} \{C(j,v)\} + \sigma(\lambda(w),\varepsilon)$
    if $\lambda(w) \ne \varepsilon$ **then**
      $C(j,w) \leftarrow \max\{C(j,w), C(j-1,w) + \sigma(\varepsilon,s_j), \max_{v \to w \in E} \{C(j-1,v)\} + \sigma(\lambda(w),s_j)\}$
    end
  end
  for $w \in V - \{q_0\}$ in topological order of the forward edges **do**
    $C(j,w) \leftarrow \max\{C(j,w), \max_{v \to w \in E \text{ and } v \ne q_0} \{C(j,v)\} + \sigma(\lambda(v),\varepsilon)\}$
  end
  if $C(j,f) \ge t$ **then**
    write "match found at position", j
  end
end

Figure 53: Myers-Miller algorithm

In Figure 53, V is the set of all vertices in any given row of G, E denotes the set of all edges coming into any vertex $v \in V$, and $F \subseteq E$ contains all forward edges from E. The

[19] To simplify the treatment, this formulation of the algorithm compares the threshold t to the absolute score of the alignment. The conversion from absolute to length-relative scores is straightforward.

sets V and E (i.e., the ε-NFA) can be computed in a single preprocessing pass over the pattern, i.e., in time $O(M)$.

The edit graph G consists of N+1 rows, each of which contains $O(M)$ vertices. The algorithm as presented above therefore requires $O(MN)$ space to hold the score values for the entire edit graph. However, since the scores in any given row depend only on scores from the same row and from the one preceding it, the space requirements can be reduced to $O(M)$ by retaining only the two "most recent" rows. Myers and Miller have shown that the number of incoming edges at any vertex of G is at most five. Hence, the score at each vertex can be computed in time $O(1)$, resulting in a worst-case time complexity of $O(MN)$ for the entire algorithm.

Approximate Networks

Given an approximate "network" p (a regular expression without Kleene closure) of size M, together with a threshold t and a database s of length N, the algorithm needs to locate every substring of s for which the score of the highest-scoring alignment with any string matched by p equals or exceeds the threshold t. Since networks are a subset of regular expressions, the problem can be solved with the algorithm by Myers and Miller described in the previous section. There does, however, exist a modification to this algorithm that reduces its time complexity to $O(tN)$ in expectation. The modified algorithm is due to Myers, and described in a forthcoming paper by Myers and Mehldau [1991]. A similar technique was first used in the algorithm described by Myers and Mount [1986] for finding approximate matches to keywords. The improvement is based on the observation that, for large values of t, the score values of only a small subset of the vertices in any given row of the edit graph equal or exceed t. Hence, the algorithm can be

made more efficient by computing the score values for only this subset of the vertices. However, the improvement requires that the edit graph does not contain any back edges. Since every back edge in the edit graph is caused by a Kleene closure in the pattern, the modified algorithm applies only to networks, but not to regular expressions.

The following discussion of the modified algorithm uses the same notation as the description of the Myers-Miller algorithm in the previous section. Instead of computing the score values for every vertex $v \in V$, the modified algorithm only computes the score values for a small set $T \subseteq V$. This "threshold set" is defined as follows. A vertex $v \in V$ is an element of T if either $C(j,v) \geq t$, or if there exists a vertex w, such that $v \rightarrow w \in E$, $w \in T$, and the removal of v would disconnect the set. The second part of the condition ensures that for every vertex $v \in T$, there exists a path beginning at q_0 and ending at v, such that all vertices along the path are in T. In addition to computing the score values for the vertices in T, the algorithm also needs to adjust T as the computation progresses from one row of the edit graph to the next. Initially, T contains all vertices $v \in V$, and hence the algorithm needs to compute the score values for the entire first row of the edit graph. Score values are calculated as in the algorithm by Myers and Miller. After the score values have been computed, the set T is "trimmed back" by processing all vertices in T in reverse topological order, and removing those that have a score value less than t, and whose removal does not disconnect the set. When processing the following row, the algorithm needs to compute new score values for all vertices in T, and then "grow" T by processing the vertices on the boundary in topological order. All vertices which can achieve a score value equal to or greater than t by deleting or substituting a single character are added to T. The set is then trimmed back again as described above, and the entire process is repeated for the following row. Figure 54 summarizes the algorithm.

$C(0,q_0) \leftarrow 0$
$T \leftarrow V - \{q_0\}$
for $w \in T$ in topological order **do**
  $C(0,w) \leftarrow \max_{v \rightarrow w \in E} \{C(0,v)\} + \sigma(\lambda(w),\varepsilon)$
end
for $v \in T$: $C(j,v) < t$ **and** $\forall w$: $v \rightarrow w \in E$: $w \notin T$ **or** ($\exists$ u: $u \in T$ **and** $u \rightarrow w \in E$)
  in reverse topological order **do**
  $T \leftarrow T - \{v\}$
end
if $C(0,f) \geq t$ **then**
  write "match found at position", 0
end
for $j \leftarrow 1$ **to** N **do**
  $C(j,q_0) \leftarrow \max\{0, C(j-1,q_0) + \sigma(\varepsilon,s_j)\}$
  for $w \in T$ in topological order **do**
    $C(j,w) \leftarrow \max_{v \rightarrow w \in E} \{C(j,v)\} + \sigma(\lambda(w),\varepsilon)$
    if $\lambda(w) \neq \varepsilon$ **then**
      $C(j,w) \leftarrow \max\{C(j,w), C(j-1,w) + \sigma(\varepsilon,s_j), \max_{v \rightarrow w \in E} \{C(j-1,v)\} + \sigma(\lambda(w),s_j)\}$
    end
  end
  for $w \notin T$: $\exists$ v: $v \in T$ **and** $v \rightarrow w \in E$ in topological order **do**
    $C(j,w) \leftarrow \max_{v \rightarrow w:\, v \in T \text{ and } v \rightarrow w \in E} \{C(j,v)\} + \sigma(\lambda(w),\varepsilon)$
    if $\lambda(w) \neq \varepsilon$ **then**
      $C(j,w) \leftarrow \max\{C(j,w), \max_{v \rightarrow w:\, v \in T \text{ and } v \rightarrow w \in E} \{C(j-1,v)\} + \sigma(\lambda(w),s_j)\}$
    end
    if $C(j,w) \geq t$ **then**
      $T \leftarrow T \cup \{w\}$
    end
  end
  for $v \in T$: $C(j,v) < t$ **and** $\forall w$: $v \rightarrow w \in E$: $w \notin T$ **or** ($\exists$ u: $u \in T$ **and** $u \rightarrow w \in E$)
    in reverse topological order **do**
    $T \leftarrow T - \{v\}$
  end
  if $C(j,f) \geq t$ **then**
    write "match found at position", j
  end
end

Figure 54: Network algorithm

In Figure 54, T is the threshold set computed as described above, V is the set of all vertices in any given row of G, and E denotes the set of all edges coming into any vertex $v \in V$. The sets V and E (i.e., the ε-NFA) can be computed in a single preprocessing pass over the pattern, i.e., in time $O(M)$.

Like the algorithm by Myers and Miller described in the previous section, the modified algorithm requires space $O(M)$ to retain two rows of the edit graph. Likewise, the worst-case time complexity of the algorithm remains $O(MN)$, since for a sufficiently small threshold t the set T is equivalent to V, and hence the algorithm must still compute the score values for all vertices in G. On average, however, T represents only a small portion of each row, and its size is proportional to t. The fact that G is a series-parallel graph is exploited by a sophisticated traversal technique, in order to guarantee that the set T can be maintained in time $O(t)$. Hence, the expected-time complexity for the modified algorithm is $O(tN)$.

Other Algorithms

In addition to the algorithms described or mentioned above, several other pattern matching algorithms were considered. This section gives a brief overview of these algorithms, and explains why they were not included in the library.

Suffix trees as described by, among others, Weiner [1973] and McCreight [1976], are tries of all suffixes that function as an index into the database to be searched. This index can be used to look up the position of any particular string or set of strings very efficiently. Given a pattern of length M, and a database of length N, locating a match only takes time $O(M)$. However, constructing the suffix tree requires $O(N)$ preprocessing time, and $O(N)$ space is needed for storing the tree. Hence, suffix tree algorithms are most

useful when the same database must be searched frequently and remains unchanged over long periods of time.

