Report Number: WUCS-92-30

1992-01-01

A Model for Detecting Motifs in Biological Sequences

Andrew F. Neuwald and Phillip P. Green

A method for detecting patterns in biological sequences is described that incorporates rigorous statistics for determining significances, and an algebraic system that, in combination with a depth first search procedure, can be used to efficiently search for all patterns up to a specified length. This method includes a context free command language grammar and is formulated using a mathematical model amendable to additions enhancements, The method was implemented and verified by detection of various types of patterns in protein sequences.

A Model for Detecting Motifs in Biological Sequences

Andrew F. Neuwald, Philip P. Green

WUCS-92-30

August 1992

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
Saint Louis , Missouri 63130 (314) 935-6160

Abstract

A method for detecting patterns in biological sequences is described that incorporates rigorous statistics for determining significance, and an algebraic system that, in combination with a depth first search procedure, can be used to efficiently search for all patterns up to a specified length. This method includes a context free command language grammar and is formulated using a mathematical model amenable to additional enhancements. The method was implemented and verified by detection of various types of patterns in protein sequences.

Supported by National Library of Medicine Training Program grant 5-T-LM07049.

LIST OF FIGURES

1.1. Structure of amino acids 2
1.2. Protein structure 3
1.3. Conserved regions between inositol monophosphatase and homologous proteins 6
2.1. Optimal alignment of cytochromes c 9
3.1. A population of aligned protein segments containing amino-terminal ends 20
3.2. Depth first pattern search using elementary blocks 36
3.3. Representation of segments in a block by an array of bits 38
3.4. A lookup table for determining the cardinality of a non-null byte 38
3.5. An algorithm for calculating both the intersection of two sets and the cardinality of the resultant set using a list 38
3.6. Algorithm used to implement the search_blocks( ) operation 47
3.7. Algorithm used to implement the dfs( ) operation 47
3.8. Pruning a pattern search tree 48
4.1. Discrepancy between theoretical and actual pattern probability cutoffs 55
4.2. Effect of varying the minimum block size on the search_blocks( ) operation 56
4.3. Effect of varying the segment length on the search_blocks( ) operation 57
4.4. Detection of 'HIGH' and 'KMSKS' motifs by ASSET in a 3-residue pattern search 59
4.5. Detection of 'HIGH' and 'KMSKS' motifs by ASSET in a 4-residue pattern search 60
4.6. Detection of 'HIGH' and 'KMSKS' motifs by MOTIF 62
4.7. Execution times for ASSET and MOTIF at various segment lengths 63
4.8. Patterns found by ASSET in 15 adenine methylases 64
4.9. ASSET output for the eval_var(*D*) operation corresponding to the variable generator *D.G* = "?..PPY" 65
4.10. Procedure used to detect patterns in the PIR database 69
4.11. Detection of the ATP/GTP-binding site motif A among sequences in the PIR protein database 70
4.12. Detection of 2-residue patterns in the amino-terminal region of proteins 71
5.1. A group of 23 related patterns detected by ASSET in 29 reverse transcriptases 76
6.1. A 2x2 contingency table 82

TABLE OF CONTENTS

LIST OF FIGURES iv
LIST OF TABLES v
1. INTRODUCTION 1
1.1. Sequence Analysis 1
1.2. Biological Sequences 2
1.2.1 Nucleic Acids 2
1.2.2 Proteins 2
1.3. Motifs as Indicators of Structure and Function 4
1.4. Problem Statement 5
1.5. Overview 7
2. SEQUENCE ANALYSIS METHODS 8
2.1. Methods Based On Alignments 8
2.1.1. Amino Acid Weighting Schemes 9
2.1.2. Global Alignment Methods 10
2.1.3. Local Alignment Methods 10
2.1.4. Multiple Alignment Methods 11
2.1.4.1. Heuristic Methods that Add Sequences One at a Time 11
2.1.4.2. Methods Used to Detect Ordered Patterns 12
2.1.4.3. Methods Using Diagonals 13
2.1.5. Profile Analysis 13
2.2. Methods Not Needing Alignments 14
2.2.1. Lexicographically Ordered Matrix Methods 14
2.2.2. Methods Based on Fixed Patterns 15
2.2.3. A Method Based on Expectation Maximization 17
2.2.4. Pattern Matching Programs 17
3. A MODEL FOR DETECTING MOTIFS IN BIOLOGICAL SEQUENCES 18
3.1. Informal Overview 19
3.2. Biological Sequence Entities 22
3.2.1. Definitions 22
3.2.2. Attributed Grammar and Operations 23
3.2.3. Implementation 24
3.3. Pattern Generators 25
3.3.1. Definitions 25

3.3.2. Operations and Attributed Grammar 27
3.3.3. Implementation 31
3.4. Blocks 32
3.4.1. Definitions 32
3.4.2. An Algebra for Blocks 33
3.4.3. Operations 37
3.4.4. Implementation 37
3.5. Aligned Segment Populations 39
3.5.1. Definition 39
3.5.2. Attributed Grammar and Operations 39
3.5.3. Implementation 42
3.6. Generator-Population Dyads 43
3.6.1. Definitions 43
3.6.2. Operations and Attributed Grammar 43
3.6.3. Implementation 46
4. VERIFICATION OF THE METHOD 51
4.1. Implementation - the ASSET Program 51
4.2. ASSET Performance Profile 51
4.2.1. Statistical Significance 52
4.2.2. Statistical Verification 53
4.2.3. ASSET Execution Performance 55
4.3. Detecting Motifs Among Distantly Related Proteins 58
4.3.1. Detection of Motifs in 19 tRNA Synthetases 58
4.3.1.1. Three-Residue Pattern Search 58
4.3.1.1. Four-Residue Pattern Search 59
4.3.2. Comparisons with the MOTIF Program 61
4.4. Detecting Patterns Having Either of Two Amino Acids at a Single Position 64
4.5. Testing for Correlations 64
4.6. Detecting Motifs Present in Only a Minority of Proteins in a Group 66
4.6.1. Finding Minor Patterns in 29 Reverse Transcriptases 67
4.6.2. Searching Large Databases for Related Proteins 69
4.7. Detecting Motifs which are Internally Repeated 70
4.8. Detecting Patterns near Amino-terminal Regions 71
5. CONCLUSION AND FUTURE WORK 73

5.1. Advantages and Disadvantages of the ASSET Method 73
5.2. Further Enhancements 74
5.2.1. Incorporation of a Relatedness Scoring Matrix 74
5.2.2. A Method for Combining Related Patterns into Groups 76
5.3. Tool Development and Future Applications 77
5.3.1. Tools for Retrieving Groups of Proteins 77
5.3.2. Seed Pattern Blocks 77
5.3.3. Application to Nucleic Acid Sequences 77
6. APPENDIX 78
6.1. Definitions of Terms 78
6.2. Statistical Formulas 79
6.2.1. Cumulative Binomial Distribution 79
6.2.2. Fisher's Exact Test 80
6.2.3. "Low Entropy" Segments 81
6.3. Table Abstract Data Type 82
6.3.1. Definition 82
6.3.2. Operations 82
6.4. Proof for $P_{adj} < Np + (Np)^2$ 83
6.5. Protein Sequence Entities 85
6.6. Sample Session 86
7. BIBLIOGRAPHY 89

LIST OF TABLES

1.1. The 20 amino acids that make up proteins 3
3.1. Concise notation for pattern generators 26
3.2. Elementary blocks of length 5 for amino acids 35
4.1. Calculated and empirically determined pattern probabilities 54
4.2. Contingency table corresponding to the table generator "{YF:ILV}.{D:N}PPY" 65
4.3. Sequence context for the "PPY" motif in 15 adenine methylases 66
4.4. Patterns Detected in 29 Reverse Transcriptases 68
5.1. A comparison of MOTIF and ASSET 73
5.2. A PAM 40 matrix for calculating relatedness scores 75
6.1. Definitions and descriptions of terms used in the text 78

A MODEL FOR DETECTING MOTIFS IN BIOLOGICAL SEQUENCES

1. INTRODUCTION

This report proposes a new method for detecting patterns in groups of biological sequences. This method offers several significant improvements over other sequence analysis methods: [1] it does not require an alignment, [2] it has a rigorous statistical basis, [3] it can efficiently search for all patterns up to a given length without heuristics, and [4] it is formulated using a mathematical model amenable to additional enhancements.

1.1. SEQUENCE ANALYSIS

Biological science is undergoing a revolution; there is a growing emphasis on sequence data, as manifested by efforts to sequence the genomes of human and model organisms. This shift in emphasis reflects a belief among molecular biologists that there is much to be gained from gathering sequence data prior to knowing what those sequences encode. Comparison of DNA sequences, or of inferred protein sequences, with other sequences can often lead to valuable insights into their biological function. This has motivated a search for efficient and sensitive sequence analysis methods for detecting patterns shared by groups of nucleic acids or proteins. Some background information on biological sequences and sequence motifs will provide a better understanding into these methods.

1.2. BIOLOGICAL SEQUENCES

1.2.1 NUCLEIC ACIDS

Deoxyribonucleic acid (DNA) is the genetic material of living organisms and thus encodes the information needed for synthesis of the cellular components. DNA consists of two antiparallel polynucleotide chains of deoxyadenosine(A), deoxycytosine(C), deoxyguanosine(G) and deoxythymidine(T) residues connected by phosphodiester linkages. Ribonucleic acid (RNA) is a polynucleotide which plays an essential role in the expression of the genetic information encoded in DNA. RNA consists of a single chain of adenosine(A), cytosine(C), guanosine(G) and uridine(U) residues.

The characterization of nucleic acid sequences is an extensive and rapidly expanding area of research. However, since the focus of this work is on protein sequence analysis, it will only be pointed out that the method to be described here can easily be applied to nucleic acid sequences by incorporating an "alphabet" for nucleotide residues.

1.2.2 PROTEINS

In order to understand the molecular biology of an organism it is important to understand the structure and function of its molecular constituents. Protein is an especially important component since it plays a key structural and catalytic role in the cell. Proteins are made from an "alphabet" of twenty amino acids (Table 1.1) that differ from each other only in a part of the molecule called the R group or side chain (Figure 1.1).

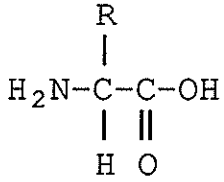


Figure 1.1. Structure of amino acids. Amino acids have a common structure except for their side chain or R group, which is different for each amino acid.

Table 1.1. The 20 amino acids that make up proteins.

Symbol	Chemical Name	Side Chain (R)
A	alanine	small aliphatic
C	cysteine	forms disulfide cross links
D	aspartate	small acidic
E	glutamate	acidic
F	phenylalanine	aromatic
G	glycine	hydrogen atom side chain
H	histidine	imidazole group
I	isoleucine	aliphatic
K	lysine	basic
L	leucine	aliphatic
M	methionine	aliphatic, contains sulfur
N	asparagine	amide
P	proline	substituted α-amino group
Q	glutamine	amide
R	arginine	basic
S	serine	hydroxyl
T	threonine	hydroxyl
V	valine	aliphatic
W	tryptophane	aromatic
Y	tyrosine	aromatic, hydroxyl

The primary structure of proteins consists of a sequence of amino acid residues linked by covalent peptide bonds. Thus proteins consist of a backbone of repeating units where the side chains vary depending on the amino acid residue present at each position (Figure 1.2).

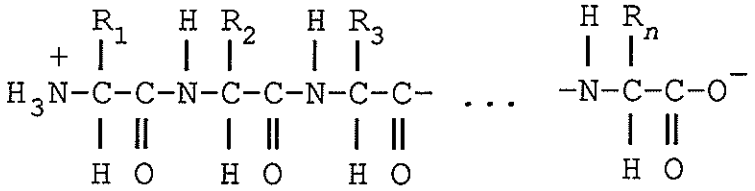


Figure 1.2. Protein structure. The primary structure of a protein consists of amino acid residues linked by covalent peptide bonds.

The biological activity of a protein is due to its three dimensional or tertiary structure. The tertiary structure is determined by the primary structure through interactions of the individual residues with each other, with the solvent and with other cellular components.

Therefore, assuming a complete understanding of biochemistry, the structure of a protein could be determined from its primary sequence and the cellular environment. This is an underlying assumption of attempts to predict protein secondary structure (i.e., structure due to interactions between more or less adjacent residues) from the primary sequence alone (1, 2).[1] However, these methods have not so far been very successful, presumably due to our lack of biochemical understanding and to computational limitations. On the other hand, sequence motif analysis (see next section) assumes very little about the actual chemical interactions determining the biological activity of a protein. Nevertheless, such analyses can often point to functionally important residues conserved among a group of related proteins and thereby provide valuable clues to protein structure and function.

1.3. MOTIFS AS INDICATORS OF STRUCTURE AND FUNCTION

A **motif** is a sequence pattern that occurs frequently within a group of proteins or nucleic acids. For example, the pattern "G . . G . GK" is frequently found in proteins that bind to a purine nucleotide triphosphate (ATP/GTP); therefore this pattern is a motif characterizing ATP/GTP-binding proteins. (A dot '.' indicates that any amino acid residue is allowed at that position in the pattern or motif.) Presumably the amino acid residues that make up such motifs are conserved among related proteins because they play important structural or functional roles. Evidence supporting this hypothesis has been obtained for many motifs. For example, the 3-dimensional structures of two ATP/GTP-binding proteins, adenylate kinase and p21 ras protein, have been determined. Superposition of the ATP/GTP-binding regions reveal a similar role for motif residues of both proteins in substrate binding (3); the lysine (K) and glycine (G) residues in the motif all closely interact with the substrate.

---

[1]The numbers in parentheses in the text indicate references in the Bibliography.

As a second example of the predictive value of motif analysis, Neuwald et al. (4) found that inositol monophosphatase (IMP) and several homologous proteins share two sequence motifs with inositol polyphosphate 1-phosphatase (IPP) (Figure 1.3). They suggested that these motifs correspond to binding sites within IMP and IPP for inositol phosphates or for lithium since both substances are bound by these proteins. This prediction was recently confirmed by a 3-dimensional analysis of IMP (Philip Majerus, personal communication); residues of both motifs are involved in binding to inositol monophosphate. Although many of the proteins homologous[2] to IMP have clear biological roles, there is as yet no experimental evidence that they have phosphatase activity. Nevertheless, the shared motifs strongly suggest that these proteins also bind phosphorylated compounds. Thus, this example illustrates that motif analysis is useful, not only for predicting which residues in a protein play essential functional roles, but also for predicting biological activities for proteins of unknown function.

1.4. PROBLEM STATEMENT

In some cases sequence motifs can be found by visual inspection, but more often they are found by examining alignments of related proteins as is seen in Figure 1.3. However obtaining an alignment for a large number of sequences can be difficult and alignment methods are often unreliable when there is little sequence homology. Thus, one is confronted with the problem of detecting motifs independent of an alignment. This problem can be restated as follows: given a possibly large group of proteins (or nucleic acids), which may be distantly related, find the most significant shared patterns among these sequences. Ideally we would like to find the most biologically significant patterns, but of course this cannot be determined by sequence analysis alone.

---

[2]Two proteins are homologous if they are related through a common ancestoral protein; homology is inferred from extensive sequence similarity.

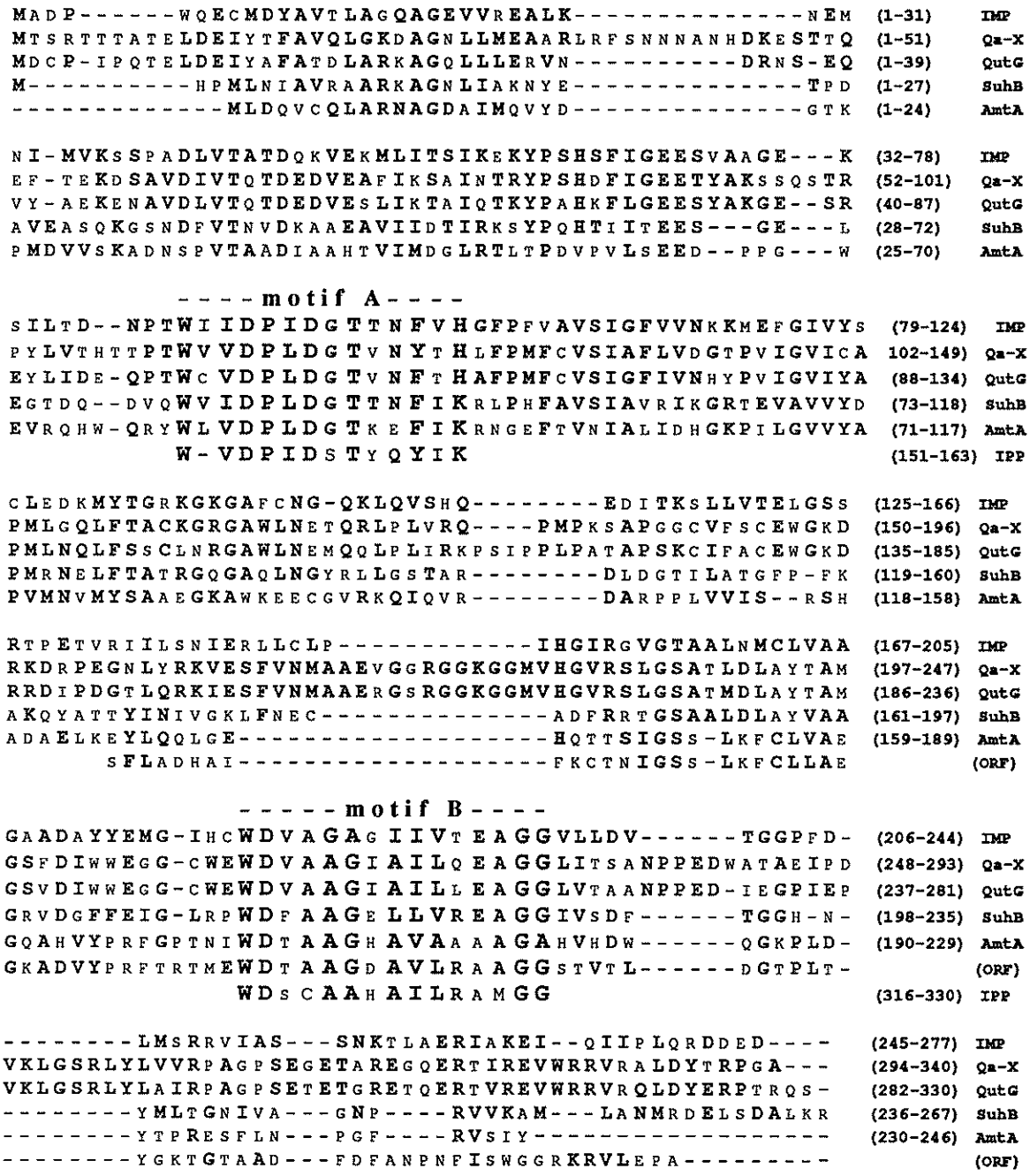


Figure 1.3. Conserved regions between inositol monophosphatase and homologous proteins. Similar residues are bold. Protein sequences shown are bovine inositol monophosphatase (IMP), the products of the *Neurospora crassa qa-x* gene (Qa-X), the *Aspergillus nidulans qutG* gene (QutG) and the *Escherichia coli suhB* and *amtA* genes (SuhB, AmtA) and the C-terminal peptide fragment inferred from the 3' end of an ORF upstream from the *Rhizobium leguminosarum pss* gene. IMP and bovine inositol polyphosphate 1-phosphatase (IPP) are not homologous but they share two motifs (A and B). Conserved residues within motifs are enlarged. For references see (4).