Several highly specific classes of patterns have received special attention from researchers. For example, numerous algorithms exist for locating immediately repeated occurrences of the same string, or "squares" ([Crochemore, 1981; Apostolico and Preparata, 1983; Main and Lorentz, 1984, 1985; Rabin, 1985]), as well as strings that read the same left to right and right to left, or "palindromes" ([Manacher, 1975; Galil, 1976; Seiferas and Galil, 1977; Galil and Seiferas, 1978]). Squares can be found with any of the above algorithms in time $\Theta(N\log N)$ and space $O(N)$, and locating palindromes requires both time and space $O(N)$. However, including one (or more) of those algorithms into the pattern matching system would require a much more detailed analysis of the pattern in order to correctly detect and process these special cases, and would not result in any performance gains.

An overwhelming number of algorithms have been developed for performing various kinds of approximate pattern matching, and the following is a partial list of only the most important contributions to the field: [Needleman and Wunsch, 1970; Wagner and Fischer, 1974; Hirschberg, 1975, 1977; Hunt and Szymanski, 1977; Wagner and Seiferas, 1978; Sellers, 1980; Hall and Dowling, 1980; Waterman and Smith, 1981; Waterman, 1984; Ukkonen, 1985a, 1985b; Myers, 1986; Landau and Vishkin, 1988; Galil and Giancarlo, 1988; Myers and Miller, 1988, 1989; Baeza-Yates and Gonnet, 1989; Tarhio and Ukkonen, 1990; Chang and Lawler, 1990; Galil and Park, 1990; Wu and Manber, 1991]. Unfortunately, most of these algorithms can not be used readily with the pattern matching system presented in this dissertation for one or the other of the following reasons. Some of the algorithms were developed for aligning two sequences of characters (or,

equivalently, for computing the longest common subsequences or the shortest edit script of two sequences) and are not easily adapted to database searches. Others only consider a subset of the deletion, substitution, and insertion edit operations that are required for applications in molecular biology. Still other algorithms permit only (small) integer cost values, or a small number of differences between pattern and database. Finally, most of the algorithms listed above are only capable of searching for fixed strings, but can not locate sets of keywords or regular expressions.

CHAPTER 9: EXAMPLES

This chapter illustrates the power and flexibility of the new pattern matching system by showing, for several patterns taken directly from the biological literature, how these patterns can be expressed with PAMALA in an intuitive and concise fashion. The following chapter discusses the performance and efficiency of the implementation with respect to the patterns presented below.

Stem-Loop Structures

A common biological pattern is the so-called stem-loop, or "hairpin," structure. This pattern consists of three connected sequences of five to ten nucleotides each. The first and last sequence are inverted repeats of each other, and are connected by hydrogen bonds between complementary nucleotides. These two sequences create the stem of the hairpin structure, and the intermediate sequence forms a loop at one end of the stem, as shown in Figure 55. The chain of nucleic acids is shown as a solid line, individual nucleotides are represented by small solid circles, and the hydrogen bonds between complementary nucleotides are shown as dashed lines.

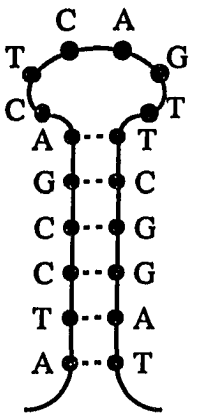

Figure 55: Stem-loop structure

Figure 56 shows a PAMALA program that defines such a stem-loop pattern. The program first establishes the nucleic acid (DNA) alphabet as the default, using the **alpha** and **score** statements. The scoring scheme itself need not be defined, since the program does not contain any approximate patterns. The pattern *HairPin* is built from two parts, appropriately called *Stem* and *Loop*, each of which is defined as a spacer of length five to ten nucleotides. The hairpin structure is obtained by defining *Stem* as a back reference pattern, and using this pattern in conjunction with the PAMALA notation for inverted repeats to create the structure shown in Figure 55. The **evaluate** statement then searches the file "Database" for all occurrences of this pattern.

```
alpha DNA   = [ACGT];
score none = DNA {};

reference Stem = <5,10>;
pattern   Loop = <5,10>;
pattern   HairPin = Stem Loop ~Stem;

evaluate contains(HairPin,"Database",PLAIN,ALL);
```

Figure 56: PAMALA definition of the stem-loop structure

The PAMALA definition in Figure 56 illustrates the power of the back reference construct. While the same pattern could theoretically be written as a regular expression of the form ($Stem_1$ <5,10> ~$Stem_1$ | $Stem_2$ <5,10> ~$Stem_2$ | ...) by enumerating all possible $Stem_i$, there are $(4^5+4^6+4^7+4^8+4^9+4^{10}) = 1{,}337{,}760$ combinations that would have to be listed.

α/β Protein Secondary Structure

Cohen *et al.* [1983] describe the so-called *TurnGen* procedure for decomposing an amino acid sequence into a set of non-overlapping segments, each of which contains at most one piece of secondary structure. These pieces are known individually as α-helices and β-strands, or, collectively, as α/β proteins. The *TurnGen* procedure of Cohen *et al.* is based on the assumption that a single piece of secondary structure consists of at least ten and at most eighteen proteins, and that consecutive pieces are separated by short, highly hydrophilic fragments, called turn segments. Figure 57 shows a complete PAMALA program that locates α/β proteins by performing a hierarchical decomposition of the amino acid sequence.

```
alpha protein = [ARNDCQEGHILKMFPSTWYV];
score none    = protein {};

symbol hs = "[DEGHKNPQRS]";
symbol hy = "[DEGHKNPQRSY]";
symbol ha = "[ADEGHKNPQRSY]";

pattern defTurn = {3hs,hy} | {4hs,.} | {5hs,2.};
pattern posTurn = {3hs,.} | {4hs,2ha,.} | {3hs,4hy};
pattern mayTurn = {4hs,4.} | {5hs,4.} | {4hs,2hy,3.};

reference p = <10,99>;

pattern defTurnGen = (p \ (PTurn(p))) @ defTurn;
pattern posTurnGen = (p \ (MTurn(p))) @ posTurn;
pattern mayTurnGen = <10,18> @ mayTurn;

function int PTurn (char *s) = {
    return(!contains(defTurn,s,STRING,ANY) &&
           matches(posTurnGen,s,STRING));
};

function int MTurn (char *s) = {
    return(!contains(posTurn,s,STRING,ANY) &&
           matches(mayTurnGen,s,STRING));
};

evaluate contains(defTurnGen,"ProteinDatabase",PIR,ALL);
```

Figure 57: PAMALA version of the *TurnGen* procedure

The program in Figure 57 defines three density patterns, representing decreasingly hydrophilic turn segments — *defTurn*, *posTurn*, and *mayTurn*. These turn segments are then used to break the amino acid sequence into small pieces (which presumably represent the desired α/β proteins) in a hierarchical fashion. First, the entire database is broken into segments that are separated by occurrences of the pattern *defTurn*. The resulting segments are passed to the function *PTurn*, which breaks them into still smaller pieces, based on matches to the pattern *posTurn*. These pieces are then passed to the function *MTurn*, which breaks any pieces that are still longer than eighteen proteins, based on matches to the pattern *mayTurn*.

tRNA Secondary Structure

Several authors have presented systems for locating matches to tRNA genes, among them Staden [1980], Saurin and Marliere [1987], and Gautheret *et al.* [1990]. tRNA genes are characterized by a common two-dimensional structure in the shape of a cloverleaf, which is composed of three stem-loop structures, with an additional stem at its base. Figure 58 shows an example of a tRNA gene.

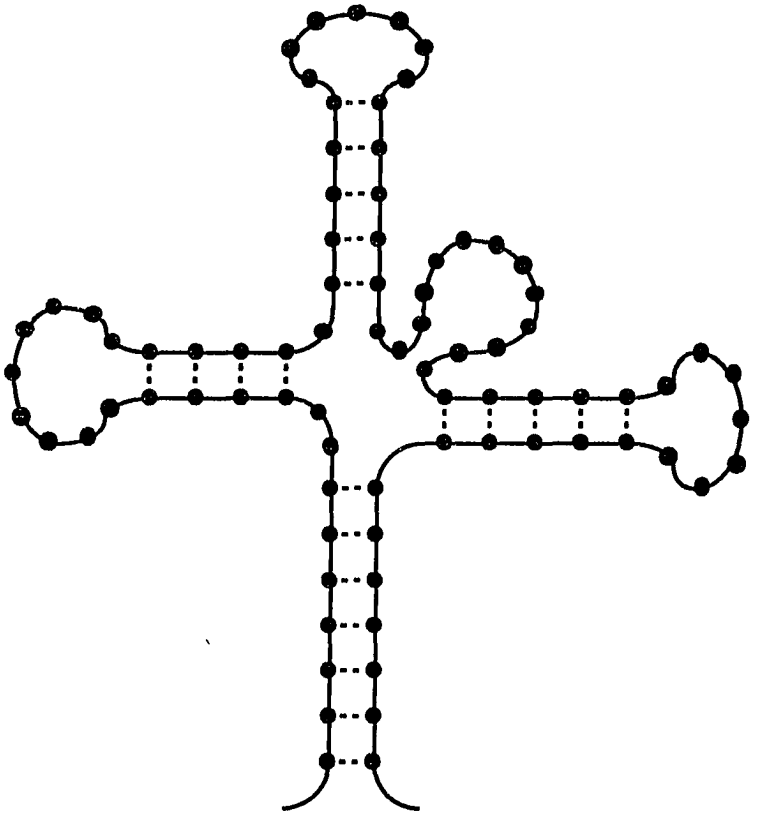

Figure 58: tRNA secondary structure

The one-page PAMALA program in Figure 59 simulates the entire pattern matching system by Staden [1980]. The program defines a *tRNA* pattern by combining four spacers (*a*, *d*, *ac*, and *t*) and their respective complements (which represent the four stems of the

structure) with (unnamed) spacers of appropriate lengths that form the loops at the ends of three of the stems, as well as the connections between the stems. In his program, Staden allows some variation on the basic pattern by scoring each of the stems based on the nucleotides that make up the stem, and by rejecting matches that do not achieve a high enough score. Rather than using PAMALA's approximate matching capabilities, the program in Figure 59 simply simulates the behavior of Staden's program by appending the appropriate condition to the definition of the *tRNA* pattern.

```
alpha DNA  = [ACGT];
score none = DNA {};

int AStem;
int DStem;
int ACStem;
int TStem;
reference a  = <7>;
reference d  = <3,4>;
reference ac = <5>;
reference t  = <5>;
reference aC  = ~a;
reference dC  = ~d;
reference acC = ~ac;
reference tC  = ~t;

function void getParams (void) = {
    printf("Input aminoacyl score:  "); scanf("%d",&AStem );
    printf("Input dhu stem score:   "); scanf("%d",&DStem );
    printf("Input anti codon score: "); scanf("%d",&ACStem);
    printf("Input tu stem score:    "); scanf("%d",&TStem );
};

function int Value (register char c) = {
    switch (c) {
        case 'A': return(1);
        case 'C': return(2);
        case 'G': return(3);
        case 'T': return(4);
    }
};

function int Score (register char *s, register char *sc) = {
    register int v, i;
    for (v = i = 0; s[i] != '\0'; i++)
        if (Value(s[i])+Value(sc[i]) == 5)
            v += 2;
        else if (Value(s[i])+Value(sc[i]) == 7)
            v += 1;
    return(v);
};

pattern tRNA = a <2> d <8,13> dC <1> ac <7> acC <5,21> t <7> tC aC \
        (Score(a,aC) >= AStem  &&  Score(d,dC) >= DStem  &&
         Score(ac,acC) >= ACStem  &&  Score(t,tC) >= TStem);

evaluate getParams();
evaluate contains(tRNA,"Database",PLAIN,ALL);
```

Figure 59: PAMALA program for finding tRNA secondary structures

Methyltransferase

The last example illustrates the approximate pattern matching capabilities of the new system. Posfai *et al.* [1989] describe the overall structure of a biological pattern known as cytosine methyltransferase ("MTase"), and list the actual protein sequences for thirteen bacterial MTases. A complete MTase pattern consists of ten motifs, each of which represents the consensus of several protein sequences, separated by spacers of different lengths, as shown in Figure 60. The PAMALA program in Figure 60 also demonstrates how the **include** statement can be used to structure a PAMALA program. The included files separately define the PAM250 scoring scheme, motifs m_1 through m_{10}, and thresholds t_1 through t_{10}, together with the function *getThresholds*, respectively.

```
include PAM250;

include Motifs;

include Thresholds;

pattern MTase = m1%t1 <1,29> m2%t2 <5,10> m3%t3 <6,24> m4%t4 <-3,31>
                m5%t5 <9> m6%t6 <11,32> m7%t7 <3,11> m8%t8 <84,272>
                m9%t9 <5,15> m10%t10;

evaluate getThresholds();
evaluate contains(MTase,"ProteinDatabase",PIR,ALL);
```

Figure 60: PAMALA program for finding MTase patterns

The contents of the file "PAM250" are shown in Figure 61. The *PAM250* scoring scheme is based on the well-known Dayhoff metric [Dayhoff, 1978], which defines the cost of substituting one protein for another, based on the likelihood of this substitution occuring in the evolutionary process. Since the values of the *PAM250* scheme do not have an easily characterizable pattern, they must be listed individually, using the enumerative format of the initialization statement for scoring schemes.

```
alpha protein = [ARNDCQEGHILKMFPSTWYV];

score PAM250  = protein { <A.A>  #  0.90;
                          <A.R>  #  1.09;
                          <A.N>  #  0.99;
                          <A.D>  #  0.98;
                          <A.C>  #  1.12;
                          ..............
                        };
```

Figure 61: PAMALA definition of the PAM250 scoring scheme

Posfai *et al.* derive the motifs that constitute the MTase pattern as the consensus of thirteen known bacterial MTases. At any given position of a motif, the pattern can match one or more proteins, depending on whether or not the consensus at that particular position was unanimous. The PAMALA definition given in Figure 62 for the first of the ten motifs was created by mechanically translating invariant positions of the consensus motif into the corresponding character, positions which match two or three different proteins into the appropriate character classes, and all other pattern positions into the wildcard symbol. Leading and trailing wildcards were omitted.

```
pattern m1 = "[ILM]" "[SD]" "[LF]" "F" "[SAC]" "G" .
             "[GM]" "[GA]" "[FLI]" .. "[GAS]" .... "G";
```

Figure 62: PAMALA definition of a single MTase motif

The definition of the pattern given in Figure 62 is equivalent to the motif used by Posfai *et al.* to search for the MTase pattern. It is possible, however, to refine the definition by using positional weights and consensus characters. These improvements are developed below.