Consequently, the next best thing is to develop a method for detecting statistically significant patterns; a quantitative measure of that significance would be very useful as well.

Fortunately, the development of sequence analysis methods is facilitated by the nature of biological information. Since nucleic acids and proteins consist of strings of residues, they can be readily represented by a formal mathematical model. Then, abstract data types can be developed from the mathematical model and finally, appropriate data structures and algorithms can be found to insure efficient implementation of the method.

1.5. OVERVIEW

The next chapter will review previous sequence analysis methods for detecting motifs. In Chapter 3 a new method is developed which is then verified in Chapter 4 by using it to analyze several groups of proteins. In the final chapter, the method of Chapter 3 is evaluated and compared with other methods and suggestions for future enhancements are discussed. Definitions (Section 6.1) and statistical formulas (Section 6.2), are given in the Appendix .

2. SEQUENCE ANALYSIS METHODS

Many sequence analysis methods involve sequence alignments while a number of more recent methods do not. This chapter reviews both types with an emphasis on those methods that can be used to detect patterns in distantly related sequences.

2.1. METHODS BASED ON ALIGNMENTS

The major purpose behind alignment methods is to determine the "minimum distance" between two sequences. This can be defined as the smallest number of insertions, deletions, and residue substitutions required to change one sequence into another (5). However, with biological sequences it is more typical to view the distance between sequences as the minimum number of mutations required during evolution to transform one sequence into two related sequences. Since mutations are not all equally probable, a scoring scheme is used that puts more weight on more probable mutations.

It is important to realize that these mutational probabilities depend, not only on the probability of the mutation occuring, but also on its selective advantage or disadvantage. For this reason, a residue is often more likely to be replaced by a chemically similar residue than by a dissimilar one. For example, a hydrophobic amino acid residue, which is commonly found in the interior of a protein, is often replaced by another hydrophobic residue because replacement by a hydrophilic residue, which is commonly found on the exterior, may disrupt the protein's three dimensional structure.

The minimal distance between two sequences is determined by finding an optimal alignment (Figure 2.1) using a scoring scheme that places positive weights on matches (identical or similar residues) and negative weights on mismatches (dissimilar residues) and on gaps (insertions or deletions). Usually these match-mismatch weights are stored in a matrix of $|\Sigma| \times |\Sigma|$ cells, where Σ is either the amino acid or nucleotide alphabet, so that the

similarity or dissimilarity of two residues, r_1 and r_2, can be found by looking at the row for r_1 and the column for r_2 (or vice versa). If the weights accurately reflect true mutational probabilities, then the highest scoring (or optimal) alignment reflects the shortest evolutionary path from one sequence to the other.

```
         10         20         30          40         50
GPKEPEVTVPEGDASAGRDIFDSQCSACHAIEGDST--AAPVLGGVIGRKAGQEK-FAYS
.. .   ..: :...:: .:::...:..:::.....  ..: : :..:::..: .  ..::
ASFS---EAPPGNPDAGAKIFKTKCAQCHTVDAGAGHKQGPNLHGLFGRQSGTTAGYSYS
            10         20         30          40         50

 60         70         80         90        100
KGMKGSGITWNEKHLFVFLKNPSKHVPGTKMAFAGLPADKDRADLIAYLK---SV
 . :.....:.:. :. .: ::.:.:::::.:::   .:::::::::::   :
AANKNKAVEWEENTLYDYLLNPKKYIPGTKMVFPGLKKPQDRADLIAYLKKATSS
 60         70         80         90        100        110
```

Figure 2.1. Optimal alignment of cytochromes c. Sequences are from *Tetrahymena pyriformis* (top) and wheat (bottom). Residue identities are indicated by a colon, similarities by a dot and gaps by a dash.

2.1.1. Amino Acid Weighting Schemes

Some amino acid weighting schemes are based on the chemical properties of the amino acids (6) and assume that transformations to similar residues are favored during evolution. Just as English letters can be grouped by similarities in their pronunciation into gutturals (G,K,C), labials (B,P,F), dentals (T,D), etc., amino acids can be grouped by similarities in the chemical properties of their R groups. For example, the side chains of both aspartate (D) and glutamate (E) contain carboxylic acid groups (-COOH) and both alanine (A) and threonine (T) have rather small side chains. Thus amino acid pairs can be given weights proportional to these chemical similarities.

However the list of amino acid similarities and differences is quite long and interrelated. For example, alanine's side chain is short like threonine's, but threonine's side chain contains an hydroxyl group (-OH) like tyrosine's, yet tyrosine's side chain is large and aromatic like phenylalanine's, etc. Sorting out how all these similarities and differences actually influence the ability for some residues to replace others can be very difficult. In

order to circumvent this problem, Dayhoff and coworkers (7, 8) have empirically determined the likelihood that one residue will replace another by examining amino acid substitution patterns in groups of closely related proteins; weights are assigned using these likelihoods.

Determining meaningful gap weights is more difficult, but some rules of thumb have been suggested (9). Note that an optimal alignment for two sequences is a function of the weighting scheme; changing the weights may change the optimal alignment.

2.1.2. Global Alignment Methods

The global alignment problem is to find the best overall alignment for two sequences. This could be done by calculating scores for every possible alignment. However, the combinatorics of such a brute force approach quickly become unmanageable. Fortunately, dynamic programming algorithms can solve such optimization problems in a reasonable amount of time and memory. The first such algorithm was developed by Needleman and Wunsch (10); later Sellers (11, 12) described a slightly different but equivalent algorithm. These methods are guaranteed to find an optimal alignment, but they will not find other optimal or near optimal alignments and thus they may miss significant patterns.

2.1.3. Local Alignment Methods

Local alignments are used to detect conserved regions in proteins or nucleic acids which might in general have little detectable overall homology; unconserved regions do not contribute to the final score. Two approaches to local alignment have been taken. One approach described by Smith and Waterman (13) uses the Needleman-Wunsch method in a modified form and is guaranteed to detect the optimal subsequence alignment.

A second approach is to use an heuristic procedure to find locally similar segments[1] and then to extend these regions until a maximum relatedness score is obtained; such methods

[1] A segment is a contiguous stretch of residues in a sequence.

are very fast. The FASTA program (14, 15) implements this approach by combining a lookup table of the positions of all short *k*-letter words with a 'diagonal' method (16, 17) to locate regions of local similarity. A more recent method, which has been implemented in the program BLAST (18), works by identifying short length *k* segments that are likely to yield high scoring extended regions. First, a list of patterns that yield a 'threshold' relatedness score with a query sequence is created. Next, a database is scanned for segments matching these patterns using a deterministic finite automaton (19). Finally, matched segments are extended and regions scoring above a cutoff score are reported.

2.1.4. MULTIPLE ALIGNMENT METHODS

Pairwise sequence comparisons are useful for alignment of two homologous sequences and for database searches. However, often biologically significant patterns may only be detected by multiple alignment, because a pattern, which may be statistically insignificant when found in only two sequences, can be significant when found in many sequences. When a group of sequences are known to be globally homologous, a number of multiple alignment methods can be applied (20-30). However, the focus of this study is on methods to identify patterns occurring in distantly related sequences; several multiple alignment methods attempt to solve this problem.

2.1.4.1. Heuristic Methods that Add Sequences One at a Time

Bacon and Anderson (6) describe a heuristic method to prune the search space of all possible *n*-wise subsequence alignments to a manageable size. Their method includes a scoring scheme for mismatches based on the chemical attributes of amino acids and involves the following steps. First, all possible alignments of length *k* segments from the first two sequences are evaluated and the best of these are saved on a heap, H_1. Next, all alignments consisting of segment pairs in H_1 are evaluated against all segments from the third sequence, and the best of the three-way alignments are stored on a second heap H_2.

Then all ways of aligning the fourth sequence with the alignments in H_2 are tried, and the best ones replace the segments in H_1. This process continues until either H_1 or H_2 contains the best set of alignments found. This method works even for fairly long patterns in a large numbers of sequences but has several disadvantages. First, the order of the input sequences can influence the output, sometimes dramatically. For example, if the first two sequences happen to be quite distantly related, patterns common to the other sequences may never be detected. Second, the method is limited by the fixed pattern size and the need for all the sequences to contain fairly reasonable matches to the final patterns detected. Stormo and Hartzell (32) describe a related method which is framed in terms of information theory.

2.1.4.2. Methods Used to Detect Ordered Patterns

Sobel and Martinez (33) describe a method that uses a 'longest path' algorithm to find optimal and near optimal alignments of sequences. A path starts at the beginning of the sequences and proceeds toward the end of the sequences. The edges of the path correspond to successive segments common to all the sequences where the segment lengths correspond to edge lengths; no mismatches are allowed and a length penalty is added for gaps between segments. Thus, an optimal alignment for the sequences corresponds to a longest path. Near optimal alignments are found using the method of Byers and Waterman (34) which requires very little additional time and memory space once the optimal alignment is known. This method can be applied to a large number of sequences to detect patterns of variable length; however it does not incorporate a match-mismatch scoring scheme, and it requires that patterns occur in the same order and be present in all the sequences.

Karlin et al. (35) describe a similar method but use linked lists to locate identical length k patterns within a sequence. At each position in the linked list is stored the location of the next segment in the sequence matching the same pattern (null = 0 is stored at the last occurrence of a pattern). By concatenating the input sequences, the problem of finding matches between multiple sequences is reduced to one of finding long repeating

superpatterns within a single sequence. (Errors and gaps are allowed between patterns within the superpattern.) The locations of these longer repeated superpatterns is determined using the pattern information contained in the linked list. Statistical methods are used to distinguish nonrandom sequence relationships from chance configurations.

2.1.4.3. Methods Using Diagonals

Schuler, et al. (36) describe a method for finding regions of local similarity among a group of three or more sequences. First, all input sequences are compared to find 2-diagonals, i.e., alignments of two full-length sequences without gaps, that contain a segment pair with a relatedness score above a low threshold level. Next, the method attempts to construct a 3-diagonal using combinations of three 2-diagonals detected in the first step. This process is continued as long as possible in order to find a maximum *m*-diagonal which contains a set of *m* aligned sequences all of which have relatedness scores to each other above the threshold level. Finally, this *m*-diagonal is parsed, using a heuristic procedure, to locate conserved regions; a statistical criterion (37, 38) is used to aid in the identification of significant regions. This method utilizes rigorous statistics, is not limited to finding patterns of any fixed length, and incorporates a relatedness scoring scheme. A limitation of this method is that the parsing procedure is not always able to disentangle the complex relationships of patterns among the diagonals in order to choose the best conserved region. To get around this problem several tools are provided for interactively editing the output. This method can handle moderate numbers of sequences. Vingron and Argos (39) describe a similar method that uses algebraic manipulations of sequence dot-plots to locate *m*-diagonals.

2.1.5. PROFILE ANALYSIS

Profile analysis (40, 41) is a method that attempts to capture the subtle structural and functional information implicit in alignments of related proteins in order to detect distant

relationships. This information is represented in a position specific scoring table or profile. The score for a specific residue at a given position is determined from the frequency of its occurrence in the entire set of aligned sequences. A relatedness scoring matrix is used to distribute these scores more evenly among similar residues, and penalties are used for insertions or deletions at each position. Thus, the similarity of a query sequence to an entire group of aligned sequences can be tested by comparing that sequence with the profile using a dynamic programming algorithm. This method can also use information from protein structural studies to construct the profile (42). Profile analysis is useful for finding sequences matching known patterns but is not designed to detect new motifs.

2.2. METHODS NOT NEEDING ALIGNMENTS

While alignments are useful for discovering motifs, the most sensitive of these methods can require considerable effort when applied to a large number of sequences, i.e., about ten or more average size proteins, and they may be unreliable when there is little sequence homology. This has motivated development of pattern detection methods that do not require an alignment.

2.2.1. LEXICOGRAPHICALLY ORDERED MATRIX METHODS

In these methods, patterns are ordered lexicographically (and put into one-to-one correspondence with a set of succeeding integers) so that pattern data can be collected into arrays[2].

Queen et al. (43) developed one of the first methods for the detection of patterns in multiple sequences. This method relies on a two dimensional matrix consisting of a one dimensional **integer** array of length $|\Sigma|^N$ for each sequence, where Σ is the alphabet and N is the pattern length; thus, there is one integer in each array for every possible pattern.

[2]An array is an ordered set.

These arrays are used to store the number of segments in each sequence matching a pattern in at least M out of the N positions. A separate array is then used to record the number of sequences matching each pattern. If more than a minimum number, K, of sequences contain the pattern (allowing for the $N - M$ mismatches) the algorithm returns the set of sequences and their locations. The algorithm also allows specification of a maximum distance separating the patterns in the sequences; this feature detects patterns that only occur in corresponding locations for each sequence. This method works reasonably well for even large numbers of nucleotide sequences and allows for mismatches, but its usefulness in the analysis of protein sequences is limited due to the large array sizes needed for the amino acid alphabet. A second limitation is that the patterns must be contiguous and of a fixed and rather short length.

Waterman et al. (44) proposed a closely related method but with improved data analysis that increases its utility. They also generalized the concept of a mismatch into what they call a neighborhood function to describe closely related patterns (this allows for insertion and deletions) and provide a way to estimate the statistical significance. Staden (45) has recently described yet another version of this technique.

2.2.2. Methods Based on Fixed Patterns

Smith, et al. (46) describe a method for finding motifs in groups of functionally related proteins which they have implemented in the MOTIF program. This method involves the following steps:

Step 1: Select all 3-amino acid patterns occurring in at least k of the proteins in the group such that no more than r redundant (internally repeated) copies of the pattern occur.

Step 2: Obtain the pattern scores by measuring the average relatedness value of the protein segments containing those patterns using a Dayhoff relatedness odds (PAM 250) matrix (7, 8).

The highest scoring patterns along with matching segments are reported. (Unmatched segments that have high scoring alignments with the matched segments are also reported.)

This method works fairly well even on large numbers of sequences, and has been incorporated into a method for obtaining a database of the most conserved regions of related proteins (47). In addition, this method has inspired development of a method to be described in Chapter 3. Nevertheless, it has several deficiencies. First, it lacks a statistical basis and therefore cannot give a direct measure of significance. (However, a Monte Carlo simulation routine for empirically determining the statistical significance is included with the MOTIF program.) Second, the method is limited to detection of rather short motifs and does not allow for mismatches in the 3-amino acid patterns of the first step. Third, it is not likely to find motifs which occur in only a minority of the proteins or which are internally repeated. The reason for this is as follows. The parameters k (the minimum number of proteins that must contain a 3-amino acid pattern in Step 1) and r (the maximum number of internal repeats) are used to cut down on detection of random background patterns by relying on two underlying assumptions. [1] True motifs will usually occur in a majority of the proteins and [2] they will tend not to be repeated within a single protein. Thus, in order to eliminate background noise, usually r is small and k is $\geq N \div 2$, where N is the total number of proteins. However, these assumptions are not valid since there are instances of internally repeated motifs (48, 49) and proteins with the same function could belong to several different subgroups sharing distinct motifs due to differing mechanisms. Thus, this method is mostly limited to detecting motifs among groups of proteins that are known to be related before the analysis and, even then, it may miss patterns present only in a minority of the proteins.

An earlier method, which is similar to the MOTIF method, is described by Posfai et al. (50). This method is used to produce a global alignment of a set of similar sequences by performing the following steps: [1] locate the most significant (short) pattern common to all

the sequences; [2] align the sequences at this pattern (this divides each sequence in the set in two so that two sets of half sequences are created); [3] recursively repeat steps 1 and 2 on the two sets of half sequences until no common pattern can be found.

2.2.3. A Method Based on Expectation Maximization

Lawrence and Reilly (51) describe a rigorous statistical method for pattern identification and characterization. Unaligned sequences contain no explicit information on the locations of patterns in a group of sequences. This method employs the "missing information principle" (52) to develop an expectation maximization (EM) algorithm which overcomes this information deficit. The algorithm estimates the probabilities of pattern locations being at each position in each sequence and thereby predicts the most likely sites simultaneously with the maximum likelihood estimates of the model parameters. An advantage of this method is its ability to detect long patterns in a large set of sequences, but this is offset by the need for a fixed pattern size and for each sequence to contain at least one common pattern.