A particular position of a motif is said to be totally conserved if it matches the same protein in all of the known sequences; the position is called strongly conserved if it matches only a few (chemically similar) proteins, and weakly conserved, if it matches

several (possibly chemically different) proteins. The relative importance of the various positions in a motif, which depends on the conservedness of the proteins at those positions, can be easily expressed with positional weights. In this particular example, invariant positions are given a weight of three, positions matching two different proteins are assigned a weight of two, and positions with three different proteins use the (default) weight of one. The wildcard symbol, which is used for all other positions, does not contribute to the score, and therefore does not need to be weighted. Adjacent positions with the same weight are collected into groups. Figure 63 shows the motif from Figure 62, with the appropriate weights attached.

```
pattern m1 = "[ILM]" ("[SD]" "[LF]"):2 "F":3 "[SAC]" "G":3 .
             ("[GM]" "[GA]"):2 "[FLI]" .. "[GAS]" .... "G":3;
```

Figure 63: PAMALA definition of a single MTase motif, with positional weights

Furthermore, it is possible to account for the different frequencies of the various proteins at non-conserved positions. This is done by replacing the character classes with consensus characters, and assigning weights to individual proteins according to the frequency with which this protein occurs at a particular position. The resulting consensus characters are normalized with positional weights, such that the most frequently occuring protein at each position yields an absolute score of one. Figure 64 shows the resulting pattern for motif m_1; the remaining nine motifs can be derived in a similar fashion.

```
pattern m1 = "[> I:7 LM:3]":.143
             ("[> S:7 D:6]":.143 "[> L:12 F]":.083):2
             "F":3
             "[> S:7 A:5 C]":.143
             "G":3
             .
             ("[> G:12 M]":.083 "[> G:10 A:3]":.100):2
             "[> F:6 L:4 I]":.167
             ..
             "[> G:7 A:5 S]":.143
             ....
             "G":3;
```

Figure 64: PAMALA definition of a single MTase motif, with consensus characters

Finally, the file "Thresholds" contains the declarations for the numeric variables t_1 through t_{10}, as well as the definition of the function *getThresholds*, which interactively obtains values for those variables from the user. When determining reasonable threshold values, it is important to take into account any positional weights attached to (portions of) the pattern, as well as the fact that PAMALA normalizes scores with respect to the length of the matched pattern. Hence, the maximum possible score for a particular motif is computed as the sum of the maximum score at each position, divided by the number of positions that do not contain the wildcard symbol. For motif m_1, the value of this score is (1+2+2+3+1+3+0+2+2+1+0+0+1+0+0+0+0+3)/11 = 1.909. A threshold t_1 of 1.8 would therefore result in a "tight" search, and a threshold of 1.5 or 1.6 in a more "loose" search.

CHAPTER 10: RESULTS

The PAMALA pattern matching system described in this dissertation has been successfully implemented on a number of hardware platforms and under different operating systems. The entire system consists of about 10,000 lines of C source code, with the first phase (the source-to-source translator) accounting for about 25% of the total, and the remaining 75% being distributed among the implementation of the pattern evaluation algorithm, the backtracking strategy, and the various pattern matching algorithms (i.e., the second phase). The system was written in ANSI C, in order to make the code as portable as possible across different computer systems. Ironically, this very fact is currently the major obstacle to porting the system, as ANSI-compliant C compilers are not (yet) as widely available as compilers for older versions of C. However, this situation should be rectified as time goes by.

Nonetheless, the system has been implemented on several architectures, as diverse as a Sun 3 workstation[20], a Sparc-2[21], and a Sequent Symmetry multiprocessor[22], under the Sun OS 4.1.1 and DYNIX 3.1.2 operating systems, respectively. On these systems, the requirement for an ANSI C compiler was met with the GNU C compiler, developed by the GNU Project of the Free Software Foundation. Porting between the systems merely required transferring and recompiling the files containing the source code — no modifications to the code were necessary.

Upon installation of the PAMALA software, a utility program determines the cost constants c_u and c_a for the various algorithms, using a Monte-Carlo simulation. The

---

20 Sun 3/80, with a Motorola 68030 processor, and 8 MB of RAM.
21 Sun 4/75, with a Sun Sparc processor, and 64 MB of RAM.
22 Sequent S81, with 8 Intel 386/387 processors, and 56 MB of RAM.

program searches a random database for a collection of patterns of various sizes and different complexity. Table 1 shows the values that were obtained on the Sparc-2 for the six algorithms currently implemented in PAMALA. With these values, the cost constants $c_u(p,D)$ and $c_a(p)$ in Figures 31 and 32 represent the time, in system clock ticks, for performing an unanchored and anchored search, respectively.

	c_u	$c_{u_{max}} - c_{u_{min}}$	c_a	$c_{a_{max}} - c_{a_{min}}$
c^{SP}	5.564	0.198	5.707	0.097
c^{BM}	10.172	1.272	24.032	3.719
c^{AC}	3.560	2.171	3715.875	41.952
c^{RE}	4.577	0.255	1443.932	11.878
c^{AN}	172.926	15.004	3767.753	23.561
c^{AR}	5.522	0.782	103.901	5.559

Table 1: Cost constants

The concept of cost constants is based on a model of the asymptotic expected performance of the various algorithms, which is reflected in the formulae in Figures 31 and 32. To evaluate the quality of this model, the variance of the data obtained from the Monte-Carlo simulation was examined. While for none of the implemented algorithms the values were absolutely constant, the difference between the maximum and the minimum value for any particular algorithm was generally insignificant, compared to the difference in values between algorithms. On the other hand, the analysis also pointed out weaknesses in the model for two of the algorithms, namely the Aho-Corasick algorithm

for matching sets of keywords, and the algorithm for matching approximate networks. The performance of the Aho-Corasick algorithm in the unanchored case is not independent of the pattern, as suggested by the asymptotic analysis, but instead depends on the "width" of the trie generated from the pattern — with a trie that is very "narrow" near the root, the algorithm is almost twice as efficient as with a very "wide" trie. The model for the unanchored case of the approximate network algorithm turned out to be ill-suited as well. Since the cost of using this algorithm is a function of the threshold, the cost constant depends heavily on the particular pattern, especially if positional weights or unusual scoring schemes are used. The constant as determined above is therefore appropriate only for "normalized" patterns.

Since most of the constants in Table 1 above depend on different aspects of the pattern p (e.g., the length of a spacer, or the threshold of an approximate pattern), they can not be used to directly compare the relative performance of different algorithms. Instead, this was done by conducting pairwise comparisons — i.e., using two different algorithms to search for the same pattern. The results of this evaluation support not only the general strategy of using different algorithms for different classes of patterns, but also most of the particular hierarchy of algorithms used by PAMALA. The only exception was the Aho-Corasick algorithm, whose performance turned out to be merely at par with that of the regular-expression algorithm in the unanchored case, and decidedly worse in the anchored case. This result suggests re-examining the current implementation for any inefficiencies, or replacing the algorithm altogether, if none are found.

The remainder of this chapter examines PAMALA in terms of the time spent executing the different phases of the system, and analyzes the pattern matching strategy generated by the pattern evaluation algorithm.

The first phase parses the PAMALA program and translates it into C code, which is then compiled and linked with the code for the second phase. Table 2 shows, for each of the four patterns defined in the previous chapter, the execution times (in seconds) for the three stages of the first phase. All times are CPU times in seconds, and were measured on the Sparc-2.

	Stem-Loop	α/β Protein	tRNA	MTase
Translation	0.0	0.0	0.0	0.3
Compilation	0.5	0.6	0.7	1.7
Linking	0.4	0.3	0.3	0.3
Total	0.9	0.9	1.0	2.3

Table 2: Execution times (Phase 1)

With the exception of the MTase pattern, the time for translating the PAMALA code and compiling and linking the resulting C code is truly insignificant — not only in absolute terms, but in particular relative to the amount of time spent evaluating and searching for the patterns (as shown in Table 3, below). In the case of the MTase pattern, however, the execution time for the first phase amounts to more than 10% of the combined execution time for both phases. This percentage not only reflects the length of this particular PAMALA program (some 380 lines — the second-longest program, the specification of the tRNA pattern, is less than 50 lines long), but also is a result of the fact that the execution time of the MTase pattern is by far the shortest among the four patterns. In all four cases, the majority of the time spent in this phase is used for the C compiler and the linker. Consequently, execution time for the first phase can be

reduced significantly only by generating less C code (rather than by producing it in less time), or by using a faster C compiler.

The second phase evaluates the pattern in order to determine the optimum decomposition and search strategy, and then searches the database. Table 3 shows the time spent evaluating and searching for each of the four patterns. The search was performed over a random DNA database of over 1,000,000 nucleotides for the stem-loop and tRNA patterns, and a random protein database of more than 1,000 amino acids for the α/β protein and MTase patterns. The different lengths of the two databases were chosen to reflect the different lengths of DNA and protein sequences, respectively. All times are CPU times in seconds, and were measured on the Sparc-2.