2.2.4. Pattern Matching Programs

A number of methods are available for finding user designated patterns in a database of protein or nucleic acid sequences; for example see (53-55). Some particularly useful methods, which allow for approximate matching to regular expressions (56), have been developed by researchers at the University of Arizona (57).

3. A MODEL FOR DETECTING MOTIFS IN BIOLOGICAL SEQUENCES

From the previous chapter it seems clear that the ideal sequence analysis method has not yet been found. Such a method would [1] be based on rigorous statistics, [2] efficiently detect significant patterns of variable length among large numbers of distantly related sequences, [3] incorporate a relatedness scoring scheme and, as an additional indicator of significance, [4] take into account the order in which the patterns occurred in the sequences. There are other enhancements, which could also be included, but which might either contribute very little or be computationally intractable. For example, the additional complexity needed to directly detect long patterns with gaps could easily become unmanageable; a simpler method might detect these patterns as an ordered arrangement of short contiguous motifs separated by variable length regions.

Although development of an ideal method is not likely in the immediate future, the method described here incorporates a number of key features. It has a rigorous statistical basis and is capable of detecting significant patterns among fairly large groups of distantly related sequences. In addition, it is based on a mathematical model that lays a foundation for further improvements, such as incorporation of a relatedness scoring scheme (see Section 5.2.1).

A model for detecting motifs should include a set of patterns, a population of sequence segments to be matched with those patterns, and a set of operations that perform the matching and determine statistical significance. In this chapter, such a mathematical model, called a generator-population dyad, is described. A generator-population dyad is composed of an aligned segment population, which defines a set of segments (called a universal block) derived from one or more biological sequences, and a pattern generator, which is an abstract object used to generate a set of patterns for analysis. The model also incorporates

an algebraic system for sets of segments (called blocks) matching specific patterns. Before proceeding to a formal description of the model, an informal overview of the basic method will be presented.

3.1. INFORMAL OVERVIEW

A key aspect of the method described here is how to obtain a population of sequence segments of length k to be analyzed; there are at least two ways to do this. One way is to generate a set of overlapping segments from each member of a group of sequences, either proteins or nucleic acids. For example, if one sequence in a group of proteins is:

```
MAGTHFGGKTRRTNMVLCWSTAGPK
```

then the set of corresponding overlapping segments of length 12 would be:

```
MAGTHFGGKTRR
 AGTHFGGKTRRT
  GTHFGGKTRRTN
   THFGGKTRRTNM
    HFGGKTRRTNMV
     FGGKTRRTNMVL
      GGKTRRTNMVLC
       GKTRRTNMVLCW
        KTRRTNMVLCWS
         TRRTNMVLCWST
          RRTNMVLCWSTA
           RTNMVLCWSTAG
            TNMVLCWSTAGP
             NMVLCWSTAGPK
```

Another way is to obtain a set of length k segments having either a functionally important sequence feature or an arbitrary pattern as a fiducial mark for alignment. Examples of functionally important features include sites, in DNA or RNA, where transcription or translation is initiated or terminated, where transposon insertion or RNA splicing occurs, or, in proteins, where phosphorylation, protease cleavage or substrate binding occurs. For example, protein amino-terminal regions aligned at their amino-terminal ends would create a population of segments like the one shown in Figure 3.1.

```
NH2-MDPVVVLVLGLCCLLLLSIWKQNSGRGKLPPGPTPFPIIG  Progesterone 21-monooxygenase
NH2-PRASGNLIPQEKIVNLLNDIFGAGFDTVTTAISWSLMYLV  Cytochrome P450IA2
NH2-MSVSALSSTRFTGSISGFLQVASVLGLLLLLVKAVQFYLQ  Laurate omega-hydroxylase
NH2-MAVLSRMRLRWALLDTRVMGHGLCPQGARAKAAIPAALRD  Vitamin D3 25-hydroxylase
NH2-MVLAGLLLLLTLLSGAHLLWGRWKLRNLHLPPLVPGFLHL  Steroid 21-monooxygenase
   :     :     :     :     :     :              :
NH2-MVNPKHHIFVCTSCRLNGKQQGFCYSKNSVEIVETFMEEL  Ferredoxin (2Fe-2S)
```

Figure 3.1. A population of aligned protein segments containing amino-terminal ends.

Alternatively, protein segments, aligned using a common arbitrary 'seed' pattern, such as the pattern "`.....GK.....`", would create a segment population like the following,

```
TVEYLKGKDNHVAD
RFQSRQGKEQLDLQ
YYSVICGKKIIATR
    :   :   :
LKCKSNGKVKKQHP
```

Once a population of aligned segments for analysis is obtained, the following method can be used to determine whether the occurrence of a given pattern in the population is statistically significant:

Statistical Evaluation Method

Given a pattern and a population of segments, perform the following steps:

STEP 1: Determine the number of segments (n) in the sample population that match the pattern.

STEP 2: Assuming statistical independence (see below), calculate the probability of n or more matches from the cumulative binomial distribution function.

STEP 3: If the probability of obtaining the observed number of matching segments is very small then conclude that the pattern is significant.

Thus, the statistic being measured by this method is the number of segments matching a pattern. The null hypothesis assumes that each segment is independent from other segments and that residues at each location in a segment are independent of residues at other locations. This implies that the statistic is binomially distributed, i.e., models a series of N independent trials where the outcome is either success (the pattern and segment match) or failure (they do not). The probability of n or more successes out of N trials can be calculated from the cumulative binomial distribution function,

$$P(N,n,p) = \sum_{i=n}^{N} \binom{N}{i} p^i (1-p)^{N-i} \qquad (3.1)$$

where p is the product of the frequencies with which the residues at each position in the pattern occur in the segment population as a whole. For example, for the following pattern and set of segments, n out of the N segments match the pattern:

```
PATTERN:      .... YEP ........
                   F

SUCCESS:      FVYF YEP YIPLSETS   1
              MIFA YEP FFLSNGGI   2
               :    :     :
              LILG FEP YGIVGEAS   n

FAILURE:      NAGH ISM KSLKEIPL   n+1
              VNTG HIQ QKIKSDPM   n+2
               :    :     :
              QDVL MKD VTNEFINI   N
```

The probability of n or more successes out of N trials is then calculated from Equation 3.1 where

p = (frequency at which '`Y`' or '`F`' occurs at position 5)

x (frequency at which '`E`' occurs at position 6)

x (frequency at which '`P`' occurs at position 7).

With this as an overview of the basic method, the next sections proceed to model the relationship between patterns and sequences using five abstract data types: sequence entities, pattern generators, blocks, aligned segment populations, and generator-population dyads. In addition to defining each abstract data structure and its associated operations, an attributed context free grammar is given, which describes a command language for generating structures and performing operations. Several key data structures and algorithms used to implement the model are described using the notation of Tarjan (58).

3.2. BIOLOGICAL SEQUENCE ENTITIES

A biological sequence entity is the fundamental object of sequence analysis and represents a protein or nucleic acid sequence that performs a specific biological function within a particular organism.

3.2.1. DEFINITIONS

A **residue** (r) is an element of Σ, where Σ is the alphabet of either nucleotides or amino acids. A biological **sequence** (S) is a string of elements of Σ. Σ^+ denotes the set of all such strings of length ≥ 1. A biological sequence **entity** (E) is an element of the function F_{Seq}: $\mathbf{N} \rightarrow \Sigma^+$ where F_{Seq} maps a unique identification number or identifier ($E.I$) for each biological sequence entity to the entity's sequence ($E.S$) so that $E.S = F_{Seq}(E.I)$. For example, the biological sequence entity, E, where

$E.I = 38574$ and

$E.S$ = "MGDVEKGKKIFVQKCAQCHTVEKGGKHKTGPNLHGLFGRKTGQAAGFSYTDAN
KNKGITWGEDTLMEYLENPKKYIPGTKMIFAGIKKKGERADLIAYLKKATNE",

represents the protein sequence for rat cytochrome c which has been given the identification number 38574.

3.2.2. ATTRIBUTED GRAMMAR AND OPERATIONS

An attributed grammar can be used to parse residues, sequences and entities from an input string. The nonterminal <residue> represents one of the symbols for an amino acid residue[1] (as given in Table 1.1) and is defined by the rule:

<residue>$_{a1}$	→ **A**	[[a1 = **A**]]
	| **C**	[[a1 = **C**]]
	| **D**	[[a1 = **D**]]
	:	
	| **Y**	[[a1 = **Y**]]

(Terminal symbols are given in bold while nonterminal symbols are enclosed in angle brackets.) The subscript symbols a1, a2, etc. represent nonterminal attributes which are defined by the boolean expression enclosed in double square brackets. A sequence can be derived from individual residues by repeated concatenation of amino acid residues. Thus the nonterminal <sequence> is defined by the rule:

<sequence>$_{a1}$	→ <sequence>$_{a2}$ <residue>$_{a3}$	[[a1 = a2 & a3]]
	| <residue>$_{a2}$	[[a1 = a2]]

The symbol '&' represents the concatenation operation. Finally, an entity can be created from a non-negative integer, corresponding to the entity's identification number, and a sequence using the following rule:

<entity>$_{a1}$	→ <integer>$_{a2}$ <sequence>$_{a3}$	[[a1 = <a2,a3>]]

The following operation, which corresponds to this rule, creates a new sequence entity:

make_entity(**integer** *I*, **sequence** *S*): Return **entity** *E* = <*I*,*S*>.

Other operations for retrieving information, and randomizing sequence entities are also defined. Two operations return information about the sequence entity:

identifier(**entity** *E*): Return **integer** *E.I*.

---

[1]A similar rule can be defined for the nucleic acid alphabet.

residue(**integer** *i*,**entity** *E*): Return **residue** *E.S*(*i*).

S(*i*) denotes the *i*th residue of sequence *S* or more generally *A*(*i*) denotes the *i*th element of an array or list.

Randomizing operations are performed on sequence entities in order to facilitate statistical analysis:

permute(**entity** *E*): Randomly permute the residues in *E.S*.

random(**entity** *E*, **real** *dfreq*[1..|*E.S*|][$r \in \Sigma$][$r \in \Sigma$]): Randomize residues in *E.S* consistent with the position dependent di-residue frequencies defined by *dfreq*. (See next section for details.)

3.2.3. IMPLEMENTATION

The operation permute( ) is implemented using a heap (59). The individual residues of sequence *E.S* are inserted into the heap using a random number as the key. (Items are ordered in a heap according to their keys.) Then minimum key residues are sequentially removed from the heap and concatenated to obtain a random permutation of the sequence.

The operation random( ) is implemented as follows. First, the elements of Σ are mapped into an array (**residue** *R*[1..|Σ|]), and an **integer** *i* is initialized to 1. Then, the following three steps are repeated while $i < |E.S|$:

STEP 1: Generate a random real number (**real** *rand*) between 0.0 and 1.0 and initialize the **integer** $n := 1$.

STEP 2: While $rand := rand - dfreq[i][E.S(i)][R(n)] > 0.0 \rightarrow n := n + 1$.

STEP 3: Set $E.S(i+1) := R(n)$ and increment *i*.

This algorithm generates a nonstationary (position dependent) first order Markov chain.

Implementation of the other operations is straightforward.

3.3. PATTERN GENERATORS

Having established a model for biological sequences, a representation for patterns present in those sequences is now formulated. An abstract data type, pattern generator, is used to generate groups of one or more sequence patterns. Sequences, patterns and generators form an hierarchy; just as one or more sequences can match a pattern, one or more patterns can match a generator. Operations and an attributed grammar are defined for three subclasses of especially useful generators.

3.3.1. DEFINITIONS

A biological sequence **pattern** (Q) is defined as a sequence of one or more subsets of Σ:

$$Q \in \left\{ \langle s_1,\ldots,s_k \rangle \mid s_i \in \mathcal{P}(\Sigma) \wedge 1 \leq i \leq k \wedge k \in \mathbf{P} \right\} \tag{3.2}$$

For example, <{F,Y},Σ,{N,D},{P},{P}> is an sequence pattern where Σ is the alphabet of amino acid residues and k=5. A sequence $S = r_1\ldots,r_k$ **matches** a pattern $Q \equiv \langle s_1,\ldots,s_{k'} \rangle$ (and Q matches S) iff $k = k'$ and $r_i \in s_i$ for $1 \leq i \leq k$. Thus, the sequence "YHDPP" matches the above pattern while "YHDPPA" and "YHDGP" do not. A **universal pattern**, $Q_U \equiv \langle \Sigma,\Sigma,\ldots,\Sigma \rangle$, matches any sequence of length $k = |Q_U|$. A **simple pattern** (q) consists of a sequence of complete or singleton sets:

$$q \in \left\{ \langle s_1,\ldots,s_k \rangle \mid s_i \in \{\{r\} | r \in \Sigma\} \cup \{\Sigma\} \wedge 1 \leq i \leq k \wedge k \in \mathbf{P} \right\} \tag{3.3}$$

It is useful to have a notation for a group of one or more patterns. To do this we introduce the concept of a pattern **generator** (G) which is defined as a sequence of one or more subsets of the power set of Σ:

$$G \in \mathbf{G} \equiv \left\{ \langle \sigma_1,\ldots,\sigma_k \rangle \mid \sigma_i \subseteq \mathcal{P}(\Sigma) \wedge 1 \leq i \leq k \wedge k \in \mathbf{P} \right\} \tag{3.4}$$

A pattern $Q \equiv \langle s_1,...,s_k \rangle$ **matches** a generator $G \equiv \langle \sigma_1,...,\sigma_{k'} \rangle$ (and G matches Q) iff $k = k'$ and $s_i \in \sigma_i$ for $1 \leq i \leq k$. For example, the above pattern matches the generator <{{Y,F}},{Σ},{{N,D},{N},{D}},{P},{{r} | r ∈ Σ}>. It is helpful to define a concise notation for generators which eliminates confusing commas and angle brackets, and introduces several abbreviations (Table 3.1).

Table 3.1. Concise notation for pattern generators. All commas and angle brackets are eliminated in the new notation; $\forall x,\forall y$: $r_x,r_{x_y} \in \Sigma$.

NEW NOTATION	OLD NOTATION	COMMENT
$\{r_{x_1}r_{x_2}...r_{x_i}:..\quad..:r_{x_j}...r_{x_n}\}$	$\{\{r_{x_1},r_{x_2},...,r_{x_i}\},...$ $,\{r_{x_j},...,r_{x_n}\}\}$	a non-singleton generator set
'?'	$\{r_1:r_2:...:r_{\|\Sigma\|}\} \equiv \{\{x\} \mid x \in \Sigma\}$	the set of all 1-residue sets
$\{r_{x_1}r_{x_2}...r_{x_n}\}$	$\{\{r_{x_1},r_{x_2},...,r_{x_n}\}\}$	a multi-residue singleton set
r_x	$\{\{r_x\}\}$	a 1-residue singleton set
'.'	$\{\Sigma\}$	the singleton set containing Σ

Thus the above generator would be designated as "{YF}.{ND:N:D}P?", using this notation. A **null generator** ($G_\emptyset$) is a sequence of 0 sets.

There are three types of generators which are particularly useful for the analysis of biological sequences: 1-pattern generators, variable generators, and 2×2-table generators.

A **1-pattern generator** (G_Q) matches only one pattern Q. For example, the generator ".{ND}PPY" can only be matched with the pattern <Σ,{N,D},{P},{P},{Y}>, and therefore is a 1-pattern generator. The concise notation for a 1-pattern generator will often be used to denote the single matching pattern. The **universal generator**, $G_{Q_U} \equiv .^k$, is a 1-pattern generator that matches the universal pattern Q_U of length k.

A **variable generator** (G_v) is derived from a 1-pattern generator G_Q by converting one or more '.'s at arbitrary positions in G_Q into '?'s. For example, the variable residue

generator G_v = ". ? . ?PPY" is derived from the 1-pattern generator G_Q = ". . . . PPY" by converting two '.'s in G_Q into '?'s.

A **2×2-table generator** (G_t) is derived from a 1-pattern generator G_Q by 2-partitioning the elements of two multi-residue singleton sets in G_Q to produce a 4-pattern generator. (Note that a 1-pattern generator, G_Q, is a sequence of singleton sets where $G_Q(i) = \{Q(i)\}$.) For example, the table generator G_t = ". {ILV:YF} . {N:D}PPY" is derived from the 1-pattern generator G_Q = ". {ILVYF} . {ND}PPY" by 2-partitioning the elements of the multi-residue singleton sets {{I,L,V,Y,F}} and {{N,D}}. The four patterns recognized by G_t can be mapped to the cells of a 2-dimensional table with the bins of the rows and columns of the table corresponding to the bins of the partitioned sets:

	{N}	{D}
{I,L,V}	.{ILV}.NPPY	.{ILV}.DPPY
{Y,F}	.{YF}.NPPY	.{YF}.DPPY

Such tables are represented by the abstract data type **table** which is defined in the Appendix (Section 6.3) and represents a 2×2 contingency table on which statistical analysis operations can be performed.

3.3.2. Operations and Attributed Grammar

The operations and attributed grammar described here can only create the three specific types of generators mentioned above; in the future, operations and grammatical rules for other types of generators may also be developed.

Since we are only interested in patterns with reference to some set of biological subsequences of length k, it is helpful to require that all patterns also be of length k. Thus the following operation creates a null generator $G\emptyset$ but requires that a total of k append operations be performed on $G\emptyset$:

make_generator(**integer** k): Create and return a null generator, $G\emptyset$, such that a total number of k append operations are required.