	Stem-Loop	α/β Protein	tRNA	MTase
Evaluation	0.0	3.8	0.0	14.5
Search	223.0	473.3	92.1	1.6
Total	223.0	477.1	92.1	16.1

Table 3: Execution times (Phase 2)

With the exception of the MTase pattern, the majority of the time spent during the second phase is used for searching the database; the pattern evaluation algorithm uses less than one percent of the total. In the case of the MTase pattern, however, pattern evaluation accounts for approximately 90% of the total time. This percentage is explained by the fact that the MTase pattern consists of ten different motifs, each of which is defined as an approximate pattern. Hence, a separate Monte-Carlo simulation must be

conducted for each motif in order to obtain initial estimates of the match frequencies, which in turn are necessary for determining the order of evaluation. Since each of these simulations is performed over a string of over 10,000 symbols, the pattern evaluation algorithm "searches" a total of more than 100,000 symbols, whereas the length of the actual database is only 1,000.

In the analysis of the pattern evaluation algorithm, the outcome of the algorithm (the pattern decomposition and order of evaluation) is of even greater interest than the time taken by the algorithm, since it directly affects the time spent searching the database. The results of evaluating the four sample patterns are described in the following paragraphs. With one exception (the α/β protein), the algorithm performed as expected.

The pattern matching strategy generated for the stem-loop structure separately searches for each of the three spacers that define the overall patttern. First, the algorithm performs an unanchored search for the rightmost spacer. Every match defines the right side of the stem of the structure. The algorithm then attempts to find the left side (i.e., the inverse repeat of the matched string) with a right-anchored anchored search for the appropriate keyword, and, if successful, verifies the length of the enclosed spacer with a both-anchored "search" for the loop of the structure.

The α/β protein consists of two different classes of patterns, namely "turns" (i.e., the density patterns *defTurn*, *posTurn*, and *mayTurn*), and the corresponding "generators" (i.e., list patterns *defTurnGen*, *posTurnGen*, and *mayTurnGen*). The regular expression algorithm is used to search for "turn" patterns, in recognition of the fact that the Aho-Corasick algorithm is not as efficient. For "generator" patterns, the evaluation algorithm decided on matching the conditional pattern first (searching for a spacer and evaluating the boolean expression), then searching for a density pattern, another conditional pattern,

et cetera. This expensive search strategy is the result of the initial cost estimate of zero for the boolean expression (functions *PTurn* and *MTurn*, respectively), and can only be remedied by calculating a more accurate estimate of the cost associated with the functions *PTurn* and *MTurn*.

The search strategy for the tRNA pattern is illustrated in Figure 65. Each of the fourteen spacers that define the tRNA pattern must be matched separately, and the search proceeds in the order indicated by the circled numbers in Figure 65. The most important question is which spacers are matched first, since they account for the majority of the search time; the remaining parts of the pattern are searched for only infrequently. The evaluation algorithm performs well in that regard, by choosing the stems that form the "leg" (1) and "right arm" (2,3) of the structure, since these spacers are the longest ones, and are of fixed length. The algorithm then continues the match by searching for the bulge between the "head" and the "right arm" of the structure (4) and the loop at the end of the "left arm" (5). Since these spacers are the ones with the largest variance, this may seem counterintuitive at first; however, it is necessary in order to reduce the size of the interval that must be searched before the stem that forms the "leg" of the structure (6) can be completed. The algorithm continues by searching for the remaining unmatched parts of the structure in clockwise order.

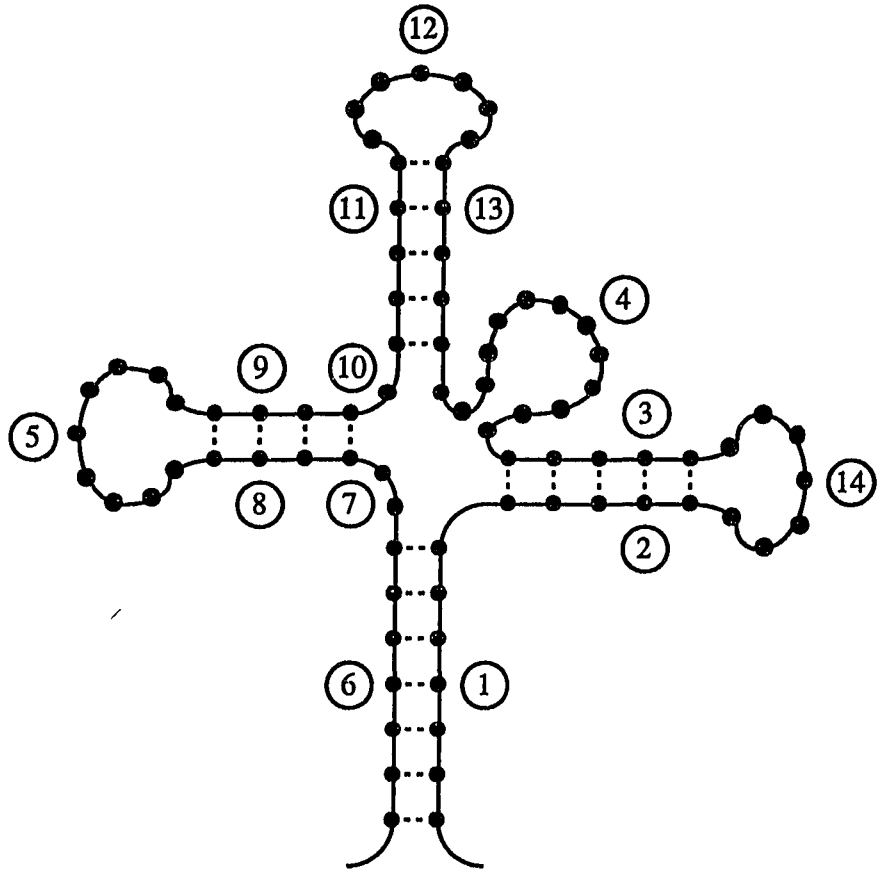

Figure 65: Pattern matching strategy for tRNA secondary structure

The search strategy generated by the pattern evaluation algorithm for the MTase pattern first searches for the motif with the lowest match frequency, and then "grows" the match outward in a linear fashion. This is an especially satisfying result, in that the generated strategy is identical to the one that is hardcoded into ANREP [Mehldau and Myers, 1991].

The values in Table 3 for time spent searching the database are not surprising, given the information on the pattern matching strategy used. Even though the tRNA pattern is more than four times as large as the single stem-loop structure, the system only needs half

as much time to search for the former than it does for the latter, since the initial stem of the single stem-loop generates six matches at each position of the database, whereas the initial stem of the tRNA only generates one. The large amount of time used by the α/β protein is caused by the incorrect pattern matching strategy, and should decrease dramatically when the correct strategy is used.

Lastly, the performance of the system was evaluated on different hardware architectures and software environments. Table 4 shows the time spent for the second phase (pattern evaluation and search) on the three systems mentioned above. All times are CPU times in seconds.

	Stem-Loop	α/β Protein	tRNA	MTase
Sparc-2	223.0	477.1	92.1	16.1
Symmetry	1771.6	2851.1	579.8	267.4
Sun 3	2229.6	3631.2	739.6	1833.3

Table 4: Performance comparison

Although not too surprising, the results are nevertheless useful in that they indicate the kind of performance that can be expected when using the system with different hardware configurations. While the Sequent Symmetry is a multiprocessor machine, the current implementation of PAMALA does not utilize more than one processor at a time, and hence does not take advantage of the available parallelism.[23] The large amount of

[23] A brute-force parallelism (where different processors are assigned to different sections of the database) would be fairly easy to implement; however, a more sophisticated approach (where different processors are assigned to different parts of the pattern) would require a major effort, and possibly involve a redesign of the overall system.

time spent searching for the MTase pattern on the Sun 3 is explained by the lack of a math coprocessor on this machine. In contrast to the algorithms for spacers and exact matches, both algorithms for approximate pattern matching make heavy use of floating-point computations.

In conclusion, it is safe to state that the experimental results strongly support the general strategy upon which the new pattern matching system is based. The single most promising improvement concerns the issue of using empirically determined constants for calculating the costs of searching for a particular pattern with different algorithms. The above results with respect to the pattern evaluation algorithm reaffirm the need for frequency and cost estimates that are highly accurate and, at the same time, efficiently calculated. Instead of relying on algorithm-specific constants, it seems preferable to determine the cost values for each pattern individually, either through analysis (in the case of algorithms for exact pattern matching), or through simulation (in the case of algorithms for approximate matching). Such an approach, however, raises the question of how much additional effort is required for accurately determining the necessary values, and whether this extra effort is worth its cost. Clearly, speed and accuracy are conflicting goals in this context, and further research is needed to determine whether a suitable compromise can be found.

CHAPTER 11: DISCUSSION

This dissertation has presented the design and implementation of a pattern matching system designed specifically for, but not limited to, applications in molecular biology. The system, called PAMALA, addresses the problems that render most of the earlier systems inadequate for biological pattern matching tasks — efficiency in the face of very large databases, the ability to describe and search for both exact and approximate matches, and the power to express complex biological patterns.

The dissertation significantly extends previous work in several ways. First, PAMALA provides a method for specifying approximate patterns that incorporates and unifies many of the previously used schemes. Second, PAMALA allows one to specify and search for a much larger class of patterns than any of the earlier systems. Third, PAMALA efficiently searches for these patterns with a novel, optimized backtracking strategy that utilizes a library of specialized pattern matching algorithms.

The new pattern matching system represents a pragmatic blend of theory and practice. It combines a collection of highly efficient algorithms with an overall design and implementation that takes into account the constraints commonly encountered in the real world. As with every large software system, however, PAMALA is not perfect. Possible improvements to the system can be classified into one of two categories — enhancements or refinements of the overall pattern matching strategy (which require considerable intellectual effort, and are considered to be topics for further research), and improvements to the particular implementation (which require "only" the expenditure of significant software engineering effort, skills, and talent). The remainder of this chapter gives examples of both types of problems, and discusses some possible solutions.

Several problems are related to the two-phase design of the system, and most of them fall into the second category of problems mentioned above. One such problem has already been discussed in the chapter on system implementation — the fact that it is not possible for the system to error-check the C code contained in a PAMALA program, and, in particular, with respect to the proper use of PAMALA variables within that code. This problem could be solved by incorporating a lexical analyzer and parser for C code into the first phase.

A related problem is the fact that the two-phase design requires several steps in order to execute a single PAMALA program — executing the first phase (the source-to-source translator), compiling and linking the resulting C program, and executing the second phase. While this process can be automated fairly easily through the use of command files or "scripts" in "conventional" computing environments (i.e., under operating systems such as UNIX, VMS, or even MS-DOS), this is considerably more difficult to do on computer systems with radically different system architectures, such as the Macintosh. Since most of PAMALA's end users are researchers in fields other than computer science, a more user-friendly environment and interface would be a definite boost to both the usefulness and popularity of the system. Unfortunately, the solution to this problem entails eliminating the second phase by adding a C interpreter to the first phase — a change that most likely would have an adverse affect on system performance.[24]

An other area open to improvements is the pattern matching library. Adding more algorithms to the library could either improve the performance of the system, or enhance

---

24 To make the system truly user-friendly, the pattern specification language would have to be replaced by a graphical interface that allowed patterns to be defined with a point-and-click paradigm.

its pattern matching capabilities, or both. For example, a simple counting scheme could replace the current expansion of density patterns into the corresponding regular expressions, unless the density expression was part of an approximate pattern. Since the generated regular expressions are of potentially exponential size, such an algorithm would certainly improve the space requirements of the system, and would possibly result in more efficient searches as well. On the other hand, developing polynomial-time algorithms (with small exponents), for matching, e.g., approximate regular expressions with conjunction and difference, would enhance the class of patterns that could be expressed and searched for with the system. However, the latter modification clearly falls under the category of "open problems for further research."

Another area where restrictions could be removed from the current system concerns the number of subpattern decompositions and backtracking orders considered by the pattern evaluation algorithm. This could be done by rearranging the parse tree — for example, by joining adjacent binary nodes of the same type (e.g., concatenation or alternation) into a single n-ary node, and considering all possible orders of evaluation among the children of this node. While such a strategy might, for some patterns, result in a more efficient backtracking order, it might also have an unwanted effect on the number of permutations that must be considered if the pattern contains back references. Again, further research is needed to evaluate the tradeoffs.

Finally, as discussed in the previous chapter, the current method for estimating the match frequency and cost of evaluation for approximate patterns is rather unsatisfactory, and a scheme that allowed these numbers to be determined analytically would clearly be preferable. Similarly, any rigorous algorithm for determining an initial estimate of the match frequency and cost of evaluation for the boolean expression in a conditional

pattern would be preferable over the current method. However, it is not known whether efficient solutions to either problem exist, and any significant research in this area is certainly worthy of a separate dissertation.

REFERENCES

[Abarbanel *et al.*, 1984]
Abarbanel, R.M., P.R. Wieneke, E. Mansfield, D.A. Jaffe, and D.L. Brutlag, "Rapid Searches for Complex Patterns in Biological Molecules," *Nucleic Acids Research*, Vol. 12, No. 1 (January 1984), pp. 263-280.

[Aho, 1980]
Aho, A.V., "Pattern Matching in Strings," *Formal Language Theory, Perspectives and Open Problems*, R.V. Book (Ed.), Academic Press, New York, NY, 1980.

[Aho, 1990]
Aho, A.V., "Algorithms for Finding Patterns in Strings," *Handbook of Theoretical Computer Science — Vol. A: Algorithms and Complexity*, J. van Leeuwen (Ed.), MIT Press, Cambridge, MA, 1990.

[Aho and Corasick, 1975]
Aho, A.V., and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, Vol. 18, No. 6 (June 1975), pp. 333-340.

[Aho *et al.*, 1974]
Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[Aho *et al.*, 1979]
Aho, A.V., B.W. Kernighan, and P.J. Weinberger, "AWK — A Pattern Matching and Scanning Language," *Software — Practice and Experience*, Vol. 9, No. 4 (April 1979), pp. 267-280.

[Aho *et al.*, 1983]
Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[Aho *et al.*, 1988]
Aho, A.V., B.W. Kernighan, and P.J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, MA, 1988.

[American National Standard, 1978]
American National Standard, *Programming Language FORTRAN* X3.9-1978, American National Standard Institute, New York, NY, 1978.

[Apostolico and Preparata, 1983]
Apostolico, A., and F.P. Preparata, "Optimal Off-Line Detection of Repetitions in a String," *Theoretical Computer Science*, Vol. 22, No. 3 (February 1983), pp. 297-315.

[Baeza-Yates and Gonnet, 1989]
Baeza-Yates, R.A., and G.H. Gonnet, "A New Approach to Text Searching," *Proceedings of the 12th Annual ACM-SIGIR Conference on Information Retrieval*, 1989, pp. 168-175.

[Barker *et al.*, 1990]
Barker, W.C., D.G. George, and L.T. Hunt, "Protein Sequence Database," *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, R.F. Doolittle (Ed.), Methods in Enzymology, Vol. 183, Academic Press, San Diego, CA, 1990.

[Boyer and Moore, 1977]
Boyer, R.S., and J.S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, Vol. 20, No. 10 (October 1977), pp. 262-272.

[Burks *et al.*, 1990]
Burks, C., *et al.*, "GenBank: Current Status and Future Direction," *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, R.F. Doolittle (Ed.), Methods in Enzymology, Vol. 183, Academic Press, San Diego, CA, 1990.

[Chang and Lawler, 1990]
Chang, W.I., and E.L. Lawler, "Approximate String Matching in Sublinear Expected Time," *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, 1990, pp. 116-124.

[Cohen *et al.*, 1983]
Cohen, F.E., R.M. Abarbanel, I.D. Kuntz, and R.J. Fletterick, "Secondary Structure Assignment for α/β Proteins by a Combinatorial Approach," *Biochemistry*, Vol. 22, No. 21 (October 1983), pp. 4894-4904.

[Cohen *et al.*, 1986]
Cohen, F.E., R.M. Abarbanel, I.D. Kuntz, and R.J. Fletterick, "Turn Prediction in Proteins Using a Pattern-Matching Approach," *Biochemistry*, Vol. 25, No. 1 (January 1986), pp. 266-275.

[Commentz-Walter, 1979]
Commentz-Walter, B., "A String Matching Algorithm Fast on the Average," *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, July 1979, pp. 118-132.