If $i < k$ append operations are performed before a non-append operation occurs a silent operation is done that sets $G := G \,\&\, .^{k-i}$. This eliminates the need to specify all k positions of G either within the command input string or within higher level operations. If more than k append operations are performed on G, the generator is destroyed, since it cannot recognize any length k pattern, and **null** is returned by further operations. The following append operations are possible:

append_set(**generator** G, **set** V): Append $\{V\}$, where $V \subseteq \Sigma$ to the end of G, i.e. $G := G \,\&\, \{V\}$.

append_variable(**generator** G): Append the set '?' to the end of generator G, i.e., $G := G \,\&$ '?'.

append_partition(**generator** G, **2-partition** $\{U,V\}$): Append the 2-partitioned residue set $\{U,V\}$ where $U,V \subseteq \Sigma$ and $U \cap V = \emptyset$ to the end of generator G, i.e., $G := G \,\&\, \{U,V\}$. The attributed grammar for G insures that two and only two partition operations are performed during creation of a table generator (see below).

Using these operations all 1-pattern, variable, and table generators can be created from input strings defined by the following attributed grammar; these input strings correspond to the concise notation for generators.

The grammar rule for generating 1-pattern generators is:

<1p_generator>$_{a1}$ → <1p_generator>$_{a2}$ <set>$_{a3}$ [[a1 = a2 & {a3}]]
| <univ_generator>$_{a2}$ <subset>$_{a3}$ [[a1 = a2 & {a3}]]

where the non-terminals <set>, <subset>, and <univ_generator> are defined by:

<set>$_{a1}$ $\rightarrow$ <subset>$_{a2}$	[[a1 = a2]]
| '.'	[[a1 = Σ]]
<subset>$_{a1}$ $\rightarrow$ '{' <residues>$_{a2}$ '}'	[[a1 = a2]]
| <residue>$_{a2}$	[[a1 = {a2}]]
<residues>$_{a1}$ $\rightarrow$ <residues>$_{a2}$ <residue>$_{a3}$	[[a1 = a2 $\cup$ {a3}]]
| <residue>$_{a2}$	[[a1 = {a2}]]
<univ_generator>$_{a1}$ $\rightarrow$ <univ_generator>$_{a2}$ '.'	[[a1 = a2 & {Σ}]]
|	[[a1 = $G\emptyset$]]

Rules for generating variable generators are defined in terms of 1-pattern generators:

<var_generator>$_{a1}$ $\rightarrow$ <var_generator>$_{a2}$ <set>$_{a3}$	[[a1 = a2 & {a3}]]
| <var_generator>$_{a2}$ '?'	[[a1 = a2 & {{x} | $x \in \Sigma$}]]
| <1p_generator>$_{a2}$ '?'	[[a1 = a2 & {{x} | $x \in \Sigma$}]]
| <univ_generator>$_{a2}$ '?'	[[a1 = a2 & {{x} | $x \in \Sigma$}]]

Rules for generating table generators require the non-terminal, <part_generator>, for creating an intermediate generator with a single partitioned set; a 2-partitioned set is represented by the non-terminal, <partition>:

<tab_generator>$_{a1}$ $\rightarrow$ <tab_generator>$_{a2}$ <set>$_{a3}$	[[a1 = a2 & {a3}]]
| <part_generator>$_{a2}$ <partition>$_{a3}$	[[a1 = a2 & a3]]
<part_generator>$_{a1}$ $\rightarrow$ <part_generator>$_{a2}$ <set>$_{a3}$	[[a1 = a2 & {a3}]]
| <1p_generator>$_{a2}$ <partition>$_{a3}$	[[a1 = a2 & a3]]
| <univ_generator>$_{a2}$ <partition>$_{a3}$	[[a1 = a2 & a3]]
<partition>$_{a1}$ $\rightarrow$ '{' <residues>$_{a2}$ ':' <residues>$_{a3}$ '}'	[[a1 = {{a2},{a3}}]]

A semantic check must be done to insure that the partitioned residue sets are disjoint.

The following operations are used to modify a 1-pattern generator G by redefining its ith set:

def_residue(**residue** r, **integer** i, **generator** G): Redefine G so that

$G(i) := \{\{r\}\}$.

add_residue(**residue** *r*, **integer** *i*, **generator** *G*): Redefine *G* so that $G(i) := \{V \cup \{r\}\}$ where $\{V\} = G(i)$ before the operation.

def_dot(**integer** *i*, **generator** *G*): Redefine *G* so that $G(i) :=$ '.'.

The following operations return information about the 1-pattern generator, G_Q:

member(**residue** *r*, **integer** *i*, **generator** G_Q): If $r \in Q(i) \in G_Q(i)$ return true; otherwise return false.

show_pattern(**generator** G_Q): Show the pattern Q.

The merge( ) operation can be used to merge two simple patterns. Two generators, G_q and $G_{q'}$, are **mergable** iff they are both simple 1-pattern generators where $|G_q| = |G_{q'}|$ and for the corresponding simple patterns, q and q': $(\forall i)(q(i) \neq q'(i) \rightarrow (q(i) \neq \Sigma \wedge q'(i) \neq \Sigma))$ and there exists one and only one $1 \leq j \leq |G_q|$ such that $q(j) \neq q'(j)$. The merged pattern, Q, is obtained from q and q' by setting $Q(i) := q(i) \cup q'(i)$ for $1 \leq i \leq |q|$. For example, merging the generators ".NPPY" and ".DPPY" yields ".{ND}PPY". The merge( ) operation is used during an exhaustive search in order to find patterns having either of two amino acids at a single position (see Section 3.6).

merge(**generator** G_q, **generator** $G_{q'}$): If G_q and $G_{q'}$ are mergable then merge them and return the generator for the merged pattern; otherwise return **null**.

The operation is_variable( ) determines whether the *i*th set of G is a '?':

is_variable(**integer** *i*, **generator** G_v): Return true if $G_v(i) =$ '?'; otherwise return false.

The following operations return information about the table generator G_t:

gen_table(**generator** G_t): Return a 2 x 2 table corresponding to the table generator G_t. The table data type is described in the Appendix (Section 6.3).

tab_pattern(**integer** R, **integer** C, **generator** G_t): Return the 1-pattern generator for the pattern found in the Rth row and the Cth column of the 2 x 2 contingency table for generator G_t. If G_t is not a table generator or if $R,C \notin \{0,1\}$ then **null** is returned.

3.3.3. IMPLEMENTATION

Since only three subclasses of pattern generators are used, this allows the generator data type to be implemented using a simpler data structure than would be required to implement every possible generator. A 1-pattern generator, G_Q, is implemented as the corresponding pattern Q (i.e., as an array of sets), so that the set $G_Q(i)$ is the singleton set containing the ith element in the array = $\{array(i)\} = \{Q(i)\}$. Each of the sets in this array is implemented using a bitarray of 32 bits. This accommodates an alphabet of up to 32 symbols where each symbol maps to a specific bit.

Variable generators are implemented using a modification of the same data structure where a variable set ('?') is indicated by a bitarray of 32 zero bits (the empty set is not otherwise needed in a pattern or generator).

A table generator is implemented using another modification: a pointer p to a table data structure is used where $p \neq$ **null** implies that G is a table generator. Recall that G_t is derived from a 1-pattern generator, G_Q, by 2-partitioning two sets in Q and that the four patterns recognized by G_t can be mapped to the cells of a 2x2 table whose rows and columns correspond to the partitioned sets of Q. Thus, the partitioned sets can be stored as part of the table data structure. The two bitarrays in G_t corresponding to the partitioned sets are set to 32 zero bits.

The table generator operations are implemented as follows. The gen_table(G_t) operation simply returns a pointer to the **table** data structure; tab_pattern(R,C,G_t) first creates a copy of the array structure for G_t. Next, it calls a table data type operation that returns the Rth bin in the row partition and the Cth bin in the column partition. The information in these sets is used to turn on the appropriate bits in the copied array, thereby creating the 1-pattern generator to be returned.

Other generator operations can be implemented by operating on bitarrays in a straightforward manner. Thus, all three generator types can be implemented using a single basic data structure.

3.4. BLOCKS

Starting with the models for individual sequences and patterns, another abstract data type, block, is formulated to represent a set of subsequences that match a particular pattern.

3.4.1. DEFINITIONS

A **segment** (e) is a 3-tuple[2] $e = \langle I$: **N**,o: **N**,k: **P**$\rangle$ where $e.I$ is the identifier ($E.I$) for some sequence entity E, $e.o$ is the offset of the segment from the start of the sequence $E.S$ such that $0 \le e.o \le |E.S| - e.k$, and $e.k$ is the segment length. Thus e corresponds to a subsequence, S, via the function f_{seg}: **N**×**N**×**P** → Σ^* such that

$$S = f_{seg}(e) = r_{e.o+1}r_{e.o+2}...r_{e.o+e.k} \text{ where there exists an } \textbf{entity } E \text{ such that}$$

$$E.S = F_{Seq}(e.I) = r_1r_2...r_n \wedge 0 \le e.o \le n - e.k.$$

($\Sigma^* = \Sigma^+ \cup \{\lambda\}$ where λ is the string of length zero.) For example, $f_{seg}(\langle 459,1,5\rangle)$ = "GDVEK" where E = <459,"AGDVEKPWQLIL">.

---

[2]The notation for a k-tuple includes the set (following the colon) from which each element is derived.

Let β_k be the set of all length k segments derivable from an arbitrary set of biological sequence entities Π:

$$\beta_k \equiv \{\ e = \langle I\colon \mathbf{N}, f\colon \mathbf{N}, k\colon \mathbf{P}\rangle \mid \langle e, S\rangle \in f_{seg} \wedge e.k = k \wedge e.I = E.I \ \wedge\ E \in \Pi\ \} \qquad (3.5)$$

Let the **universe** U be an arbitrary subset of β_k; then, given a pattern Q, a **block** B_Q is a subset of U such that $e \in U$ is a member of B_Q if and only if subsequence $S = f_{seg}(e)$ matches the pattern Q:

$$B_Q = \{\ e \in U \mid f_{seg}(e) \text{ matches } Q\ \}$$

Note that $Q = .^k$ implies $B_Q = U$; for this reason U is also called the **universal block** (B_U). The **null block** ($B_\emptyset$) is the empty set $\emptyset$.

A convenient way to represent a block is to use the concise notation for the 1-pattern generator of Q as a subscript to the block symbol. For example, $B_Q = B_{..\mathrm{PPY}}$, designates the set of segments in the universe matching the pattern $Q =$ "`..PPY`". Thus, if the universe is defined as

$$U = B_U = B_{.....} = \{\ e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6,\ e_7,\ e_8\ \}$$

where

$$f_{seg}(e_1) = \texttt{"ATPPY"},\quad f_{seg}(e_2) = \texttt{"HIPSN"},\quad f_{seg}(e_3) = \texttt{"SVPPY"},$$
$$f_{seg}(e_4) = \texttt{"AGEMA"},\quad f_{seg}(e_5) = \texttt{"FTPPY"},\quad f_{seg}(e_6) = \texttt{"ATLPY"},$$
$$f_{seg}(e_7) = \texttt{"CTPWY"},\quad f_{seg}(e_8) = \texttt{"ATDKR"}$$

then $B_{..\mathrm{PPY}} = \{\ e_1,\ e_3,\ e_5\ \}$.

Before defining operations on blocks, an algebraic system for deriving an arbitrary block, B_Q, from a small set of elementary blocks will be described.

3.4.2. AN ALGEBRA FOR BLOCKS

In order to detect a conserved pattern, called a **motif**, among a group of biological sequences it is important to determine the probability of finding a block of $|B_Q|$ segments in

B_U matching the pattern Q. Assuming independence of sequence segments and of residues at different positions in each segment, this can be obtained from the cumulative binomial distribution function:

$$\sum_{i=|B_Q|}^{|B_U|} \binom{|B_U|}{i} p^i (1-p)^{|B_U|-i} \tag{3.6}$$

where p is the product of the frequencies with which the residues at individual positions in the universal block match the pattern at those positions. (For a rigorous definition of p see Equation 3.9 below.)

An algebraic system for blocks can be used to determine which segments in a universal block match an arbitrary pattern Q and from this the corresponding values of p and $|B_Q|$ in Equation 3.6. This algebra requires that an array of elementary sequence blocks, **block** $b[r \in \Sigma][1..k]$, be created such that $b[r][i]$ contains $e \in \beta_k$, if and only if, for $S = f_{seg}(e)$, $S(i) = r$. (β_k is defined in terms of some set of entities, Π, by Equation 3.5.) An alternative representation for elementary blocks is to use, as a subscript to b, the concise notation for the generator with $\{\{r\}\}$ at position i and $\{\Sigma\}$ at all other positions. For example, the array of elementary blocks of length 5 (using the amino acid alphabet) is given in Table 3.2. Starting from these elementary blocks, a block, B_Q, for an arbitrary pattern, Q, can be obtained by performing the basic operations of set union and set intersection using the following equation:

$$B_Q = B_U \cap \left(\bigcap_{i=1}^{k} \left(\bigcup_{r \in Q(i)} b[r][i] \right) \right) \tag{3.7}$$

For example, the block $B_{.\{ND\}PPY}$ can be derived from elementary blocks $b_{.N...}$, $b_{.D...}$, $b_{..P..}$, $b_{...P.}$, and $b_{....Y}$, and the universal block $B_{.....} \equiv B_U$, by the following formula:

$$B_{.\{ND\}PPY} = B_{.....} \cap (b_{.N...} \cup b_{.D...}) \cap b_{..P..} \cap b_{...P.} \cap b_{....Y}$$

Table 3.2. Elementary blocks of length 5 for amino acids.

$r\backslash i$	1	2	3	4	5
A	$b_{A....}$	$b_{.A...}$	$b_{..A..}$	$b_{...A.}$	$b_{....A}$
C	$b_{C....}$	$b_{.C...}$	$b_{..C..}$	$b_{...C.}$	$b_{....C}$
D	$b_{D....}$	$b_{.D...}$	$b_{..D..}$	$b_{...D.}$	$b_{....D}$
E	$b_{E....}$	$b_{.E...}$	$b_{..E..}$	$b_{...E.}$	$b_{....E}$
F	$b_{F....}$	$b_{.F...}$	$b_{..F..}$	$b_{...F.}$	$b_{....F}$
G	$b_{G....}$	$b_{.G...}$	$b_{..G..}$	$b_{...G.}$	$b_{....G}$
H	$b_{H....}$	$b_{.H...}$	$b_{..H..}$	$b_{...H.}$	$b_{....H}$
I	$b_{I....}$	$b_{.I...}$	$b_{..I..}$	$b_{...I.}$	$b_{....I}$
K	$b_{K....}$	$b_{.K...}$	$b_{..K..}$	$b_{...K.}$	$b_{....K}$
L	$b_{L....}$	$b_{.L...}$	$b_{..L..}$	$b_{...L.}$	$b_{....L}$
M	$b_{M....}$	$b_{.M...}$	$b_{..M..}$	$b_{...M.}$	$b_{....M}$
N	$b_{N....}$	$b_{.N...}$	$b_{..N..}$	$b_{...N.}$	$b_{....N}$
P	$b_{P....}$	$b_{.P...}$	$b_{..P..}$	$b_{...P.}$	$b_{....P}$
Q	$b_{Q....}$	$b_{.Q...}$	$b_{..Q..}$	$b_{...Q.}$	$b_{....Q}$
R	$b_{R....}$	$b_{.R...}$	$b_{..R..}$	$b_{...R.}$	$b_{....R}$
S	$b_{S....}$	$b_{.S...}$	$b_{..S..}$	$b_{...S.}$	$b_{....S}$
T	$b_{T....}$	$b_{.T...}$	$b_{..T..}$	$b_{...T.}$	$b_{....T}$
V	$b_{V....}$	$b_{.V...}$	$b_{..V..}$	$b_{...V.}$	$b_{....V}$
W	$b_{W....}$	$b_{.W...}$	$b_{..W..}$	$b_{...W.}$	$b_{....W}$
Y	$b_{Y....}$	$b_{.Y...}$	$b_{..Y..}$	$b_{...Y.}$	$b_{....Y}$

The probability (assuming statistical independence) of finding a particular block B_Q given a universal block, B_U, can be determined by using the cumulative binomial distribution function and equations involving elementary blocks:

$$P(B_Q|B_U) = \sum_{i=|B_Q|}^{|B_U|} \binom{|B_U|}{i} p^i (1-p)^{|B_U| - i} \tag{3.8}$$

$$\text{where} \quad p = \prod_{i=1}^{k} \frac{\left| B_U \cap \left(\bigcup_{r \in Q(i)} b[r][i] \right) \right|}{|B_U|} \tag{3.9}$$

The set of segments matching an arbitrary pattern and the associated probability can be determined concurrently by recursively performing calculations involving Equations 3.7 - 3.9. For example, by combining a depth first search procedure (59) with these calculations

on elementary blocks, the blocks matching every possible simple pattern and their probabilities can be found. The root node of the search tree is obtained by selecting certain segments of interest from among all the segments (β_k) for a given set of entities (Π), thereby creating a universal block, $B_U \subseteq \beta_k$. The value of p is initialized to 1.0. Then the space of all simple patterns can be explored by recursively generating the children of each node, starting with the root, by taking the intersection of that node with elementary blocks (see Figure 3.2). At the same time the value of p_{child} for each node is determined by setting $p_{child} := p_{parent} \times |B_U \cap b[r][i]| \div |B_U|$ where the child node was derived from a parent node by using the elementary block $b[r][i]$. Thus the probability at each node can be obtained from the values of $|B_U|$, p and $|B_Q|$ using the cumulative binomial distribution function. The depth first search procedure saves time by eliminating the need to recalculate p and B_Q for ancestor nodes while searching descendant nodes.