[Crochemore, 1981]
Crochemore, M., "An Optimal Algorithm for Computing the Repetitions in a Word," *Information Processing Letters*, Vol. 12, No. 5 (October 1981), pp. 244-250.

[Davies and Bowsher, 1986]
Davies, G., and S. Bowsher, "Algorithms for Pattern Matching," *Software—Practice and Experience*, Vol. 16, No. 6 (June 1986), pp. 575-601.

[Dayhoff, 1978]
Dayhoff, M.O., *Atlas of Protein Sequence and Structure*, Vol. 5, Suppl. 3, National Biomedical Research Foundation, Washington, DC, 1978.

[Devereux *et al.*, 1984]
Devereux, J., P. Haeberli, and O. Smithies, "A Comprehensive Set of Sequence Analysis Programs for the VAX," *Nucleic Acids Research*, Vol. 12, No. 1 (January 1984), pp. 387-395.

[Farber *et al.*, 1964]
Farber, D.J., R.E. Griswold, and I.P. Polonsky, "SNOBOL, a String Manipulation Language," *Journal of the ACM*, Vol. 11, No. 2 (January 1964), pp. 21-30.

[Galil, 1976]
Galil, Z., "Two Fast Simulations Which Imply Some Fast String Matching and Palindrome Recognition Algorithms," *Information Processing Letters*, Vol. 4, No. 4 (January 1976), pp. 85-87.

[Galil and Giancarlo, 1988]
Galil, Z., and R. Giancarlo, "Data Structures and Algorithms for Approximate String Matching," *Journal of Complexity*, Vol. 4, No. 1 (March 1988), pp. 33-72.

[Galil and Park, 1990]
Galil, Z., and K. Park, "An Improved Algorithm For Approximate String Matching," *SIAM Journal on Computing*, Vol. 19, No.6 (December 1990), pp. 989-999.

[Galil and Seiferas, 1978]
Galil, Z., and J.I. Seiferas, "A Linear-Time On-Line Recognition Algorithm for 'Palstar'," *Journal of the ACM*, Vol. 25, No. 1 (January 1978), pp. 102-111.

[Gautheret *et al.*, 1990]
Gautheret, D., F. Major, and R. Cedergren, "Pattern Searching/Alignment with RNA Primary and Secondary Structures: An Effective Descriptor for tRNA," *CABIOS*, Vol. 6, No. 4 (October 1990), pp. 325-331.

[Golomb and Baumert, 1965]
Golomb, S.W., and L.D. Baumert, "Backtrack Programming," *Journal of the ACM*, Vol. 12, No. 4 (October 1965), pp. 516-524.

[Gribskov *et al.*, 1987]
Gribskov, M., A.D. McLachlan, and D. Eisenberg, "Profile Analysis: Detection of Distantly Related Proteins," *Proceedings of the National Academy of Sciences U.S.A.*, Vol. 84, No. 13 (July 1987), pp. 4355-4358.

[Griswold and Griswold, 1990]
Griswold, R.E., and M.T. Griswold, *The Icon Programming Language*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Griswold *et al.*, 1971]
Griswold, R.E., J.F. Poage, and I.P. Polonsky, *The SNOBOL4 Programming Language*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1971.

[Griswold *et al.*, 1981]
Griswold, R.E., D.R. Hanson, and J.T. Korb, "Generators in Icon," *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 2 (April 1981), pp. 144-161.

[Hall and Dowling, 1980]
Hall, P.A., and G.R. Dowling, "Approximate String Matching," *Computing Surveys*, Vol. 12, No. 4 (December 1980), pp. 381-402.

[Hirschberg, 1975]
Hirschberg, D.S., "A Linear Space Algorithm for Computing Maximal Common Subsequences," *Communications of the ACM*, Vol. 18, No. 6 (June 1975), pp. 341-343.

[Hirschberg, 1977]
Hirschberg, D.S., "Algorithms for the Longest Common Subsequence Problem," *Journal of the ACM*, Vol. 24, No. 4 (October 1977), pp. 664-675.

[Hopcroft and Ullman, 1979]
Hopcroft, J.E., and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[Horowitz, 1984]
Horowitz, E., *Fundamentals of Programming Languages*, 2nd Ed., Computer Science Press, Rockville, MD, 1984.

[Horspool, 1980]
Horspool, R.N., "Practical Fast Searching in Strings," *Software — Practice and Experience*, Vol. 10, No. 6 (June 1980), pp. 501-506.

[Hume, 1988]
Hume, A.G., "A Tale of Two Greps," *Software — Practice and Experience*, Vol. 18, No. 11 (November 1988), pp. 1063-1072.

[Hunt and Szymanski, 1977]
Hunt, J.W., and T.G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences," *Communications of the ACM*, Vol. 20, No. 5 (May 1977), pp. 350-353.

[Hunter, 1991]
Hunter, L., "Artificial Intelligence and Molecular Biology," *AI Magazine*, Vol. 11, No. 5 (January 1991), pp. 27-36.

[Kahn and Cameron, 1990]
Kahn, P., and G. Cameron, "EMBL Data Library," *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, R.F. Doolittle (Ed.), Methods in Enzymology, Vol. 183, Academic Press, San Diego, CA, 1990.

[Karp and Rabin, 1981]
Karp, R.M., and M.O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," Harvard University, Center for Research in Computing Technology, Technical Report 31-81 (December 1981).

[Kernighan and Plauger, 1976]
Kernighan, B.W., and P.J. Plauger, *Software Tools*, Addison-Wesley, Reading, MA, 1976.

[Kernighan and Ritchie, 1988]
Kernighan, B.W., and D.M. Ritchie, *The C Programming Language*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Kernighan *et al.*, 1972]
Kernighan, B.W., D.M. Ritchie, and K.L. Thompson, "QED Text Editor," AT&T Bell Laboratories, Computing Science, Technical Report 5 (1972).

[Klahr and Waterman, 1986]
Klahr, P., and D.A. Waterman, *Expert Systems: Techniques, Tools, and Applications*, Addison-Wesley, Reading, MA, 1986.

[Knuth, 1975]
Knuth, D.E., "Estimating the Efficiency of Backtrack Programs," *Mathematics of Computation*, Vol. 29, No. 129 (January 1975), pp. 121-136.

[Knuth *et al.*, 1977]
Knuth, D.E., J.H. Morris, and V.R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, Vol. 6, No. 2 (June 1977), pp. 323-350.

[Landau and Vishkin, 1988]
Landau, G.M., and U. Vishkin, "Fast String Matching with k Differences," *Journal of Computer and System Sciences*, Vol. 37, No. 1 (August 1988), pp. 63-78.

[Lapedes *et al.*, 1989]
Lapedes, A.S., C. Barnes, C. Burks, R. Farber, and K.M. Sirotkin, "Application of Neural Networks and Other Machine Learning Algorithms to DNA Sequence Analysis," *Computers and DNA*, G. Bell and T. Marr (Eds.), SFI Studies in the Sciences of Complexity VII, Addison-Wesley, Reading, MA, 1989.

[Lesk, 1975]
Lesk, M.E., "Lex — A Lexical Analyzer Generator," AT&T Bell Laboratories, Computing Science, Technical Report 39 (October 1975).

[Levenshtein, 1966]
Levenshtein, V.I., "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," *Soviet Physics Doklady*, Vol. 10, No. 8 (February 1966), pp. 707-710.

[Main and Lorentz, 1984]
Main, M.G., and R.J. Lorentz, "A $O(n\log n)$ Algorithm for Finding All Repetitions in a String," *Journal of Algorithms*, Vol. 5, No. 3 (September 1984), pp. 422-432.

[Main and Lorentz, 1985]
Main, M.G., and R.J. Lorentz, "Linear Time Recognition of Square-Free Strings," *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil (Eds.), Springer Verlag, Heidelberg, Germany, 1985.

[Manacher, 1975]
Manacher, G. "A New Linear-Time On-Line Algorithm for Finding the Smallest Initial Palindrome of a String," *Journal of the ACM*, Vol. 22, No. 3 (July 1975), pp. 346-351.

[McCreight, 1976]
McCreight, E.M., "A Space-Economical Suffix Tree Construction Algorithm," *Journal of the ACM*, Vol. 23, No. 2 (April 1976), pp. 262-272.

[McMahon, 1979]
McMahon, L.E., "SED — A Non-Interactive Text Editor," *UNIX Programmer's Manual*, 7th Ed., AT&T Bell Laboratories, Murray Hill, NJ, 1979.

[McNaughton and Yamada, 1960]
McNaughton, R., and H. Yamada, "Regular Expressions and State Graphs for Automata," *IRE Transactions on Electronic Computers*, Vol. EC-9, No. 1 (March 1960), pp. 39-47.

[Mehldau and Myers, 1991]
Mehldau, G., and E.W. Myers, "A System for Pattern Matching Applications on Biosequences," submitted to *CABIOS*.

[Minsky and Papert, 1969]
Minsky, M.L., and S. Papert, *Perceptrons*, MIT Press, Cambridge, MA, 1969.

[Myers, 1986]
Myers, E.W., "An $O(ND)$ Difference Algorithm and Its Variations," *Algorithmica*, Vol. 1, No. 2 (1986), pp. 251-266.

[Myers and Mehldau, 1991]
Myers, E.W., and G. Mehldau, "Algorithms for Approximate Network Regular Expression Pattern Matching," in preparation.

[Myers and Miller, 1988]
Myers, E.W., and W. Miller, "Optimal Alignments in Linear Space," *CABIOS*, Vol. 4, No. 1 (March 1988), pp. 11-17

[Myers and Miller, 1989]
Myers, E.W., and W. Miller, "Approximate Matching of Regular Expressions," *Bulletin of Mathematical Biology*, Vol. 51, No. 1 (1989), pp. 5-37.

[Myers and Mount, 1986]
Myers, E.W., and D.W. Mount, "Computer Program for the IBM Personal Computer which Searches for Approximate Matches to Short Oligonucleotide Sequences in Long Target DNA Sequences," *Nucleic Acids Research*, Vol. 14, No. 1, (January 1986), pp. 501-508.

[National Research Council, 1988]
National Research Council U.S.A., Committee on Mapping and Sequencing the Human Genome, *Mapping and Sequencing the Human Genome*, National Academy Press, Washington, DC, 1988.

[Needleman and Wunsch, 1970]
Needleman, S.B., and C.D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *Journal of Molecular Biology*, Vol. 48, No. 3 (March 1970), pp. 443-453.

[Posfai *et al.*, 1989]
Posfai, J., A.S. Bhagwat, G. Posfai, and R.J. Roberts, "Predictive Motifs Derived from Cytosine Methyltransferases," *Nucleic Acids Research*, Vol. 17, No. 7 (April 1989), pp. 2421-2435.

[Quinlan, 1986]
Quinlan, J.R., "Induction of Decision Trees," *Machine Learning*, Vol. 1, No. 1 (1986), pp. 81-106.

[Rabin, 1985]
Rabin, M.O., "Discovering Repetitions in Strings," *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil (Eds.), Springer Verlag, Heidelberg, Germany, 1985.

[Rosenblatt, 1962]
Rosenblatt, F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, Washington, DC, 1962.

[Sankoff, 1972]
Sankoff, D., "Matching Sequences under Deletion/Insertion Constraints," *Proceedings of the National Academy of Sciences U.S.A.*, Vol. 69, No. 1 (January 1972), pp. 4-6.

[Sankoff and Kruskal, 1983]
Sankoff, D., and J.B. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.

[Saurin and Marliere, 1987]
Saurin, W., and P. Marliere, "Matching Relational Patterns in Nucleic Acid Sequences," *CABIOS*, Vol. 3, No. 2 (June 1987), pp. 115-120.

[Seiferas and Galil, 1977]
Seiferas, J.I., and Z. Galil, "Real-Time Recognition of Substring Repetition and Reversal," *Mathematical Systems Theory*, Vol. 11, No. 2 (1977), pp. 111-146.

[Sellers, 1974]
Sellers, P.H., "On the Theory and Computation of Evolutionary Distances," *SIAM Journal on Applied Mathematics*, Vol. 26, No. 4 (June 1974), pp. 787-793.

[Sellers, 1980]
Sellers, P.H., "The Theory and Computation of Evolutionary Distances: Pattern Recognition," *Journal of Algorithms*, Vol. 1, No. 1 (March 1980), pp. 359-373.

[Sellers, 1984]
Sellers, P.H., "Pattern Recognition in Genetic Sequences by Mismatch Density," *Bulletin of Mathematical Biology*, Vol. 46, No. 4 (1984), pp. 501-514.

[Smit, 1982]
Smit, G. de V., "A Comparison of Three String Matching Algorithms," *Software — Practice and Experience*, Vol. 12, No. 1 (January 1982), pp. 57-66.

[Staden, 1980]
Staden, R., "A Computer Program to Search for tRNA Genes," *Nucleic Acids Research*, Vol. 8, No. 4, (February 1980), pp. 817-825.

[Staden, 1984]
Staden, R., "Computer Methods to Locate Signals in Nucleic Acid Sequences," *Nucleic Acids Research*, Vol. 12, No. 1 (January 1984), pp. 505-520.

[Staden, 1988]
Staden, R., "Methods to Define and Locate Patterns of Motifs in Sequences," *CABIOS*, Vol. 4, No. 1 (March 1988), pp. 53-60.

[Staden, 1989]
Staden, R., "Methods for Calculating the Probabilities of Finding Patterns in Sequences," *CABIOS*, Vol. 5, No. 2 (April 1989), pp. 89-96.

[Stormo, 1990]
Stormo, G.D., "Consensus Patterns in DNA," *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, R.F. Doolittle (Ed.), Methods in Enzymology, Vol. 183, Academic Press, San Diego, CA, 1990.

[Stormo *et al.*, 1982]
Stormo, G.D., T.D. Schneider, L. Gold, and A. Ehrenfeucht, "Use of the 'Perceptron' Algorithm to Distinguish Translational Initiation Sites in E. coli," *Nucleic Acids Research*, Vol. 10, No. 9 (May 1092), pp. 2997-3011.

[Tarhio and Ukkonen, 1990]
Tarhio, J., and E. Ukkonen, "Approximate Boyer-Moore String Matching," University of Helsinki, Department of Computer Science, Technical Report A-1990-3 (March 1990).

[Thompson, 1968]
Thompson, K., "Regular Expression Search Algorithm," *Communications of the ACM*, Vol. 11, No. 6 (June 1968), pp. 419-422.

[Ukkonen, 1985a]
Ukkonen, E., "Algorithms for Approximate String Matching," *Information and Control*, Vol. 64, No. 1/3 (January/March 1985), pp. 100-118.

[Ukkonen, 1985b]
Ukkonen, E., "Finding Approximate Patterns in Strings," *Journal of Algorithms*, Vol. 6, No. 1 (March 1985), pp. 132-137.

[Wagner and Fischer, 1974]
Wagner, R.A., and M.J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, Vol. 21, No. 1 (January 1974), pp. 168-173.

[Wagner and Seiferas, 1978]
Wagner, R.A., and J.I. Seiferas, "Correcting Counter-Automaton-Recognizable Languages," *SIAM Journal on Computing*, Vol. 7, No. 3 (August 1978), pp. 357-375.

[Waterman, 1984]
Waterman, M.S., "General Methods for Sequence Comparison," *Bulletin of Mathematical Biology*, Vol. 46, No. 4 (1984), pp. 473-500.

[Waterman and Smith, 1981]
Waterman, M.S., and T.F. Smith, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, Vol. 147, No. 1 (March 1981), pp. 195-197.

[Weiner, 1973]
Weiner, P., "Linear Pattern Matching Algorithms," *Proceedings of the 14th Symposium on Switching and Automata Theory*, October 1973, pp. 1-11.

[Wirth, 1971]
Wirth, N., "The Programming Language Pascal," *Acta Informatica*, Vol. 1, No. 1 (1971), pp. 35-63.

[Wu and Manber, 1991]
Wu, S. and U. Manber, "Fast Text Searching With Errors," University of Arizona, Department of Computer Science, Technical Report 91-11 (June 1991).