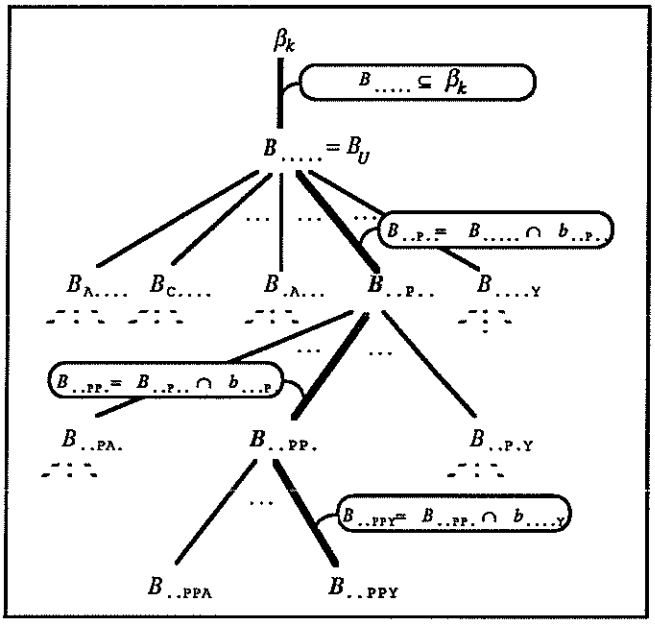


Figure 3.2. Depth first pattern search using elementary blocks.

3.4.3. OPERATIONS

The following operation creates a **null** block:

make_block( ): Create and return the **null** block, $B_{\emptyset}$.

Set operations for blocks are:

member_block(**segment** e, **block** B): Return true if $e \in B$; otherwise return false.

add_segment(**segment** e, **block** B): Assign $B := B \cup \{e\}$.

delete(**segment** e, **block** B): Assign $B := B - \{e\}$.

cardinality(**block** B): Return $|B|$.

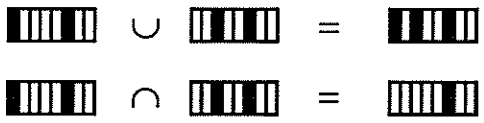
intersect(**block** $B1$, **block** $B2$, **block** B): Assign $B := B1 \cap B2$.

card_intersect(**block** $B1$, **block** $B2$, **block** B): Assign $B := B1 \cap B2$ and return $|B|$.

union(**block** $B1$, **block** $B2$, **block** B): Assign $B := B1 \cup B2$.

3.4.4. IMPLEMENTATION

If an order is imposed on the set of all segments, β_k, then these segments can be represented as an array of $|\beta_k|$ bits. This allows block intersect and union operations to be done in parallel, 8 bits (1 byte) at a time.



Even greater efficiency is achieved by maintaining a list of those bytes which contain at least one element; operations on **null** bytes need not be done since the intersection of a **null** set with any set is always **null** (i.e., $s \cap$ **null** = **null** for any set s) (see Figure 3.3). This speeds up succeeding intersect operations since the bit array gets increasingly sparse with each intersection. A globally defined lookup table (**integer** cardinality[1..255]) can

be used to determine the cardinality of each non-**null** byte (see Figure 3.4). An important algorithm which uses this data structure to perform the card_intersect( ) operation is given in Figure 3.5.

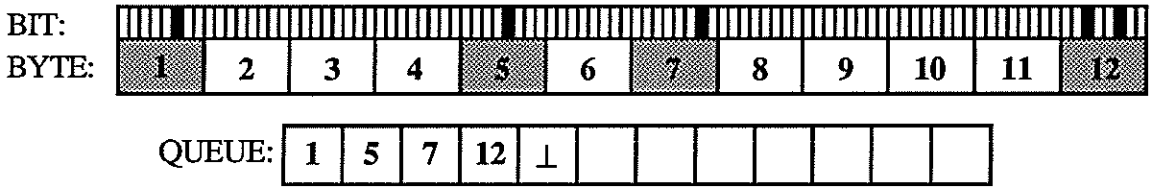


Figure 3.3. Representation of segments in a block by an array of bits. Intersections are performed only on non-**null** bytes by maintaining a list.

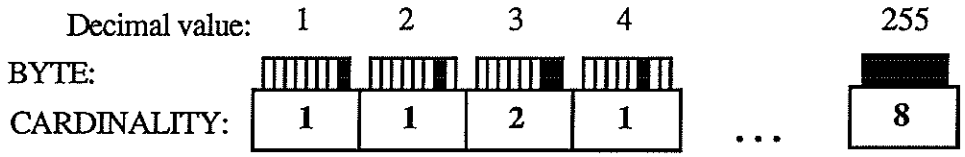


Figure 3.4. A lookup table for determining the cardinality of a non-**null** byte.

```
integer function card_intersect(set S1,S2, modifies set S)
    integer i,n := 0;
    S.queue := [ ];
    for i ∈ S2.queue →
        S.byte[i] := S1.byte[i] ∩ S2.byte[i];
        if S.byte[i] ≠ { } →
                S.queue := S.queue & [i];
                n := n + cardinality[S.byte[i] ];
        fi;
    rof;
    return n;
end card_intersect;
```

Figure 3.5. An algorithm for calculating both the intersection of two sets and the cardinality of the resultant set using a list.

3.5. ALIGNED SEGMENT POPULATIONS

Using the model just developed for blocks an abstract data type, aligned segment population, is formulated to represent a set of sequence segments to be searched for patterns of potential biological significance.

3.5.1. DEFINITION

An aligned **segment population** (**seg_pop** P) is defined as a 4-tuple, $<\Pi,k,B_U,\Sigma>$, where Π is a set of biological sequence entities and B_U is a subset of β_k, the set of all length k segments derived from Π (see Equation 3.5), and Σ is the amino acid or nucleotide alphabet. A **null segment population** ($P_\emptyset$) is a **seg_pop** with $\Pi = \emptyset$, $k = 0$ and consequently with $B_U = \emptyset$ (i.e., $P_\emptyset = <\emptyset,0,\emptyset,\Sigma>$).

3.5.2. ATTRIBUTED GRAMMAR AND OPERATIONS

A population can be created from an input string of biological sequence entities defined by the following attributed grammar:

$$\text{<seg\_pop>}_{a1} \rightarrow \text{<new\_pop>}_{a2} \qquad [\![\ a1 = <a2.\Pi, k_{default}, \beta_{k_{default}}, a2.\Sigma>\ ]\!]$$

$$\begin{aligned} \text{<new\_pop>}_{a1} \rightarrow\ & \text{<new\_pop>}_{a2}\ \text{<entity>}_{a3} && [\![\ a1 = <a2.\Pi \cup \{a3\},0,\emptyset,a2.\Sigma>\ ]\!] \\ & \mid \text{<entity>}_{a2} && [\![\ a1 = <\{a2\},0,\emptyset,\Sigma>\ ]\!] \end{aligned}$$

where Σ and the value of $k_{default}$ is globally defined. Three operations corresponding to these rules are:

make_pop(**set** Σ): Create and return a null segment population $P_\emptyset$.

add_pop(**entity** E, **seg_pop** P): Assign $P.\Pi := P.\Pi \cup \{E\}$.

reset(**integer** k', **seg_pop** P): Assign $P.k = k'$ and $P.B_U = \beta_{k'}$.

After the universal block is initialized to β_k using the reset( ) operation, several other operations can be applied to modify B_U. Two of these operations require specification of the set of segments which is to be retained or removed from B_U:

select(**block** B, **seg_pop** P): Assign $P.B_U := P.B_U \cap B$.

remove(**block** B, **seg_pop** P): Assign $P.B_U := P.B_U - B$.

Another selection operation, purgeH( ), removes "low entropy" segments from the population. Such segments contain a disproportionate number of certain residues. For example, the segment "GPGGPAPGGPSPGG" has a high proportion of glycines and prolines. Such segments interfere with the analysis by contradicting underlying assumptions about statistical independence. purgeH( ) removes all segments from the population above a cutoff "entropy score" which is based on the negative logarithm of the multinomial probability function. (For a more detailed description of entropy scores see the section on statistical formulas in the Appendix, Section 6.3.)

purgeH(**integer** x, **seg_pop** P): Remove all $e \in P.B_U$ with "entropy score" above x.

Often the detection of patterns conserved among distantly related[3] proteins or nucleic acids is more useful for determining functionally or structurally important residues. The operation blast( ) insures that $P.B_U$ contains only segments derived from more distantly related sequence entities in $P.\prod$; it uses a method developed by Altschul et al. (18).

blast(**integer** x, **seg_pop** P): Group all entities $E,E' \in P.\prod$ with a sequence similarity BLAST (PAM 120) score greater than x into the same equivalence class. Select one entity $\in P.\prod$ from each class. Remove every segment $e \in P.B_U$ which was derived from an unselected entity.

[3]Two sequences are distantly related if they share little similarity based on a relatedness score as described in Section 2.1.

The universal block can be changed back to its initial state using the revert( ) operation:

revert(**seg_pop** P): Add back all $e \in \beta_k$ previously removed from B_U.

Two operations randomize sequence entities in the segment population:

shuffle(**seg_pop** P): Revert(P); call the operation permute(E) on all $E \in P.\prod$ and reset(P).

randomize(**seg_pop** P): Revert(P); call the operation random(E, dfreq(P)) on all $E \in P.\prod$ and reset(P) (assumes that all $E.S$ are of the same length). The dfreq(P) operation is defined below.

The grammar used to create a segment population can be extended to allow for modification of that population using the operations just defined:

<seg_pop>a1 → <new_pop>a2
[[a1 =<a2.$\prod$,$k_{default}$, $\beta_{k_{default}}$, a2.Σ>]]
| **select** <1p_generator>a2 <seg_pop>a3
[[a1=<a3.$\prod$, a3.k, a3.$B_U \cap B_{a2}$, a3.Σ>]]
| **remove** <1p_generator>a2 <seg_pop>a3
[[a1=<a3.$\prod$, a3.k, a3.$B_U - B_{a2}$, a3.Σ>]]
| **purge** <integer>a2 <seg_pop>a3
[[a1=<a3.$\prod$, a3.k, $B_{\text{purgeH}_{a2}}$, a3.Σ>]]
| **blast** <integer>a2 <seg_pop>a3
[[a1=<a3.$\prod$, a3.k, $B_{\text{BLAST}_{a2}}$, a3.Σ>]]
| **revert** <seg_pop>a2
[[a1 = <a2.$\prod$, a2.k, $\beta_{a2.k}$, a2.Σ>]]
| **shuffle** <seg_pop>a2
[[a1=<$\prod_{\text{shuffled}}$, a2.k, $\beta_{a2.k}$, a2.Σ>]]
| **randomize** <seg_pop>a2
[[a1=<$\prod_{\text{randomized}}$, a2.$k$, $\beta_{a2.k}$, a2.Σ>]]
| **reset** <integer>a2 <seg_pop>a3
[[a1=<a3.$\prod$, a2, β_{a2}, a3.Σ>]]
| [[a1=***Pop***]]

The non-terminal <1p_generator> defines the block of segments to be selected or removed from the segment population. The operation reset( ) can be performed whenever a pattern

search using a different segment length is desired. The global variable **seg_pop** ***Pop*** allows access to the segment population.

Three operations return information about the segment population.

freq(**seg_pop** P): Return a $k \times |\Sigma|$ array of real numbers where cell[i][r] of the array is the frequency with which residue r is found at position i in B_U.

dfreq(**seg_pop** P): Return a $k \times |\Sigma| \times |\Sigma|$ array of real numbers where cell[i][r][s] of the array is the frequency with which the di-residue rs is found at position i in B_U.

elementaryblocks(**seg_pop** P): Return the array of elementary blocks $b[r \in \Sigma][1..k]$, where $k = P.k$ *and where* $e \in \beta_k$ is an element of $b[r][i]$ iff $S(i) = r$ where $S = f_{seg}(e)$.

The array returned by the freq( ) operation can be used for calculating the value of p for the cumulative binomial distribution function (Equation 3.8); this saves time by eliminating repeat calculations of $|B_U \cap b[r][i]| \div |B_U|$ in Equation 3.9. The array returned by the dfreq( ) operation is used as an argument for the random( ) operation of the sequence entity abstract data type.

3.5.3. IMPLEMENTATION

The segment population data structure is implemented using an array of type **segment** for β_k as was mentioned in Section 3.4 and an array of type **entity** for $P.\prod$. Segments in β_k are ordered and mapped to the numbers from 1 to $|\beta_k|$. Then each segment, e, is examined, using the segment offset ($e.o$) and the residue( ) operation, to determine which segments are to be added to which elementary blocks. The selection operations, select( ), remove( ), purgeH( ), blast( ) and revert( ), are implemented by calling block operations and modify only the universal block $P.B_U$. The disjoint sets data structure described by

Tarjan (58) is used to group sequence entities into equivalence classes for the blast( ) operation. The shuffle( ) and randomize( ) operations are implemented by calling the **entity** operations permute( ) and random( ), respectively. Values of cells in the array returned by freq( ) are calculated by determining the value of $freq[i][r] = |B_U \cap b[r][i]| \div |B_U|$ for $1 \leq i \leq k$ and $r \in \Sigma$. Similarly, values of cells in the dfreq array are calculated by determining the value of $dfreq[i][r][s] = |B_U \cap b[r][i] \cap b[s][i+1]| \div |B_U|$ for $1 \leq i < k$ and all $r,s \in \Sigma$.

3.6. GENERATOR-POPULATION DYADS

Here the final form of the model is described which combines a segment population and a pattern generator into one structure (a generator-population dyad). This allows various questions to be asked about the segment population using patterns specified either by the primary generator or by secondary generators derived from the primary generator.

3.6.1. DEFINITIONS

A generator-population **dyad** (D) is defined as a 2-tuple $<P,G>$ where $D.P$ is an aligned segment population and $D.G$ is a pattern generator, called the **primary generator** of D. Additional generators, called **secondary generators**, are derived from $D.G$ or from other secondary generators during a dyad operation. For example, secondary 1-pattern generators, are derived from the universal 1-pattern generator G_{QU} during an exhaustive simple pattern search; these generators recognize all simple patterns that have more than 0 but less than $d + 1$ singleton sets where d is the depth of the search.

3.6.2. OPERATIONS AND ATTRIBUTED GRAMMAR

The following operation creates a dyad from an aligned sequence population and a generator G:

make_dyad(**seg_pop** *P*, **generator** *G*): Create and return **dyad** *D* = <*P*,*G*>.

The attributed grammar associated with this operation classifies dyads syntactically according to the type of generator used in its construction:

<dyad>$_{a1}$	→	<seg_pop>$_{a2}$ <1p_generator>$_{a3}$	[[a1 = <a2,a3>]]
<var_dyad>$_{a1}$	→	<seg_pop>$_{a2}$ <var_generator>$_{a3}$	[[a1 = <a2,a3>]]
<tab_dyad>$_{a1}$	→	<seg_pop>$_{a2}$ <tab_generator>$_{a3}$	[[a1 = <a2,a3>]]
<srch_dyad>$_{a1}$	→	<seg_pop>$_{a2}$ <univ_generator>$_{a3}$	[[a1 = <a2,a3>]]

Dyad operations using a single 1-pattern generator can reveal [1] which segments in the population match the specified pattern, [2] which sequence entities these segments were obtained from, and [3] the associated probability.

block(**dyad** *D*): Show the segments of block, B_Q, corresponding to the 1-pattern generator $D.G_Q$.

what(**dyad** *D*): Show the sequence entities that contain segments matching the 1-pattern generator $D.G_Q$.

evaluate(**dyad** *D*): Determine the probability $P(B_Q \mid D.P.B_U)$ where the block, B_Q, corresponds to the 1-pattern generator $D.G_Q$. Show the pattern Q, the number of segments matching B_Q, the number of sequence entities from which those segments were derived, the expected number of matching segments, and the probability from the cumulative binomial distribution function.

The eval_var( ) operation, using a variable generator, returns statistical information about closely related patterns:

eval_var(**dyad** D): Derive all secondary 1-pattern generators, G_Q, for patterns matching the primary variable generator, $D.G_v$, and sequentially call the operation evaluate(<$D.P$,G_Q>).

The cntable( ) operation, using a table generator, detects significant correlations between residues at different positions in a pattern.

cntable(**dyad** D): Show the contingency table and the probability, using Fisher's Exact Test (60), corresponding to the 2x2 table generator, $D.G_t$.

The search_blocks( ) operation reveals [1] which simple patterns are most significant for a given universal block of a segment population, and [2] which of a specified number of the most significant simple patterns can be merged (see Section 3.3) into a pattern having either of two residues at a single position.

search_blocks(**integer** d, **integer** n, **dyad** D, **real** $prob$): Call evaluate(<$D.P$,G_Q >) for up to n of the most significant d-residue patterns, Q, with an adjusted probability (see Section 4.2.1) greater than $prob$. Also call evaluate(<$D.P$,G_Q >) for all merged patterns derived from the n most significant 2-residue to d-residue patterns that have an adjusted probability greater than $prob$. The 2- to d-residue secondary generators (G_Q) are derived from the primary generator ($D.G_{Q_U}$) using generator operations.

A d-residue pattern is a simple pattern having d singleton sets. The operation monte_carlo( ) randomizes the population a specified number of times calling the search_blocks( ) operation each time; this allows empirical verification of the statistical calculations.

monte_carlo(**integer** *i*, **integer** *d*, **integer** *n*, **dyad** *D*, **real** *p*):

Randomize(*D.P*) *i* times calling the operation search_blocks(*d*, *n*, *D*, *p*) each time.

The following grammar with action symbols execute these operations; the search parameters *n* and *prob* are globally defined but can be redefined:

```
<command> → <dyad>a1                    { evaluate(a1) }
          | <var_dyad>a1                { eval_var(a1) }
          | <tab_dyad>a1                { cntable(a1) }
          | block <dyad>a1              { block(a1) }
          | search <integer>a1 <srch_dyad>a2
                                        { search_blocks(a1,n,a2,prob) }
          | rand <integer>a1 search <integer>a2 <srch_dyad>a3
                                        {monte_carlo(a1,a2,n,a3,prob)}
```

See Section 6.6 for examples of how command languge strings are used.

3.6.3. IMPLEMENTATION

The cntable( ) operation is implemented as follows. The 2x2 table for the table generator $D.G_t$ is obtained by calling table($D.G_t$) (see Section 3.3). Then, the value of each cell of the table is set to $|B_Q|$, where B_Q is the block for pattern Q corresponding to the 1-pattern generator G_Q for that cell. This is done in the following way. First, call the operation get_pattern(*row*,*column*,G_t) which will return the secondary 1-pattern generator G_Q that corresponds to that cell. Second, call the operation evaluate(<$D.P$,G_Q>) to determine the number of segments matching B_Q. After the table values are assigned, the table operation exact_test( ) is performed to determine the significance of correlation (see Section 6.3).

The search_blocks operation is implemented as follows. The search_blocks( ) algorithm (see Figure 3.6) acts as a driver for a depth first simple pattern search (see Figure 3.2). A min-max heap (61) is used to save the *n* most significant *d*-residue patterns found.

```
procedure search_blocks(integer depth, heap_size, dyad D, real p)
    integer k := D.P.k;
    block b[ ][ ], B_U := D.P.B_U;
    heap H := makeheap(heap_size);
    generator G := D.G_QU;
    b := elementaryblocks(D.P);
    dfs(b, B_U, B_U, G, 1.0, depth, 1, k, H);
    do minkey(H) < p and G := deletemin(H) ≠ null → evaluate(<D.P,G>); od;
end search_blocks;
```

Figure 3.6. Algorithm used to implement the search_blocks( ) operation. The merge( ) operation has been omitted for clarity.

```
procedure dfs(block b[ ][ ],B_Q,B_U, generator G_Q, real p, integer d, i, k,
                                                    modifies heap H)
  element r; real p'; block B_Q';
  do i ≤ k →
    for r ∈ Σ →
      B_Q' := b[r][i] ∩ B_Q;
      if | B_Q' | ≥ minblock →
          p' := p · | b[r][i] ∩ B_U | ÷ | B_U |;
          G_Q(i) := {{r}};
          if | B_Q' | − p' · | B_U | > 2 · √(| B_U | · (1−p')) →
                insertheap(G_Q, P( | B_U |, | B_Q' |, p'), H);
          fi;
          if d > 0 → dfs(b, B_Q', B_U, G_Q, p', d-1, i+1,k, H) fi;
      fi;
    rof;
    G_Q(i) := {Σ}; i := i + 1;
  od;
end dfs;
```

Figure 3.7. Algorithm used to implement the dfs( ) operation. The operation insertheap( ) is used to maintain a min-max heap (H) of the most significant patterns.

The minkey( ) heap operation returns the value of the minimum key for the heap. The operation evaluate( ) is called for patterns with a calculated probability greater than the specified cutoff.

The dfs( )[4] algorithm is given in Figure 3.7 and incorporates two features to improve its performance. First, the search tree is pruned by cutting off nodes with a block size less than a minimum value (*minblock*) which is globally defined (see Figure 3.8). Second, the cumulative binomial probability, which is used as a key to insert patterns into the heap, is calculated only if the observed size of a sequence block is more than two standard deviations above the mean. (If the heap is full then the insertheap( ) operation only inserts a pattern Q into the heap if its key is less than the maximum valued key; the pattern having the maximum valued key is deleted from the heap before such an insertion.) Implementation of the evaluate( ) and eval_var( ) operations is similar to the search_blocks( ) operation except that single recursive paths are taken.

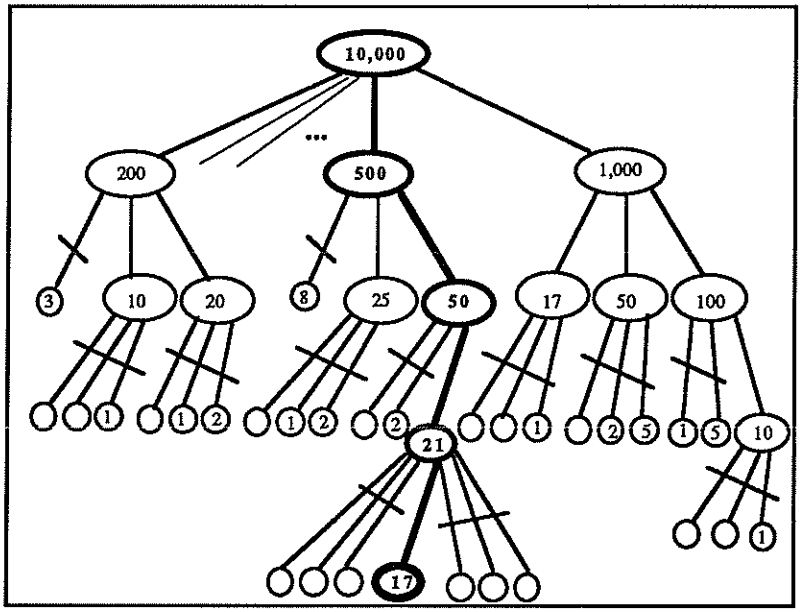


Figure 3.8. Pruning a pattern search tree. The search tree is pruned at nodes with block size less than a minimum cutoff (*minblock* = 9 in figure). This speeds up the search for pathways leading to true motifs.

[4]dfs stands for depth first search.

If the universal block $D.P.B_U$ consists of fully overlapping segments, then it is not necessary for the search_blocks( ) operation to search for all simple patterns but only for "left justified" simple patterns, i.e., patterns that start with a single residue set. A set of **fully overlapping segments** is defined by the following procedure. First, an infinite string of dummy residues 'ø' are attached to the end of each biological sequence entity $E \in D.P.\prod$ to create a modified set of entities $\prod'$. (A dummy residue is also added to all Σ-sets of the simple patterns but not to single residue sets.) Next, a set of overlapping segments is derived from $\prod'$ as was described in Section 3.1. Finally, segments consisting only of dummy residues are discarded to create the set of fully overlapping segments. For example, if there exists a sequence entity $E \in D.P.\prod$ such that

$$E.S = \text{"MTHFSGGKTRRTN"}$$

then the corresponding modified sequence entity $E' \in \prod'$ would have the sequence

$$E'.S = \text{"MTHFSGGKTRRTNøøøøøø.....\text{"}}$$

and the set of fully overlapping segments (of length = 7) would be:

MTHFSGG
THFSGGK
HFSGGKT
FSGGKTR
SGGKTRR
GGKTRRT
GKTRRTN
KTRRTNø
TRRTNøø
RRTNøøø
TNøøøøø
Nøøøøøø

The search_blocks( ) operation need only search such a set for "left justified" patterns because for any arbitrary pattern, Q, that begins with the Σ-set and that matches some segment in an arbitrary sequence entity $E' \in \prod'$,

```
Q:                  ..T.R.N
E'.S:       MTHFSGGKTRRTNøøø...
matching segment:       GKTRRTN
```

there always exists a "left justified" pattern that detects the same residues in the sequence.

```
Q:                  T.R.N..
E'.S:       MTHFSGGKTRRTNøøø...
matching segment:       TRRTNøø
```

Thus a more efficient form of the search_blocks( ) operation (not shown), that searches only for "left justified" patterns, is used when the universal block $D.P.B_U$ is a set of fully overlapping sequences. (Note that certain selection operations can produce a universal block that is not a set of fully overlapping sequences.)

Thus, the dyad abstract data type can be used to search for statistically significant simple patterns in a variety of segment populations (obtained by varying the value of $D.P.k$ and by selection operations on $D.P$). In addition, the simple patterns detected may be merged into patterns having either of two amino acids at a single position; this may provide opportunities to test for correlations between residues at different positions using the cntable( ) operation. Chapter 4 explores the potential of this model to detect patterns in a variety of aligned segment populations.

4. VERIFICATION OF THE METHOD

4.1. IMPLEMENTATION - THE ASSET PROGRAM

The model described in the previous chapter was implemented in the C programming language on a Sun4/260 under the UNIX operating system (62). The command language grammar was implemented using Lex (63) and Yacc (64); key operations were linked into the grammar through the use of action symbols. The program, called ASSET (Aligned Segment Statistical Evaluation Tool), was used to verify the method.

4.2. ASSET PERFORMANCE PROFILE

In this section the performance of ASSET is evaluated. Before proceeding, several terms, to be used throughout this section, are in need of clarification. The term "pattern probability" is used to denote the probability (under appropriate independence assumptions) of finding the observed number, or more, of segments in a population that match a specific pattern; this is obtained using the cumulative binomial distribution function as described in Section 3.1. The term "adjusted probability" requires a lengthier explanation. An aligned segment population is often searched using a large number, N, of different patterns, where N may be on the order of a million or so. The pattern probability p associated with each pattern, can be viewed as a random variable; the "adjusted probability" P_{adj}, associated to p and N, is the probability that during a search of N patterns, one or more of these N random variables is $\leq p$. P_{adj} can be computed from p and N using the cumulative binomial distribution function (Equation 3.1) (see next section). When $N = 1$ (i.e., a single pattern, selected in advance is tested) the pattern probability = the adjusted probability. In general, however, the adjusted probability is larger and measures the significance of a detected pattern given the size of the set of patterns which was tested.

4.2.1. STATISTICAL SIGNIFICANCE

A pattern is statistically significant (i.e., the null hypothesis is rejected) if its adjusted probability is less than some cutoff (called the level of significance). Conventionally (and in this study) the significance level is taken to be 0.01. If the adjusted probability is greater than 0.01 the null hypothesis cannot be rejected and no conclusion can be reached. It is important to realize that a pattern could have an adjusted probability > 0.01 and still be biologically significant. Moreover, statistical significance of a pattern does not reveal what that significance might mean. It could be that the pattern is due to chance (although this is unlikely) or that the pattern corresponds to an evolutionary conserved region without functional significance, or that the pattern is functionally important. Therefore, the statistical analysis should be seen only as an heuristic for formulating experimentally testable hypotheses and not as a means to determine the biological significance of a pattern.

As was mentioned above, the adjusted probability P_{adj} (i.e., the probability of detecting one or more patterns having the same or a smaller pattern probability) can be determined using the cumulative binomial distribution function where N is the number of patterns tested, and p is the pattern probability. An upper bound on P_{adj} can be obtained from the inequality

$$P_{adj} = \sum_{i=1}^{N} \binom{N}{i} p^i (1-p)^{N-i} \quad < \quad Np + (Np)^2 \tag{4.1}$$

(see Section 6.4 for a proof). For patterns with $P_{adj} \leq 0.01$, which are of interest here, Np is small and therefore provides an excellent approximation to the value of P_{adj}. Consequently, it is used throughout the studies described below.

The performance of the ASSET program was evaluated with reference to several parameters: the search depth, the search minimum block size (*minblock*), and the population segment length ($P.k$). The search depth, d, indicates the lowest level searched

during a depth first pattern search (see Figure 3.2 and 3.7); note that at a depth of d the search procedure tests for d-residue patterns, i.e., patterns matching only single residues at d positions and any residue at other positions. The parameter *minblock* is used to prune the search space (see Section 3.6 and Figure 3.8).

4.2.2. STATISTICAL VERIFICATION

In order to verify correct implementation of the statistical method and that the ASSET statistical model is accurate, Monte Carlo simulations were performed using the monte_carlo( ) operation described in Section 3.6.2. For patterns having small (pattern) probabilities, these probabilities were generally in agreement with empirical probabilities. Empirical probabilities were determined by the following procedure. [1] Select a probability value $0 < p << 1$ to be tested. [2] Generate a random aligned segment population using either the randomize( ) or the shuffle( ) operation. [3] Find the number of 2- and 3-residue patterns having a pattern probability $\leq p$ for that random segment population using the evaluate( ) operation. [4] Repeat steps 2 and 3 n times (each iteration is called a run). [5] Divide the total number of patterns having a pattern probability $\leq p$ by the total number of patterns tested, i.e., # patterns found ÷ (# patterns searched per run x n runs).

In one experiment, where 19 tRNA synthetases were randomized using the shuffle( ) operation, the empirically determined pattern probabilities were found to be slightly lower (more significant) than the calculated probabilities (Table 4.1). This may be partly due to the fact that sets of overlapping segments are not really independent. It is also partly explained by the fact that the cumulative binomial probability is a step function (see Figure 4.1). In any case, since the calculated probabilities appear to err slightly toward the conservative side, unwarranted rejection of the null hypothesis is unlikely (see Section 3.1).

Table 4.1. Calculated and empirically determined pattern probabilities. Empirical probabilities were based on about 36,000,000 patterns for the shuffle( ) operation (100 runs) and on about 4,890,000 patterns for the randomize( ) operation (5 runs) using either diresidue or monoresidue frequencies.

CALCULATED PROBABILITIES	EMPIRICAL PROBABILITIES		
	using shuffle( )	using randomize( ) w/ diresidue freq.	using randomize( ) w/ monoresidue freq.
$\leq 10^{-4}$	$\leq 10^{-4.4}$	not determined	$\leq 10^{-4.5}$
$\leq 10^{-5}$	$\leq 10^{-5.5}$	$\leq 10^{-4.7}$	$\leq 10^{-5.6}$
$\leq 10^{-6}$	$\leq 10^{-6.5}$	$\leq 10^{-5.3}$	$\leq 10^{-6.2}$
$\leq 10^{-7}$	$\leq 10^{-7.2}$	$\leq 10^{-5.8}$	not determined

In a second experiment, about 8,000 21-residue subsequences matching the seed pattern "`.....GK.....`" were obtained from among sequences in the PIR protein database (65) version 31. These subsequences were randomized using the random( ) operation. (Recall that the random( ) operation randomizes residues consistent with the position dependent diresidue frequencies obtained from the population.) In this experiment the empirically determined probabilities were found to be slightly higher (less significant) than the calculated probabilities (see Table 4.1). This could be due to a violation of the assumption that residues at different positions in the segments are statistically independent since the random segment population was created using diresidue frequencies. Also, since the ASSET method is designed to detect short patterns, it is not surprising that retaining diresidue frequencies should result in elevated empirical pattern probabilities, since short 2-residue patterns present in the original population would tend to be retained. (Randomizing the population using triresidue frequencies should produce an even more marked effect.) In order to investigate this further, the random( ) operation was modified to generate sequences based on position dependent monoresidue frequencies. Empirically determined probabilities using this modified operation were slightly lower

(more significant) than the calculated probabilities (Table 4.1). These results confirm that the ASSET statistical method, although appearing to error on the conservative side, is generally reliable.

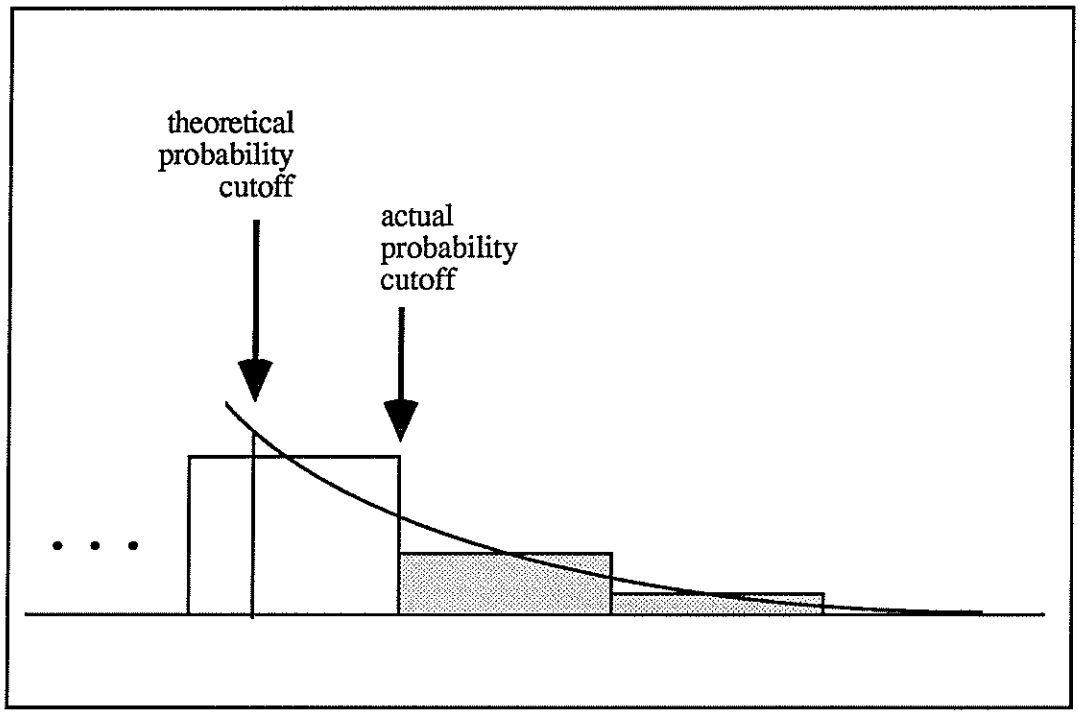


Figure 4.1. Discrepancy between theoretical and actual pattern probability cutoffs. Empirically determined pattern probabilities are lower than expected because the cumulative binomial probability is a step function. How this occurs can best be shown using an extreme example, where the pattern probability for a single pattern is determined using a large number of random segment populations that all have the same residue frequencies and number of segments. In this case, the values of N and p in the cumulative binomial probability function (Equation 3.1) are constant, so that the pattern probability for each segment population is calculated using the same distribution. (However, the number of matching segments will vary.) Consequently, only if the theoretical probability cutoff is on the edge of a step will it equal the actual cutoff; otherwise the actual cutoff will be at a smaller value and will lead to an empirical probability that appears to be lower than expected.

4.2.3. ASSET EXECUTION PERFORMANCE

In order to evaluate the execution performance of the ASSET program, experiments were performed using a group of tRNA synthetases. These proteins are, in general,

distantly related and have only two short conserved patterns, the 'HIGH' and 'KMSKS' motifs; therefore they act as a particularly stringent test of the sensitivity of the ASSET method. Closely related proteins where eliminated using the blast( ) operation with a BLAST score of 200 using a PAM 120 (7, 8) relatedness scoring matrix. The remaining 19 synthetases have a combined length of about 13,000 residues.

Experiments were done to investigate how the pattern search depth and the minimum block size (*minblock*) affect the execution time of and the number of patterns detected by the search_blocks( ) operation. In these experiments the minimum block size was varied from 1 to 9 using search depths of 3, 4, and 5 and with a segment length of 11. The value of *minblock* had essentially no effect on the run time for a 3-residue search, but did affect deeper searches; especially when *minblock* = 1 (Figure 4.2A).

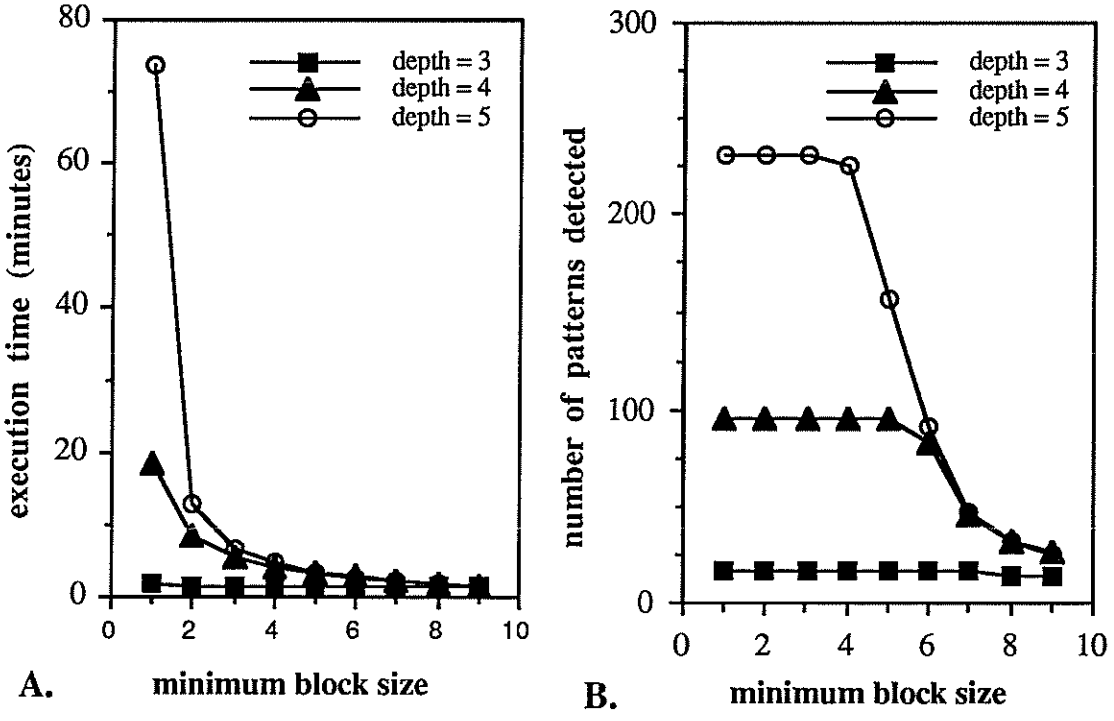


Figure 4.2. Effect of varying the minimum block size on the search_blocks( ) operation. A. Execution time on a Sun4/260. B. Number of patterns detected with adjusted probabilities < 0.01.

The increase in execution time needed for small values of *minblock* yielded no additional information; essentially the same number of patterns were detected for *minblock* values between 1 and 4 inclusive (Figure 4.2B).

In another set of experiments on the same data, the segment length was varied from 5 to 25, again using search depths of 3, 4 and 5 with a constant *minblock* value of 5. The number of patterns searched at level d is given by $N_d = \binom{n}{d} \times 20^d$, where n is the segment length and d is the depth of the search. Thus, the anticipated execution time is $O(n^d)$ where the number of segments is assumed to be constant. However, the actual execution time, which does increase significantly with increased segment length (Figure 4.3A), seems to have a much lower upper bound than $O(n^d)$ as revealed by nearly identical run times for 4- and 5-residue searches (as well as for a 10-residue search which is not shown); this is probably due to efficient pruning of the search tree.

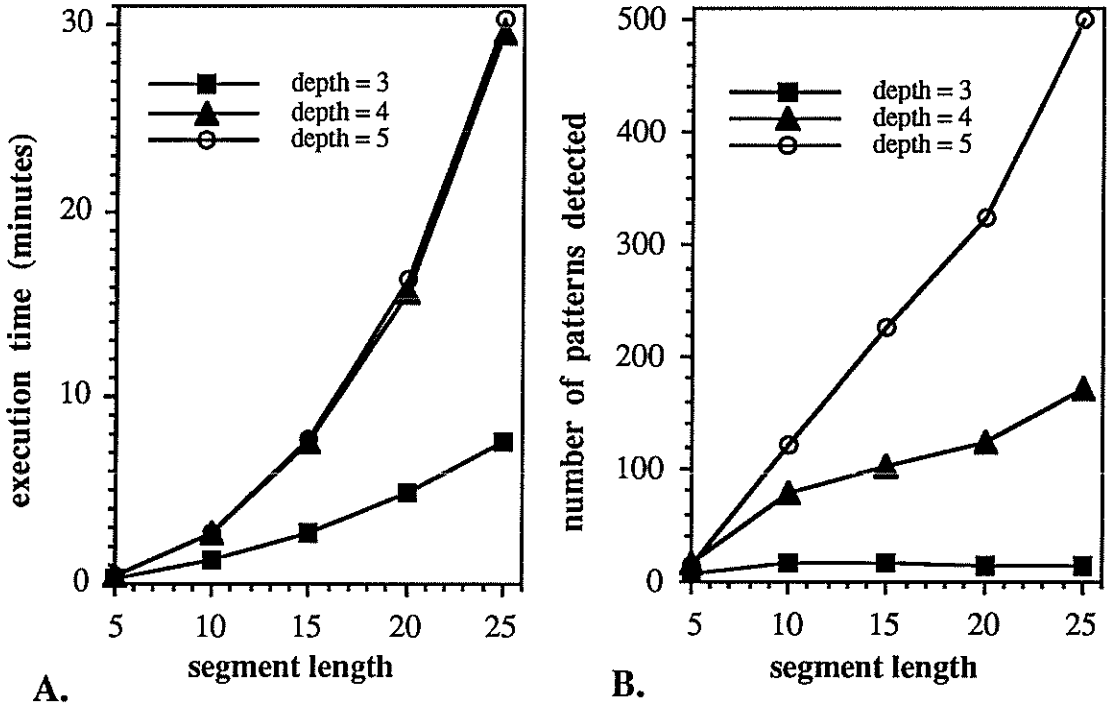


Figure 4.3. Effect of varying the segment length on the search_blocks( ) operation. A. Execution time on a Sun4/260. B. Number of patterns detected with adjusted probabilities < 0.01.

The number of patterns detected increased with increased segment length for 4- and 5-residue pattern searches but not for a 3-pattern search (Figure 4.3B). These results suggest that a *minblock* setting greater than 3 is best and that for 3-residue searches a modest segment length of 10 to 15 residues should be adequate; however, more tests on a variety of data are needed to verify this conclusion.

4.3. DETECTING MOTIFS AMONG DISTANTLY RELATED PROTEINS

In this section ASSET is compared with a program called MOTIF, which implements the method of Smith et al. (46). A strict comparison of ASSET and MOTIF is not possible since the parameters for the two programs are not identical. However, a rough comparison can be made by choosing parameters appropriately. The MOTIF program uses three parameters: the minimum number of proteins that must match a pattern before that pattern can be detected (designated here as the minimum match setting), the maximum number of internally repeated copies allowed in order to detect a pattern (the maximum internal repeats), and a parameter that determines the segment length. The ASSET minimum block size is roughly equivalent to MOTIF's minimum match setting, so these parameters were given equivalent values for comparable pattern searches. The programs were tested in their ability to detect motifs in the 19 tRNA synthetases mentioned above on a Sun4/260.

4.3.1. DETECTION OF MOTIFS IN 19 TRNA SYNTHETASES

The ASSET program was used for both a 3- and a 4-residue pattern search of the 19 tRNA synthetases.

4.3.1.1. Three-Residue Pattern Search

An exhaustive search by ASSET for all 3-residue patterns took 1.3 minutes on a Sun4/260 (minimum block size = 9; segment length = 11). The 'HIGH'-related pattern "`H.GH`" was detected in 16 proteins (adjusted probability = $10^{-13.7}$). This pattern along

with other 'HIGH'-related patterns, having adjusted probabilities < 0.0002, were detected in all 19 proteins. The 'KMSKS'-related pattern "KMS . ." was detected in 11 proteins (adjusted probability = $10^{-5.6}$). This pattern and other KMSKS-related patterns, having adjusted probabilities < 0.0008, were detected in 14 proteins. A total of 14 patterns related to the 'HIGH' and 'KMSKS' motifs had adjusted probabilities below the significance level ($p \leq 0.01$) (Figure 4.4).

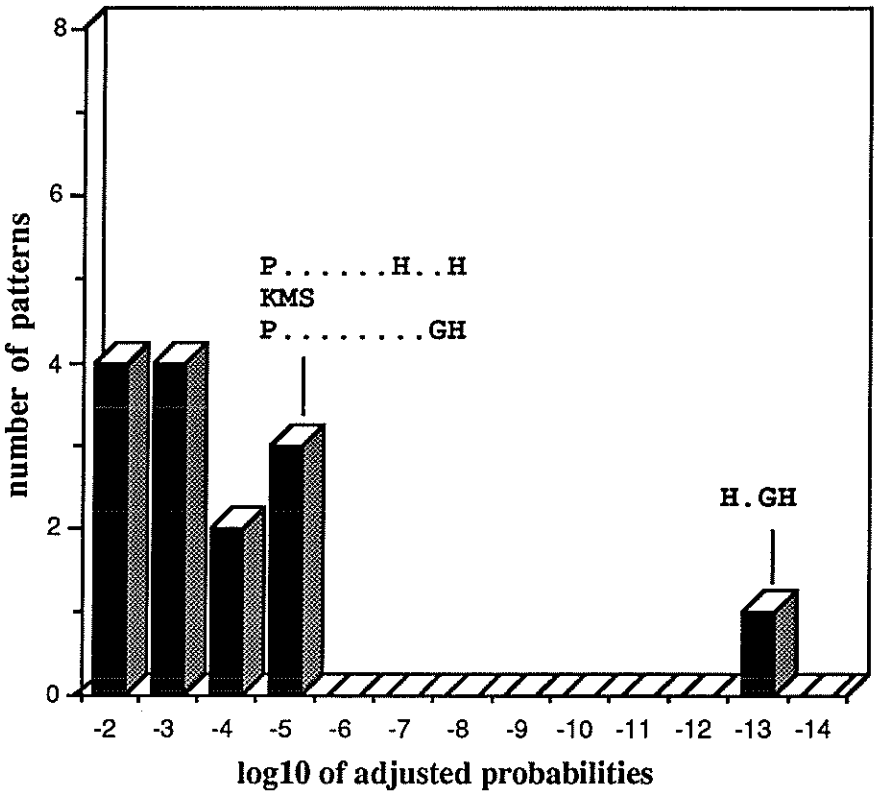


Figure 4.4. Detection of 'HIGH' and 'KMSKS' motifs by ASSET in a 3-residue pattern search of 19 tRNA synthetases (minimum block size = 9). Unlabeled patterns are also related to the 'HIGH' and 'KMSKS' motifs.

4.3.1.1. Four-Residue Pattern Search

Greater sensitivity in detecting the 'KMSKS' motif was achieved by searching for all 4-residue patterns (minimum block size = 5; segment length = 11) which took 2.65 minutes (Figure 4.5).

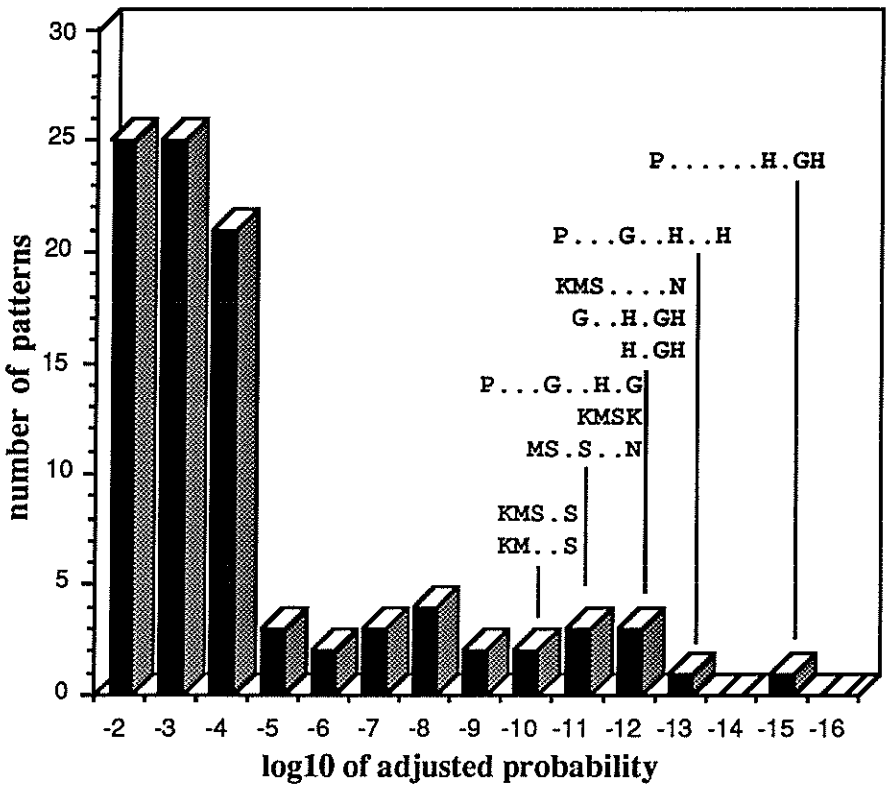


Figure 4.5. Detection of 'HIGH' and 'KMSKS' motifs by ASSET in a 4-residue pattern search of 19 tRNA synthetases (minimum block size = 5). Unlabeled patterns are also related to the 'HIGH' and 'KMSKS' motifs.

A number of 'KMSKS'-related patterns were detected in 11 proteins with adjusted probabilities $\leq 10^{-11.4}$ and in 18 proteins with adjusted probabilities ≤ 0.004. This search was of comparable sensitivity in detecting the 'HIGH' motif; a number of 'HIGH'-related patterns were detected in 16 proteins with adjusted probabilities $\leq 10^{-12}$ and in 19 proteins with adjusted probabilities ≤ 0.006. A total of 95 'HIGH'- and 'KMSKS'-related simple patterns were detected with adjusted probabilities below the significance level cutoff of 0.01. The same region of a sequence was often detected by several related patterns, for example, residues 92-102 of a *Neurospora* leucyl-tRNA synthetase, contain all five 'HIGH'-related patterns shown in Figure 4.5 (the sequence in this region is "PYPSGHLHLGH"). Although, in general, fewer proteins were detected for an arbitrary

4-residue pattern than for a 3-residue pattern, the total number of proteins matching a 'HIGH'- or 'KMSKS'-related pattern was equivalent or higher in the 4-residue search due to the greater number of patterns detected. Thus it seems that, at least for certain patterns and groups of proteins, a 4-residue search may be slightly more sensitive.

4.3.2. Comparisons with the MOTIF Program.

By contrast, a search using the MOTIF program (minimum match setting = 9; maximum internal repeats = 1; segment length = 11) detected the 'HIGH'-related pattern "H.GH" in 16 proteins but was unable to distinguish the 'KMSKS'-related pattern "K.S.S", present in 12 proteins, from random background patterns (Figure 4.6A). However, decreasing the maximum number of internal repeats to 0 resulted in detection of the pattern ".M.KS" in 11 proteins at an adjusted probability of about 10^{-2} as judged by the number of random background patterns having similar scores in 100 Monte Carlo simulations (Figure 4.6B). A 20 fold increase in the number of Monte Carlo simulations (from 5 to 100) increased the highest random background score by about 50 to 100 points (see Figure 4.6). This suggests that a random background score between 700 and 750 may be obtained after about 1,000 to 10,000 simulations. Therefore, the 'HIGH' motif seems to have an adjusted probability of about 10^{-3} to 10^{-4} for the statistic measured by the MOTIF method. Since ASSET detected the 'KMSKS'- and 'HIGH'-related motifs at adjusted probabilities of $10^{-5.6}$ and $10^{-13.7}$, respectively, the ASSET program seems to be at least 3-4 orders of magnitude more sensitive than MOTIF in distinguishing these patterns from random background. Increasing the segment length from 11 to 21 increased the 'KMSKS'-motif score from 694 (pattern: "M.KS") to 717 (pattern: "MS....N"). However, the larger segment length also caused random background scores to increase; 18 (length = 21) versus 9 (length = 11) random background patterns had scores $\geq$ 600 in 100 shuffled runs. A further increase in the segment length to 37 residues did not increase the 'KMSKS'-motif score.

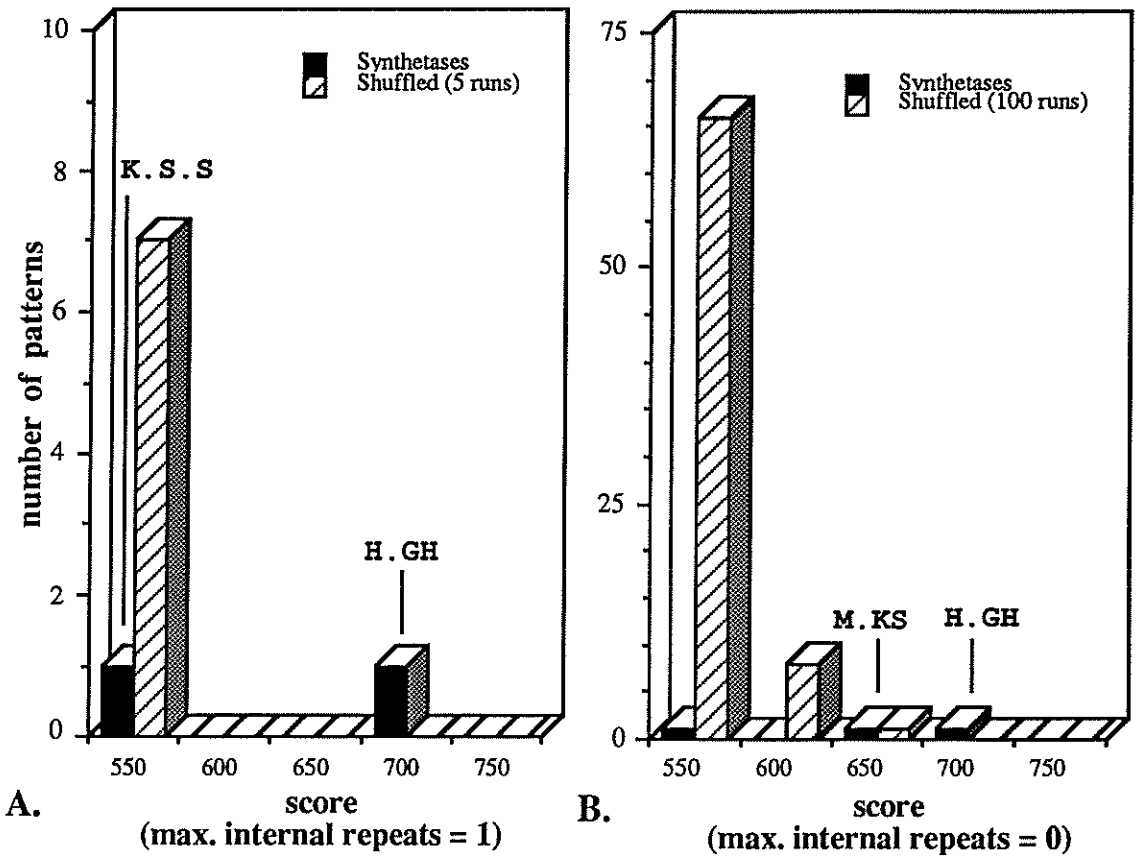


Figure 4.6. Detection of 'HIGH' and 'KMSKS' motifs by MOTIF in a pattern search of 19 tRNA synthetases (minimum matches = 9, segment length = 11). The number of patterns with scores ≥ 550 detected in Monte Carlo simulations (shuffled runs) are also shown; these simulations were performed in order to obtain an estimate of statistical significance. A. Maximum internal repeats = 1. B. Maximum internal repeats = 0.

ASSET's better sensitivity did not result in significantly poorer performance, despite the fact that ASSET, unlike MOTIF, uses no heuristics. Using a minimum block size and a minimum match setting of 9, ASSET's run times were about 40% longer than run times for MOTIF (Figure 4.7); at higher minimum match settings MOTIF performs even better. Nevertheless, this benefit is offset by the need to rerun MOTIF several times using various minimum match and maximum internal repeat settings to insure that nothing has been missed. To be safe, MOTIF could be run at a low minimum match and a high maximum

internal repeat setting, but this can produce adverse effects. When the minimum number of matches was decreased to 5, MOTIF's run time increased dramatically and was much worse than ASSET's (see Figure 4.7). A similar but less dramatic effect is seen when the maximum internal repeats setting is raised. This increase in run time may reflect MOTIF's need to calculate relatedness scores for many more random background patterns at the suboptimal settings. However, for detection of patterns present in a significant majority of proteins, MOTIF will save a few minutes of computation time over ASSET, but with a possible loss of sensitivity and with no direct measure of significance. Since MOTIF searches only for 3-residue patterns, a comparison with ASSET for 4-residue patterns could not be done.

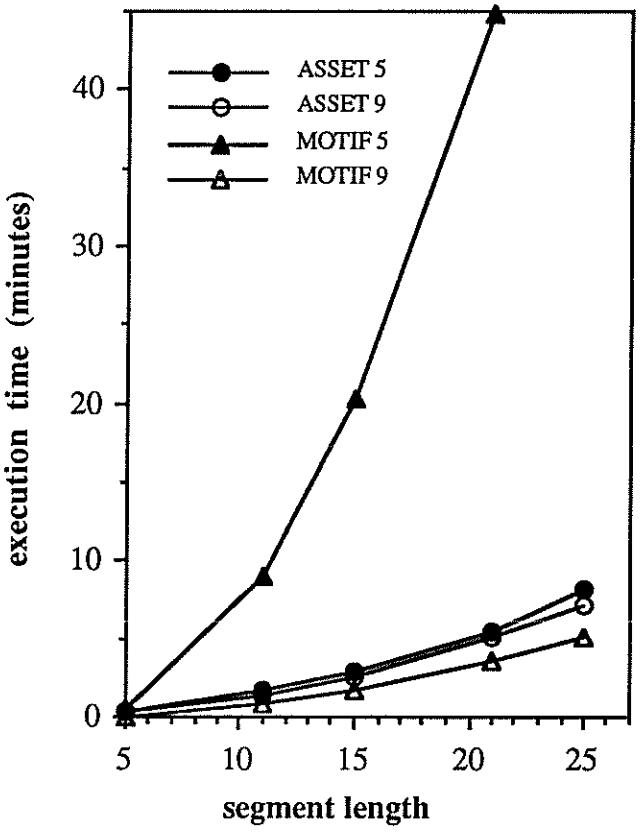


Figure 4.7. Execution times for ASSET and MOTIF at various segment lengths. MOTIF minimum match settings (or ASSET minimum block sizes) of 5 and 9 were used. MOTIF's maximum internal repeats setting was zero.

4.4. DETECTING PATTERNS HAVING EITHER OF TWO AMINO ACIDS AT A SINGLE POSITION

Protein sequence motifs are often found to have two or more related amino acids at a single position. Such motifs were detected using the merge( ) operation during a search of 15 adenine methylases. Several 3-residue patterns having either of two amino acids at a single position were detected including "{YV}..PP.", "..{ND}PP.", and "..{ND}P.Y" (Figure 4.8). Detection of these patterns offers an opportunity to test for correlations between residues at different positions.

PATTERN	OBS	(E)	EXP	log10(prob) PTRN	log10(prob) ADJST
PPY.....	14	(13)	0.48	-15.6	-9.6
D.....PP..	11	(10)	0.60	-10.3	-4.3
DP.Y.....	11	(10)	0.69	-9.7	-3.7
DPP.....	10	(9)	0.60	-9.0	-3.0
D......PY.	10	(9)	0.69	-8.5	-2.5
D.....P.Y.	10	(9)	0.69	-8.5	-2.5
D.PY.....	10	(9)	0.69	-8.5	-2.5
NPP..... D	15	(14)	1.30	-11.2	-3.6
N.PY..... D	16	(13)	1.40	-11.5	-3.9
Y..PP..... V	13	(11)	1.10	-9.9	-2.3

Figure 4.8. Patterns found by ASSET in 15 adenine methylases. Abbreviations: OBS, number of matching segments observed; (E), number of entities containing the pattern; EXP, number of matching segments expected; log10(prob)_PTRN, logarithm (base 10) of the probability of finding the observed number or more segments matching the pattern ; log10(prob)_ADJST, logarithm (base 10) of the adjusted probability.

4.5. TESTING FOR CORRELATIONS

The adenine methylases mentioned in the previous section have two chemically similar groups of amino acids at a position near the most significant pattern detected, "PPY". The residue at this variable position ("?..PPY") is either one of the aromatic amino acids,

tyrosine (Y) or phenylalanine (F), or one of the aliphatic amino acids, isoleucine (I), leucine (L) or valine (V) (see Figure 4.9). Is there a correlation between finding an aspartate (D) or asparagine (N) at position 3 and an amino acid from one of these groups at position 1 of the matching segments? (Only aspartate (D) or asparagine (N) occur at position 3.) When Fisher's exact test is performed on the table generator "{YF:ILV}.{D:N}PPY" a significant correlation is found ($p = 0.001$); an aromatic amino acid always occurs with aspartate and an aliphatic amino acid always occurs with asparagine (Table 4.2). However, since this test was suggested by looking at the data, this correlation is more likely to be due to a chance observation; consequently, further testing on an independent set of data is required to confirm these results.

PATTERN	OBS (E)	EXP	log10(prob) PTRN
I..PPY....	1 (1)	0.032	-1.5
L..PPY....	1 (1)	0.047	-1.3
F..PPY....	2 (2)	0.026	-3.5
Y..PPY....	8 (7)	0.023	-17.7
V..PPY....	2 (2)	0.028	-3.4

Figure 4.9. ASSET output for the eval_var(*D*) operation corresponding to the variable generator *D.G* = "?..PPY". The segment population (*D.P*) was derived from 15 adenine methylases. Abbreviations are as given in Figure 4.7.

Table 4.2. Contingency table corresponding to the table generator "{YF:ILV}.{D:N}PPY". The segment population was derived from 15 adenine methylases. Expected values are in parentheses.

	D	N	
ILV	0(2.9)	4(1.1)	4
YF	10(7.1)	0(2.9)	10
	10	4	14

Examination of the regions near these motifs reveals that, except for two homologous proteins (ECOP15BMO_1 and PP1MOD_1), these sequences are not particularly similar except in the vicinity of the motifs (see Table 4.3). Thus, it seems unlikely that this correlation is simply due to a close evolutionary relationship between subgroups of these sequences. One possible explanation for this correlation is that mutation of a residue at one position may require a compensating mutation at the other position in order to maintain protein function, i.e., there is an interaction between residues at the two positions. These results demonstrate that ASSET is able to detect correlations between sequence patterns.

Table 4.3. Sequence context for the "PPY" motif in 15 adenine methylases. Sequences containing the "{ILV}.NPPY" motif are above the double line while sequences containing the "{YF}.DPPY" motif are below it.

SEQUENCE	PROTEIN	LOCATION
LLWKGGKFDFIVGN PPY VVRPSGYKNDNRI	CHV1AMB3_1	(105-134)
LAPLEGQFDFVVGN PPY VRPELIPAPLLAE	PSEPAER7_1	(106-135)
LWEPGEAFDLILGN PPY GIVGEASKYPIHV	MTTA_THEAQ	(91-120)
IENYSPKYNKAILN PPY LKIAAKGRERALL	PROIRM_1	(139-168)
VNAYAEKVKMIYID PPY NTGKDGFVYNDDR	ECOP15BMO_1	(109-138)
VNAYAEKVNMIYID PPY NTGKDGFVYNDDR	PP1MOD_1	(109-138)
SLIEKVYGDILYID PPY NGRQYISNYHLLE	FVBFOKMR_1	(204-233)
NFSQLDQNDLVYCD PPY LITTGSYNDGNRG	FVBFOKMR_1	(534-563)
KDVKILDGDFVYVD PPY LITVADYNKFWSE	PT4T4G69_2	(157-186)
AIVDVRTGDFVYFD PPY IPLSETSAFTSYT	STRDPN2A_1	(180-209)
KTIPNESIDLIFAD PPY FMQTEGKLLRTNG	HEAMTEN_1	(22-51)
SKMKPESMDMIFAD PPY FLSNGGISNSGGQ	STRDPN2A_2	(22-51)
TIGMVNRDDVVYCD PPY IGRHVDYFNSWGE	PLBECORV_2	(179-208)
SMARADDASVVYCD PPY APLSATANFTAYH	DMA_ECOLI	(167-196)

4.6. DETECTING MOTIFS PRESENT IN ONLY A MINORITY OF PROTEINS IN A GROUP

ASSET was next tested for its ability to detect patterns present in only a small number of proteins in a group. This was done in two ways. First, the ASSET program was used to search a large group of functionally related proteins for minor motifs which had escaped detection by the MOTIF program. Second, a procedure was developed to search for

patterns among proteins in a large database where functional relationships are not known prior to analysis.

4.6.1. Finding Minor Patterns in 29 Reverse Transcriptases

Smith et al. (46) describe a search of 33 reverse transcriptases using the MOTIF program that yielded three motifs. 29 of these 33 proteins were retrieved from the protein databases (4 were not found, see Section 6.5) and analyzed using both ASSET and MOTIF. In an attempt to detect patterns occurring in a minority of the proteins, MOTIF minimum match settings between 8 and 10 were used (and from 0 to 3 internal repeats were allowed); this yielded only one additional pattern having a score ≥ 600 for a total of 4 patterns detected by MOTIF. In addition to detecting these patterns, ASSET found 7 other groups of patterns (a pattern group contains many related patterns); these occurred in a minor fraction of the 29 proteins (see Table 4.4). Thus ASSET appears to be more sensitive than MOTIF in detecting 'minority' patterns. Six of the 11 patterns, including the 4 detected by the MOTIF program, include amino acid residues which were found, by substitution mutations, to be functionally important in HIV transcriptases (66). This suggests that ASSET is locating patterns that point to key residues in these proteins.

Table 4.4. Patterns Detected in 29 Reverse Transcriptases. Twelve different groups of related patterns were detected by ASSET; only a few of the most significant patterns for each group are shown. The patterns detected by MOTIF are related to four of these groups. Known functionally important residues, as revealed by substitution mutations in HIV transcriptases (66), are bold. The numbers in parentheses (under the PROTEINS column) indicate the number of matching segments for internally repeated patterns.

PATTERN	log10 ADJUST. PROB.	PROTEINS	MOTIF	ASSET
Y.DD..	-12	25(27)	YES	YES
Y.DD.L	-14	15		
YMDD..	-20	13		
P**Q**G.....	-3	18(25)	YES	YES
.**Q**G...SP	-13	15		
P**Q**G....P	-11	14		
P**Q**G...SP	-25	13		
..R...D.R..N	-18	16	YES	YES
K.....D.R..N	-8	12		
K.R...D.R..N	-17	10		
D..D..F...L	-8	12	YES	YES
D..D..F.I..	-8	11		
D..D..F..P.	-7	10		
N...**DS.Y**	-9	11	NO	YES
.I..**DS.Y**	-4	9		
NI..**D..Y**	-5	9		
NI..**DS.Y**	-16	9		
..QK..G...**W**	-6	9	NO	YES
D.Q...G...**W**	-6	8		
QWPL....	-7	10	NO	YES
QWP....K	-6	9		
QWPL...K	-15	9		
WQ.D.T.	-13	11	NO	YES
W..D.TH	-12	10		
WQ.D.TH	-25	10		
A...V..C..C	-13	9	NO	YES
A..I...C..C	-12	9		
P.N......VE.	-5	10	NO	YES
P........VER	-5	10		
AFT.P	-5	10	NO	YES

4.6.2. Searching Large Databases for Related Proteins

The PIR protein database version 31 was searched for motifs using a general procedure which is described as follows. First, every subsequence in the protein database matching a "seed" pattern, e.g. two amino acids flanked by a number of unspecified residues, are selected. Second, similar subsequences are removed from the selected set using the blast( ) operation with a specific cutoff score. This eliminates the most closely related subsequences from the analysis. Third, the subsequence population is analyzed with ASSET for statistically significant patterns. (Low entropy sequences may first need to be removed from the population using the purgeH( ) operation.) Last, the proteins containing these patterns are obtained and analyzed by ASSET as a group to detect additional patterns. Thus, this procedure could detect motifs among proteins not previously known to be related. Collections of subsequences matching all two amino acid seed patterns could be generated from a large protein database in order to do an exhaustive search for patterns in the database.

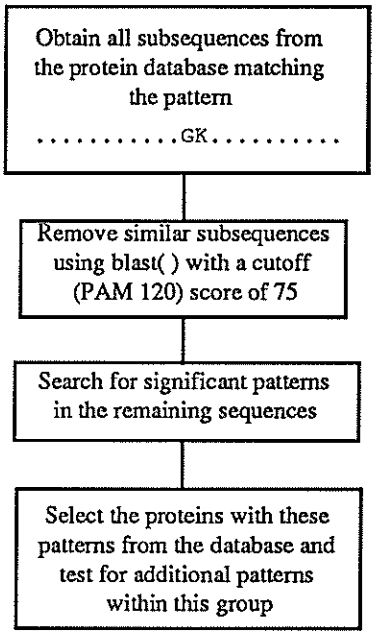


Figure 4.10. Procedure used to detect patterns in the PIR database.

This procedure was applied to sequences in the PIR protein database using the "seed" pattern "`.....GK.....`" (Figure 4.10). Similar sequences were removed using a (PAM 120) cutoff score of 75. ASSET detected a number of ATP/GTP-binding site motifs (3) in the database, some with very low adjusted probabilities (Figure 4.11). This demonstrates that ASSET is sensitive enough to detect patterns even among a large database of proteins. The last step of the procedure (i.e., selecting proteins with significant patterns for further testing) was not done.

.....**GK**..... (seed pattern)				
			log10(prob)	
PATTERN		OBS	EXP	PTRN	ADJST
.....G..G.**GK**S.....		55	5	-34.9	-26.5
	T				
.....G....**GK**ST.....		43	5	-23.9	-15.6
	T				
.....G.**GK**ST.....		37	4	-19.5	-11.1
	T				
...I.G..G.**GK**.....		37	5	-17.2	-8.9
L					
.....G.PG.**GK**.....		32	4	-15.4	-7.0
S					
...I.G....**GK**.T.....		34	5	-15.3	-6.9
L					
.....G..G.**GK**.....I..		33	5	-15.1	-6.7
	L				
.....GS**GK**T.....		33	5	-14.3	-5.9
T					
.....G....**GK**S.....L..		35	6	-14.0	-5.6
	T				
.....GS**GK**S.....		32	5	-13.5	-5.1
	T				
...L....G.**GK**S.....		34	6	-13.4	-5.0
	T				

Figure 4.11. Detection of the ATP/GTP-binding site motif A among sequences in the PIR protein database. (See text for details.)

4.7. DETECTING MOTIFS WHICH ARE INTERNALLY REPEATED

Since the ASSET method is not based on the number of proteins having a specific pattern but rather on the number of matching segments, it can detect internally repeated